



Professional

SQL Server™ 2005 Programming

Robert Vieira



Updates, source code, and Wrox technical support at www.wrox.com

Professional SQL Server™ 2005 Programming

Robert Vieira



Wiley Publishing, Inc.

Professional SQL Server™ 2005 Programming

Published by

Wiley Publishing, Inc.

10475 Crosspoint Boulevard

Indianapolis, IN 46256

www.wiley.com

Copyright © 2007 by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN-13: 978-0-7645-8434-3

ISBN-10: 0-7645-8434-0

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

1B/QW/RR/QW/IN

Library of Congress Cataloging-in-Publication Data:

Vieira, Robert.

Professional SQL server 2005 programming / Robert Vieira.

p. cm.

ISBN-13: 978-0-7645-8434-3 (paper/website)

ISBN-10: 0-7645-8434-0 (paper/website)

1. Client/server computing. 2. SQL server. I. Title.

QA76.9.C55V55 2006

005.2'768—dc22

2006023047

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4355, or online at <http://www.wiley.com/go/permissions>.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

For general information on our other products and services please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Trademarks: Wiley, the Wiley logo, Wrox, the Wrox logo, Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. SQL Server is a trademark of Microsoft Corporation in the United States and/or other countries. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Credits

Executive Editor

Bob Elliott

Senior Development Editor

Kevin Kent

Technical Editor

John Mueller

Production Editor

Pamela Hanley

Copy Editor

Foxxe Editorial Services

Editorial Manager

Mary Beth Wakefield

Production Manager

Tim Tate

Vice President and Executive Group Publisher

Richard Swadley

Vice President and Executive Publisher

Joseph B. Wikert

Graphics and Production Specialists

Denny Hager, Jennifer Mayberry, Barbara Moore, Lynsey Osborn, Heather Ryan, Alicia B. South, Ronald Terry

Quality Control Technician

John Greenough

Project Coordinators

Michael Kruzel, Kristie Rees

Proofreading and Indexing

Techbooks

About the Author

Experiencing his first infection with computing fever in 1978, **Robert Vieira** knew right away that this was something “really cool.” In 1980 he began immersing himself into the computing world more fully—splitting his time between building and repairing computer kits and programming in BASIC as well as Z80 and 6502 Assembly. In 1983, he began studies for a degree in Computer Information Systems, but found the professional mainframe environment too rigid for his tastes and dropped out in 1985 to pursue other interests. Later that year, he caught the “PC bug” and began the long road of programming in database languages from dBase to SQL Server. Rob completed a degree in business administration in 1990 and since has typically worked in roles that allow him to combine his unique knowledge of business and computing. Beyond his bachelor’s degree, he has been certified as a Certified Management Accountant; Microsoft Certified as a Solutions Developer (MCSD), Trainer (MCT), and Database Administrator (MCDBA); and even had a brief stint certified as an emergency medical technician (EMT).

Rob is currently self-employed as a high-level consultant specializing in architectural analysis, long-term planning, and product viability analysis and is recognized internationally for his expertise in database architecture and management. He has published four books on SQL Server development.

He resides with his daughters Ashley and Adrianna in Vancouver, Washington.

*This book is dedicated with all my heart to my children **Ashley** and **Addy**,
who put up with me “disappearing” into my home office during the
several months that I worked on this book. They provide the energy
that powers my writing, and I love them to no end. I only wish*

*Wrox would let me print a picture of the two women
in my life on the cover of this book rather than my ugly mug.*

Acknowledgments

Nearly six years have gone by since I finished my first book on SQL Server, and my how life has changed. It's been only a few months since I completed the new *Beginning SQL Server 2005 Programming* title that is something of a companion to this book, yet there remain quite a few people to thank in that time (and before for that matter).

I'll start with my kids, who somehow continue to be just wonderful even in the face of dad stressing out over this and that. It's nice to be able to tell my youngest that I'm finally "done with that book" that she has asked me the "when?" question about for over a year now (I'm not sure she has completely gotten the idea that there have been two this time around). She has been tremendously patient with me all during the development of this as well as the *Beginning* book. I think I'll miss that occasional time that she would come in and just sit and watch me write just to be with me. Having just watched my eldest graduate high school, I wonder where the years have gone. The "thank you's" definitely need to begin with those two.

You—the readers. You've written me mail and told me how I helped you out in some way. That was and continues to be the number one reason I find strength to write another book. The continued support of my *Professional* series titles has been amazing. We struck a chord—I'm glad. Here's to hoping we help make your SQL Server experience a little less frustrating and a lot more successful.

I also want to pay special thanks to several people past and present. Some of these are at the old Wrox Press and have long since fallen out of contact, but they remain so much of who I am as I writer that I need to continue to remember them. Others are new players for me, but have added their own stamp to the mix—sometimes just by showing a little patience:

Kate Hall—Who, although she was probably ready to kill me by the end of each of my first two books, somehow guided me through the editing process to build a better book each time. I have long since fallen out of touch with Kate, but she will always be the most special to me as someone who really helped shape my writing career. I will likely always hold this first "professional" dedication spot for her. Wherever you are Kate, I hope you are doing splendidly.

Adaobi Obi Tulton—Who has had enough of her own stresses this year, so my apologies go out to her for all the stress I've placed in her regarding delivery schedules. If I ever make it rich, I may hire Adaobi as my spiritual guide. While she can be high stress about deadlines, she has a way of displaying a kind of "peace" in just about everything else I've seen her do—I need to learn that.

Bob Elliott—Mostly just a "thanks for hanging in there"—he'll understand what I mean.

Kevin Kent—Who had to pick up in the middle and shepherd things along.

Dominic Shakeshaft—Who got me writing in the first place (then again, given some nights filled with writing instead of sleep lately, maybe it's not thanks I owe him).

Acknowledgments

Catherine Alexander—Who played Kate’s more than able-bodied sidekick for my first title and was central to round two. Catherine was much like Kate in the sense she had a significant influence on the shape and success of my first two titles.

John Mueller—Who had the dubious job of finding my mistakes. I’ve done tech editing myself, and it’s not the easiest job to notice the little details that were missed or are, in some fashion, wrong. It’s even harder to read someone else’s writing style and pick the right times to say, “You might want to approach this differently” and the right times to let it be. John did a terrific job on both counts.

There are not quite as many other players in this title as there have been in my previous titles, but this book has been in development for so long and touched enough people that I’m sure I’ll miss one or two—if you’re among those missed, please accept my humblest apologies and my assurance that your help was appreciated. That said, people who deserve some additional thanks (some of these go to influences from WAY back) include **Paul Turley**, **Greg Beamer**, **Itzik Ben-Gan**, **Kalen Delaney**, **Fernando Guerrero**, **Gert Drapers**, and especially **Richard Waymire**.

Contents

Acknowledgments	v
Introduction	xxiii
<hr/>	
Chapter 1: Being Objective: Re-Examining Objects in SQL Server	1
So, What Exactly Do We Have Here?	1
An Overview of Database Objects	2
The Database Object	2
The Transaction Log	5
The Most Basic Database Object: Table	5
Schemas	6
Filegroups	7
Diagrams	7
Views	7
Stored Procedures	9
User-Defined Functions	9
Users and Roles	10
Rules	10
Defaults	10
User-Defined Data Types	10
Full-Text Catalogs	11
SQL Server Data Types	11
NULL Data	14
SQL Server Identifiers for Objects	15
What Gets Named?	15
Rules for Naming	15
Summary	16
<hr/>	
Chapter 2: Tool Time	17
Books Online	18
The SQL Server Configuration Manager	19
Service Management	19
Network Configuration	20
The Protocols	21
On to the Client	23

Contents

The SQL Server Management Studio	24
Getting Started	25
Query Window	27
SQL Server Business Intelligence Development Studio	32
SQL Server Integration Services (SSIS)	32
Reporting Services	33
Bulk Copy Program (bcp)	33
SQL Server Profiler	33
sqlcmd	34
Summary	34
 Chapter 3: Basic T-SQL	 35
 The Basic SELECT Statement	 36
The SELECT Statement and FROM Clause	36
The JOIN Clause	38
The WHERE Clause	45
ORDER BY	49
Aggregating Data Using the GROUP BY Clause	52
Placing Conditions on Groups with the HAVING Clause	56
Outputting XML Using the FOR XML Clause	57
Making Use of Hints Using the OPTION Clause	57
DISTINCT	57
Adding Data with the INSERT Statement	59
The INSERT INTO . . . SELECT Statement	61
Changing What You've Got with the UPDATE Statement	62
The DELETE Statement	64
Exploring Alternative Syntax for Joins	66
An Alternative INNER JOIN	67
An Alternative OUTER JOIN	67
An Alternative CROSS JOIN	68
The UNION	69
Summary	72
 Chapter 4: Creating and Altering Tables	 73
 Object Names in SQL Server	 73
Schema Name (a.k.a. Ownership)	74
The Database Name	76
Naming by Server	76
The CREATE Statement	77
CREATE DATABASE	77
CREATE TABLE	82

The ALTER Statement	89
ALTER DATABASE	89
ALTER TABLE	92
The DROP Statement	95
Using the GUI Tool	96
Creating or Editing the Database	96
Creating and Editing Tables	97
Summary	99
Chapter 5: Reviewing Keys and Constraints	101
Types of Constraints	102
Domain Constraints	103
Entity Constraints	103
Referential Integrity Constraints	104
Constraint Naming	104
Key Constraints	105
PRIMARY KEY Constraints	106
FOREIGN KEY Constraints	108
UNIQUE Constraints	117
CHECK Constraints	118
DEFAULT Constraints	119
Defining a DEFAULT Constraint in Your CREATE TABLE Statement	120
Adding a DEFAULT Constraint to an Existing Table	121
Disabling Constraints	121
Ignoring Bad Data When You Create the Constraint	122
Temporarily Disabling an Existing Constraint	124
Rules and Defaults: Cousins of Constraints	126
Rules	126
Defaults	128
Determining Which Tables and Data Types Use a Given Rule or Default	129
Triggers for Data Integrity	129
Choosing What to Use	129
Summary	131
Chapter 6: Asking a Better Question: Advanced Queries	133
What Is a Subquery?	134
Building a Nested Subquery	135
Nested Queries Using Single Value SELECT Statements	136
Nested Queries Using Subqueries That Return Multiple Values	137
The ANY, SOME, and ALL Operators	138

Contents

Correlated Subqueries	139
How Correlated Subqueries Work	140
Correlated Subqueries in the WHERE Clause	140
Correlated Subqueries in the SELECT List	142
Derived Tables	144
The EXISTS Operator	146
Using EXISTS in Other Ways	147
Mixing Data Types: CAST and CONVERT	148
Using External Calls to Perform Complex Actions	150
Performance Considerations	151
JOINS vs. Subqueries vs. ?	152
Summary	153
 Chapter 7: Daring to Design	 155
Normalization 201	155
Where to Begin	156
Getting to Third Normal Form	157
Other Normal Forms	157
Relationships	158
Diagramming	159
A Couple of Relationship Types	160
The Entity Box	160
The Relationship Line	161
Terminators	162
Logical versus Physical Design	165
Purpose of a Logical Model	165
Parts of a Logical Model	166
Dealing with File-Based Information	168
Subcategories	171
Types of Subcategories	172
Keeping Track of What's What — Implementing Subcategories	173
Getting Physical — The Physical Implementation of Subcategories	175
Adding to Extensibility with Subcategories	176
Database Reuse	177
Candidates for Reusable Databases	177
How to Break Things Up	178
The High Price of Reusability	179
De-Normalization	179
Partitioning for Scalability	180

The SQL Server Diagramming Tools	181
Tables	183
Dealing with Constraints	185
Summary	187
Chapter 8: SQL Server — Storage and Index Structures	189
SQL Server Storage	189
The Database	189
The File	190
The Extent	190
The Page	191
Rows	193
Full-Text Catalogs	194
Understanding Indexes	194
To “B,” or Not to “B”: B-Trees	195
How Data Is Accessed in SQL Server	199
Index Types and Index Navigation	200
Creating, Altering, and Dropping Indexes	208
The CREATE INDEX Statement	208
Creating XML Indexes	214
Implied Indexes Created with Constraints	215
ALTER INDEX	215
DROP INDEX	218
Choosing Wisely: Deciding What Index Goes Where and When	218
Selectivity	218
Watching Costs: When Less Is More	219
Choosing That Clustered Index	219
Column Order Matters	222
Dropping Indexes	222
Use the Database Engine Tuning Advisor	222
Maintaining Your Indexes	223
Fragmentation	223
Identifying Fragmentation	224
Summary	228
Chapter 9: Views	231
Simple Views	232
More Complex Views	233
Using a View to Change Data — Before INSTEAD OF Triggers	236
Editing Views with T-SQL	237

Contents

Dropping Views	238
Auditing: Displaying Existing Code	238
Protecting Code: Encrypting Views	239
About Schema Binding	241
Making Your View Look Like a Table with VIEW_METADATA	241
Indexed (Materialized) Views	242
Partitioned Views	244
Summary	244
 Chapter 10: Scripts and Batches	 247
 Script Basics	 248
The USE Statement	248
Declaring Variables	249
Using @@IDENTITY	252
Using @@ROWCOUNT	252
Batches	253
Errors in Batches	255
When to Use Batches	256
SQLCMD	259
Dynamic SQL: Generating Your Code on the Fly with the EXEC Command	260
The Gotchas of EXEC	262
Control-of-Flow Statements	265
The IF . . . ELSE Statement	266
The CASE Statement	270
Looping with the WHILE Statement	275
The WAITFOR Statement	277
TRY/CATCH Blocks	277
Summary	280
 Chapter 11: Getting Procedural: Stored Procedures and User-Defined Functions	 281
 Creating the Sproc: Basic Syntax	 282
An Example of a Basic Sproc	282
Changing Stored Procedures with ALTER	283
Dropping Sprocs	283
Parameterization	284
Declaring Parameters	284
Creating Output Parameters	285
Confirming Success or Failure with Return Values	288
How to Use RETURN	288

Dealing with Errors	290
The Way We Were	291
Manually Raising Errors	296
Adding Your Own Custom Error Messages	299
What a Sproc Offers	301
Creating Callable Processes	301
Using Sprocs for Security	301
Sprocs and Performance	302
Extended Stored Procedures (XPs)	304
A Brief Look at Recursion	305
User-Defined Functions (UDFs)	307
What a UDF Is	308
UDFs Returning a Scalar Value	308
UDFs That Return a Table	310
Understanding Determinism	316
Debugging	317
Setting Up SQL Server for Debugging	318
Starting the Debugger	318
Parts of the Debugger	321
Using the Debugger after It's Started	322
Summary	327
Chapter 12: Transactions and Locks	329
Transactions	329
BEGIN TRAN	331
COMMIT TRAN	331
ROLLBACK TRAN	331
SAVE TRAN	331
How the SQL Server Log Works	336
Using the CHECKPOINT Command	337
CHECKPOINT on Recovery	337
At Normal Server Shutdown	338
At a Change of Database Options	338
When the Truncate on Checkpoint Option Is Active	338
When Recovery Time Would Exceed the Recovery Interval Option Setting	339
Failure and Recovery	339
Implicit Transactions	340
Locks and Concurrency	341
What Problems Can Be Prevented by Locks	342
Lockable Resources	346
Lock Escalation and Lock Effects on Performance	346

Contents

Lock Modes	347
Lock Compatibility	349
Specifying a Specific Lock Type — Optimizer Hints	349
Setting the Isolation Level	353
READ COMMITTED	353
READ UNCOMMITTED	354
REPEATABLE READ	354
SERIALIZABLE	355
Dealing with Deadlocks (a.k.a. “A 1205”)	355
How SQL Server Figures Out There’s a Deadlock	356
How Deadlock Victims Are Chosen	356
Avoiding Deadlocks	356
Summary	358
Chapter 13: Triggers	361
What Is a Trigger?	362
ON	363
WITH ENCRYPTION	363
The FOR AFTER versus the INSTEAD OF Clause	364
WITH APPEND	366
NOT FOR REPLICATION	367
AS	367
Using Triggers for Data Integrity Rules	367
Dealing with Requirements Sourced from Other Tables	367
Using Triggers to Check the Delta of an Update	369
Using Triggers for Custom Error Messages	371
Other Common Uses for Triggers	372
Updating Summary Information	372
Feeding Data into De-normalized Tables for Reporting	372
Setting Condition Flags	373
Other Trigger Issues	375
Triggers Can Be Nested	376
Triggers Can Be Recursive	376
Debugging Triggers	376
Triggers Don’t Get in the Way of Architecture Changes	377
Triggers Can Be Turned Off without Being Removed	377
Trigger Firing Order	378
INSTEAD OF Triggers	380
INSTEAD OF INSERT Triggers	381
INSTEAD OF UPDATE Triggers	384
INSTEAD OF DELETE Triggers	384

IF UPDATE() and COLUMNS_UPDATED	386
The UPDATE() Function	386
The COLUMNS_UPDATED() Function	386
Performance Considerations	388
Triggers Are Reactive Rather Than Proactive	388
Triggers Don't Have Concurrency Issues with the Process That Fires Them	389
Keep It Short and Sweet	389
Don't Forget Triggers When Choosing Indexes	389
Try Not to Roll Back within Triggers	389
Dropping Triggers	390
Debugging Triggers	390
Summary	392
 Chapter 14: Nothing But NET!	 393
Assemblies 101	394
Compiling an Assembly	394
Uploading Your Assembly to SQL Server	397
Creating Your Assembly-Based Stored Procedure	398
Creating Scalar User-Defined Functions from Assemblies	400
Creating Table-Valued Functions	403
Creating Aggregate Functions	407
Creating Triggers from Assemblies	412
Custom Data Types	417
Creating Your Data Type from Your Assembly	418
Accessing Your Complex Data Type	418
Dropping Data Types	419
Summary	419
 Chapter 15: SQL Cursors	 421
What Is a Cursor?	421
The Lifespan of a Cursor	422
Types of Cursors and Extended Declaration Syntax	427
Scope	428
Scrollability	432
Cursor Types	434
Concurrency Options	447
Detecting Conversion of Cursor Types: TYPE_WARNING	450
FOR <SELECT>	452
FOR UPDATE	452
Navigating the Cursor: The FETCH Statement	452

Contents

Altering Data within Your Cursor Summary	453
	456
Chapter 16: XML Integration	457
The XML Data Type	458
Defining a Column as Being of XML Type	458
XML Schema Collections	460
Creating, Altering, and Dropping XML Schema Collections	461
XML Data Type Methods	463
Enforcing Constraints beyond the Schema Collection	469
Retrieving Relational Data in XML Format	469
The FOR XML Clause	469
OPENXML	493
A Quick (Very Quick) Reminder of XML Indexes	497
HTTP Endpoints	497
Security	498
HTTP Endpoint Methods	499
Creating and Managing a HTTP Endpoint	499
Closing Thoughts	500
Summary	501
Chapter 17: Reporting for Duty, Sir!	503
Reporting Services 101	504
Building Simple Report Models	504
Data Source Views	507
Report Creation	512
Report Server Projects	517
Deploying the Report	522
Summary	523
Chapter 18: Buying in Bulk: the Bulk Copy Program (BCP) and Other Basic Bulk Operations	525
BCP Utility	526
BCP Syntax	526
BCP Import	531
BCP Export	534
Format Files	536
When Your Columns Don't Match	538
Using Format Files	540
Maximizing Import Performance	541

BULK INSERT	541
OPENROWSET (BULK)	542
ROWS_PER_BATCH	543
SINGLE_BLOB, SINGLE_CLOB, SINGLE_NCLOB	543
Summary	543
 Chapter 19: Getting Integrated	 545
 Understanding the Problem	 545
 An Overview of Packages	 546
Tasks	548
The Main Window	551
Solution Explorer	552
The Properties Window	552
 Building a Simple Package	 552
 Executing Packages	 560
Using the Execute Package Utility	560
Executing within Management Studio	562
 Summary	 563
 Chapter 20: Replication	 565
 Replication Basics	 566
Considerations When Planning for Replication	566
Replication Roles	568
Subscriptions	569
Types of Subscribers	570
Filtering Data	570
 Replication Models	 570
Snapshot Replication	571
Merge Replication	574
Transactional Replication	577
Immediate-Update Subscribers	580
Mixing Replication Types	581
 Replication Topology	 581
Simple Models	581
Mixed Models	584
 Planning for Replication	 587
Data Concerns	587
Mobile Devices	588
 Setting Up Replication in Management Studio	 588
Configuring the Server for Replication	588
Configuring a Publication	592

Contents

Setting Up Subscribers (via Management Studio)	598
Using Our Replicated Database	603
Replication Management Objects (RMO)	605
Summary	606
Chapter 21: Looking at Things in Full: Full-Text Search	607
Full-Text Search Architecture	608
Setting Up Full-Text Indexes and Catalogs	610
Enabling Full-Text for Your Database	610
Creating, Altering, Dropping, and Manipulating a Full-Text Catalog	611
Creating, Altering, Dropping, and Manipulating Full-Text Indexes	614
Creating Full-Text Catalogs Using the Old Syntax	619
Old Syntax for Indexes	621
More on Index Population	622
Full-Text Query Syntax	624
CONTAINS	624
FREETEXT	626
CONTAINSTABLE	626
FREETEXTTABLE	628
Dealing with Phrases	628
Booleans	629
Proximity	629
Weighting	630
Inflectional	631
Noise Words	631
Summary	632
Chapter 22: Security	633
Security Basics	634
One Person, One Login, One Password	634
Password Expiration	635
Password Length and Makeup	637
Number of Tries to Log In	637
Storage of User and Password Information	637
Security Options	638
SQL Server Security	639
Creating and Managing Logins	640
Windows Integrated Security	646

User Rights	647
Granting Access to a Specific Database	647
Granting Object Permissions within the Database	648
User Rights and Statement-Level Permissions	654
Server and Database Roles	655
Server Roles	657
Database Roles	658
Application Roles	661
Creating Application Roles	662
Adding Permissions to the Application Role	662
Using the Application Role	662
Getting Rid of Application Roles	663
More Advanced Security	664
What to Do about the guest Account	664
TCP/IP Port Settings	664
Don't Use the sa Account	665
Keep xp_cmdshell under Wraps	665
Don't Forget Views, Stored Procedures, and UDFs as Security Tools	665
Certificates and Asymmetric Keys	666
Certificates	667
Asymmetric Keys	667
Summary	667
Chapter 23: Playing a Good Tune: Performance Tuning	669
When to Tune	670
Index Choices	671
Check the Index Tuning Tool in the Database Tuning Advisor	672
Client- vs. Server-Side Processing	673
Strategic De-Normalization	674
Routine Maintenance	674
Organizing Your Sprocs Well	675
Keeping Transactions Short	675
Using the Least Restrictive Transaction Isolation Level Possible	675
Implementing Multiple Solutions if Necessary	675
Avoiding Cursors if Possible	676
Uses for Temporary Tables	677
Sometimes, It's the Little Things	677
Hardware Considerations	678
Exclusive Use of the Server	679
I/O vs. CPU Intensive	679
OLTP vs. OLAP	684

Contents

On-Site vs. Off-Site	684
The Risks of Being Down	685
Lost Data	685
Is Performance Everything?	685
Driver Support	686
The Ideal System	686
Troubleshooting	686
The Various Showplans and STATISTICS	687
The Database Consistency Checker (DBCC)	692
The Query Governor	692
The SQL Server Profiler	693
The Performance Monitor (Perfmon)	696
Summary	698
Chapter 24: Administrator	699
Scheduling Jobs	700
Creating an Operator	701
Creating Jobs and Tasks	704
Backup and Recovery	722
Creating a Backup — a.k.a. “A Dump”	722
Recovery Models	728
Recovery	729
Index Maintenance	733
ALTER INDEX	734
Index Name	734
Table or View Name	734
REBUILD	734
DISABLE	735
REORGANIZE	735
Archiving of Data	736
Summary	736
Chapter 25: SMO: SQL Management Objects	739
The History of SQL Server Management Object Models	740
SQL Distributed Management Objects	740
SQL Namespaces	740
Windows Management Instrumentation	741
SMO	741
The SMO Object Model	742

Walking through Some Examples	744
Getting Started	744
Creating a Database	745
Creating Tables	746
Dropping a Database	750
Backing Up a Database	750
Scripting	752
Pulling It All Together	753
Summary	758
Appendix A: System Functions	761
Appendix B: Connectivity	815
Appendix C: Getting Service	825
Index	843

Introduction

It's been a long road indeed. The wait for SQL Server 2005 has been the longest drought between versions since SQL Server was first introduced in the late 1980s. Even the complete rewrite of SQL Server that was accomplished with version 7.0 took only 3 1/2 years (we've been waiting on SQL Server 2005 for over five years).

Some things are, however, worth waiting for, and SQL Server 2005 falls squarely in that camp. The number and importance of new or rewritten features is almost staggering. This book is, however, about much more than just "what's new?"—it is about understanding in a very broad way a product that has grown to one of the largest, most diverse products in the marketplace. Perhaps even more importantly, it is about understanding how to develop systems and applications that both meet your performance needs and store your data in a fashion that maintains integrity of the data while keeping it reasonably usable.

For those of you that have read the previous versions of this book, there have been a few changes this time around. Specifically, the real "beginning" level discussion has been moved into its own Wrox book (*Beginning SQL Server 2005 Programming*). The beginning topics are still here for sake of completeness, but they have been condensed to be in more of a review model. Why? Well, in the previous version, we had a problem where the size of the book had grown to a point where it could not get any larger and still fit within bindery limits (in short, it wouldn't have fit in a single cover). Something had to be done, so we made the choice to split it up a bit to allow room for growth in this and future versions. All the beginning materials are still there in a more compact form, but at a much faster pace and, therefore, perhaps more suited for review than for learning from scratch.

Other than that, this book maintains most of the style and variety of content it has always had. We go from the beginning to the relative end. We cover most of the add-on services, as well as advanced programming constructs (such as the highly touted .NET assemblies) and some of the supporting object models (which allow for management of your SQL Server and its various engines).

Version Issues

This book is written for SQL Server 2005. It does, however, maintain roots going back a few versions and keeps a sharp eye out for backward-compatibility issues with SQL Server 2000 and even 7.0. Version 6.5 is old enough now that little to no time is spent on it except in passing. (In short, 6.5 has been out of date for nearly seven years as I write this and hasn't been supported in quite some time—come into the twenty-first century folks!).

If you're asking yourself, "I have the 2000 book, why should I buy a 2005 edition—has it changed that much?" let me say this:

SQL Server 2005 brings in a host of new features and a ton of refinements versus what was available on the release of SQL Server 2000. Data Transformation Services has been completely rewritten and renamed to SQL Server Integration Services. The Analysis Manager has also been completely rewritten.

Introduction

A new Reporting Services engine has been added (though some of that functionality has been downloadable for a while now). A new unified design environment (“new” is relative here since it’s really just utilizing Visual Studio) has been added for all three of these tools. There is the new Notification Services engine, which allows you to subscribe to customizable events on your server or even to specific data activity. The full-text engine has been more closely integrated into the core database engine. That’s already a lot of stuff to take in, and those are just the service-level changes!

.NET language integration, a real error handler for T-SQL, PIVOT tables, XML as an indexable, core data type—the list of changes in the core database environment is almost staggering. So, with the 2000 book, I probably would have only told you to update from the 7.0 book if you needed the new feature area. This time the changes are too wide ranging (they ought to be after five years between releases), so, yes, you probably should update your copy of this book!

Who This Book Is For

This book assumes that you have some existing experience with SQL Server and are at an intermediate to advanced level. Furthermore, the orientation of the book is highly developer focused.

While the first several chapters cover what are relatively beginner topics, they do so in a somewhat accelerated fashion and are intended more as a reference—something to look back on when you forget (time does that to us all). The advanced reader may want to skip as far as to Chapter 7 or 8. Those who are already “expert” in SQL Server may want to pay particular attention to the chapters on .NET and all of the chapters beyond Chapter 14 or so.

What This Book Covers

This book is about SQL Server. More specifically, it is oriented around *developing* on SQL Server. Most of the concepts are agnostic to what client language you use, though the examples that leverage a client language generally do so in C# (some are shown in more than one language).

For those of you migrating from earlier versions of SQL Server, some of the “gotchas” that inevitably exist any time a product versions are discussed to the extent that they seem to be a genuinely relevant issue.

How This Book Is Structured

As is the case for all my books, this book takes something of a *laissez faire* writing style. I roam around a bit within a relatively loose structure. Each chapter begins with an explanation of the things to be covered in that chapter, and then I go through those items. Within each topic covered, some background is provided, and then I work through an example if appropriate. Examples are generally created to be short, yet still quickly get at several of the concepts you need for whatever topic is being covered at the time.

In terms of order of coverage, even though this is a “professional” class book, I start by moving at a rapid pace through some beginning concepts to reinforce foundation concepts and cover any changes for the current version of SQL Server; I then move quickly move through some intermediate concepts and on to the more advanced or external concepts in SQL Server.

Feedback

I've tried, as best as possible, to write this book as though we were sitting down next to each other going over this stuff. I've made a concerted effort to keep it from getting "too heavy," while still maintaining a fairly quick pace. I'd like to think that I've been successful at it, but I encourage you to e-mail me and let me know what you think one way or the other. Constructive criticism is always appreciated and can only help future versions of this book. You can contact me directly by e-mail (robv@professionalsql.com) or on my Web site (professionalsql.com).

If you find an errata item in the book, I ask that you submit that errata to Wrox via the procedure outlined later in this introduction (they are in charge of cataloging that stuff).

Also, please understand I can't guarantee a response to every question—though I do try—so, for general questions about SQL that are not specific to the book, consider using the SQL Server newsgroups on Usenet or one of the many SQL Server portals that are out there.

What You Need to Use This Book

While much of the code in this book can be done using the free version of SQL Server (and you should have applied at least Service Pack 1), many of the examples utilize SQL Server Management Studio, and the screen images provided often are based on the assumption that you have a full license available. In addition to SQL Server, the harder-core programming portions of the book occasionally assume that you have Visual Studio 2005 or higher available. While Visual Studio is not required for the fundamentals of the book, you will need it to do much with some of the debugging discussion as well as all of the .NET and SMO discussion.

Last, but not least, you will need your SQL Server running on Windows Server 2003 or later to be able to utilize every feature we discuss in the book (though the vast majority can be seen just fine on Windows XP).

Conventions

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

Boxes like this one hold important, not-to-be-forgotten information that is directly relevant to the surrounding text.

Tips, hints, tricks, and asides to the current discussion are offset and placed in italics like this.

Introduction

As for styles in the text:

- We *highlight* new terms and important words when we introduce them.
- We show keyboard strokes like this: Ctrl+A.
- We show filenames, URLs, and code within the text like so: `persistence.properties`.
- We present code in two different ways:

In code examples we highlight new and important code with a gray background.

The gray highlighting is not used for code that's less important in the present context, or has been shown before.

Source Code

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. All of the source code used in this book is available for download at www.wrox.com. Once at the site, simply locate the book's title (either by using the Search box or by using one of the title lists) and click the Download Code link on the book's detail page to obtain all the source code for the book.

Because many books have similar titles, you may find it easiest to search by ISBN; this book's ISBN is 0-7645-8434-0 (changing to 978-0-7645-8434-3 as the new industry-wide 13-digit ISBN numbering system is phased in by January 2007).

Once you download the code, just decompress it with your favorite compression tool. Alternately, you can go to the main Wrox code download page at www.wrox.com/dynamic/books/download.aspx to see the code available for this book and all other Wrox books.

Errata

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, like a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata you may save another reader hours of frustration and at the same time you will be helping us provide even higher-quality information.

To find the errata page for this book, go to www.wrox.com and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page you can view all errata that has been submitted for this book and posted by Wrox editors. A complete book list including links to each book's errata is also available at www.wrox.com/misc-pages/booklist.shtml.

If you don't spot "your" error on the Book Errata page, go to www.wrox.com/contact/techsupport.shtml and complete the form there to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

p2p.wrox.com

For author and peer discussion, join the P2P forums at p2p.wrox.com. The forums are a Web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At <http://p2p.wrox.com> you will find a number of different forums that will help you not only as you read this book but also as you develop your own applications. To join the forums, just follow these steps:

- 1.** Go to p2p.wrox.com and click the Register link.
- 2.** Read the terms of use and click Agree.
- 3.** Complete the required information to join as well as any optional information you wish to provide and click Submit.
- 4.** You will receive an e-mail with information describing how to verify your account and complete the joining process.

You can read messages in the forums without joining P2P but in order to post your own messages, you must join.

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the Web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

Being Objective: Re-Examining Objects in SQL Server

Well, here we are. We're at the beginning, but, if you're someone who's read my Professional level titles before, you'll find we're not quite at the same beginning that we were at in previous editions. SQL Server has gotten just plain too big, and so the "Pro" is going to become a little bit more "Pro" in level right from the beginning.

So, why am I still covering objects first then? Well, because I think a review is always in order on the basics, because you may need the see what's changed versus previous editions, and, last but not least, I still want this book to retain some of its use as a reference (I hate having to use 25 books to get all the info that seems like it should have been in just one).

With this in mind, I'm going to start off the same way I always start off—looking at the objects available on your server. The difference is that I'm going to assume that you already largely know this stuff, so we're going to move along pretty quickly and make a rather short chapter out of this one.

So, What Exactly Do We Have Here?

What makes up a database? Data for sure. (What use is a database that doesn't store anything?) But a *Relational Database Management System (RDBMS)* is actually much more than data. Today's advanced RDBMSs not only store your data, they also manage that data for you, restricting what kind of data can go into the system, and also facilitating getting data out of the system. If all you want is to tuck the data away somewhere safe, you can use just about any data storage system. RDBMSs allow you to go beyond the storage of the data into the realm of defining what that data should look like. In the case of SQL Server, it doesn't just stop there. SQL Server provides additional services that help automate how your data interacts with data from other systems through such powerful features as the SQL Server Agent, Integration Services, Notification Services, and more.

Chapter 1

This chapter provides an overview to the rest of the book. Everything discussed in this chapter will be covered again in later chapters, but this chapter is intended to provide you with a roadmap or plan to bear in mind as we progress through the book. As such, in this chapter, we will take a high-level look into:

- ❑ Database objects
- ❑ Data types
- ❑ Other database concepts that ensure data integrity

An Overview of Database Objects

An RDBMS such as SQL Server contains many *objects*. Object purists out there may quibble with whether Microsoft's choice of what to call an object (and what not to) actually meets the normal definition of an object, but, for SQL Server's purposes, the list of some of the more important database objects can be said to contain such things as:

The database itself	Indexes
The transaction log	CLR assemblies
Tables	Reports
Filegroups	Full-text catalogs
Diagrams	User-defined data types
Views	Roles
Stored procedures	Users
User-defined functions	

The Database Object

The database is effectively the highest-level object that you can refer to within a given SQL Server. (Technically speaking, the server itself can be considered to be an object, but not from any real "programming" perspective, so we're not going there.) Most, but not all, other objects in a SQL Server are children of the database object.

If you are familiar with old versions of SQL Server you may now be saying, "What? What happened to logins? What happened to Remote Servers and SQL Agent tasks?" SQL Server has several other objects (as listed previously) that exist in support of the database. With the exception of linked servers, and perhaps Integration Services packages, these are primarily the domain of the database administrator and as such, you generally don't give them significant thought during the design and programming processes. (They are programmable via something called the SQL Management Objects (SMO), but that is far too special a case to concern you with here. We will look at SMO more fully in Chapter 25.)

A database is typically a group that includes at least a set of table objects and, more often than not, other objects, such as stored procedures and views that pertain to the particular grouping of data stored in the database's tables.

Being Objective: Re-Examining Objects in SQL Server

When you first load SQL Server, you will start with four system databases:

- master
- model
- msdb
- tempdb

All of these need to be installed for your server to run properly. (Indeed, for some of them, it won't run at all without them.) From there, things vary depending on which installation choices you made. Examples of some of the databases you may see include the following:

- AdventureWorks (the sample database)
- AdventureWorksDW (sample for use with Analysis Services)

The master Database

Every SQL Server, regardless of version or custom modifications, has the master database. This database holds a special set of tables (system tables) that keeps track of the system as a whole. For example, when you create a new database on the server, an entry is placed in the `sysdatabases` table in the master database. All extended and system stored procedures, regardless of which database they are intended for use with, are stored in this database. Obviously, since almost everything that describes your server is stored in here, this database is critical to your system and cannot be deleted.

The system tables, including those found in the master database, can, in a pinch, be extremely useful. That said, their direct use is diminishing in importance as Microsoft continues to give more and more other options for getting at system level information.

If you're quite cavalier, you may be saying to yourself, "Cool, I can't wait to mess around in there!" *Don't go there!* Using the system tables in any form is fraught with peril. Microsoft has recommended against using the system tables for at least the last three versions of SQL Server. They make absolutely no guarantees about compatibility in the `master` database between versions—indeed, they virtually guarantee that they will change. The worst offense comes when performing updates on objects in the `master` database. Trust me when I tell you that altering these tables in any way is asking for a SQL Server that no longer functions. Fortunately, several alternatives (for example, system functions, system stored procedures, and `information_schema` views) are available for retrieving much of the metadata that is stored in the system tables.

The model Database

The model database is aptly named, in the sense that it's the model on which a copy can be based. The model database forms a template for any new database that you create. This means that you can, if you wish, alter the `model` database if you want to change what standard, newly created databases look like. For example, you could add a set of audit tables that you include in every database you build. You could also include a few user groups that would be cloned into every new database that was created on the

Chapter 1

system. Note that since this database serves as the template for any other database, it's a required database and must be left on the system; you cannot delete it.

There are several things to keep in mind when altering the model database. First, any database you create has to be at least as large as the `model` database. That means that if you alter the `model` database to be 100MB in size, you can't create a database smaller than 100MB. There are several other similar pitfalls. As such, for 90 percent of installations, I strongly recommend leaving this one alone.

The msdb Database

`msdb` is where the *SQL Agent* process stores any system tasks. If you schedule backups to run on a database nightly, there is an entry in `msdb`. Schedule a stored procedure for one time execution, and yes, it has an entry in `msdb`.

The tempdb Database

`tempdb` is one of the key working areas for your server. Whenever you issue a complex or large query that SQL Server needs to build interim tables to solve, it does so in `tempdb`. Whenever you create a temporary table of your own, it is created in `tempdb`, even though you think you're creating it in the current database. Whenever there is a need for data to be stored temporarily, it's probably stored in `tempdb`.

`tempdb` is very different from any other database in that not only are the objects within it temporary, but the database itself is temporary. It has the distinction of being the only database in your system that is completely rebuilt from scratch every time you start your SQL Server.

Technically speaking, you can actually create objects yourself in `tempdb`—I strongly recommend against this practice. You can create temporary objects from within any database you have access to in your system—it will be stored in `tempdb`. Creating objects directly in `tempdb` gains you nothing but adds the confusion of referring to things across databases. This is another of those “Don’t go there!” kind of things.

AdventureWorks

SQL Server included samples long before this one came along. The old samples had their shortcomings though. For example, they contained a few poor design practices. (I'll hold off the argument of whether AdventureWorks has the same issue or not. Let's just say that AdventureWorks was, among other things, an attempt to address this problem.) In addition, they were simplistic and focused on demonstrating certain database concepts rather than on SQL Server as a product or even databases as a whole.

From the earliest stages of development of Yukon (the internal code name for what we know today as SQL Server 2005) Microsoft knew they wanted a far more robust sample database that would act as a sample for as much of the product as possible. AdventureWorks is the outcome of that effort. As much as you will hear me complain about its overly complex nature for the beginning user, it is a masterpiece in that it shows it *all* off. Okay, so it's not really *everything*, but it is a fairly complete sample, with more realistic volumes of data, complex structures, and sections that show samples for the vast majority of product features. In this sense, it's truly terrific.

I use it as the core sample database for this book.

AdventureWorksDW

This is the Analysis Services sample. (The DW stands for data warehouse, which is the type of database over which most Analysis Services projects will be built.) Perhaps the greatest thing about it is that Microsoft had the foresight to tie the transaction database sample with the analysis sample, providing a whole set of samples that show the two of them working together.

Decision support databases are well outside the scope of this book, and you won't be using this database, but keep it in mind as you fire up Analysis Services and play around. Take a look at the differences between the two databases. They are meant to serve the same fictional company, but they have different purposes; learn from this.

The Transaction Log

Believe it or not, the database file itself isn't where most things happen. Although the data is certainly read in from there, any changes you make don't initially go to the database itself. Instead, they are written serially to the *transaction log*. At some later point in time, the database is issued a *checkpoint*—it is at that point in time that all the changes in the log are propagated to the actual database file.

The database is in a random access arrangement, but the log is serial in nature. While the random nature of the database file allows for speedy access, the serial nature of the log allows things to be tracked in the proper order. The log accumulates changes that are deemed as having been committed, and the server writes the changes to the physical database file(s) at a later time.

We'll take a much closer look at how things are logged in Chapter 12, but for now, remember that the log is the first place on disk that the data goes, and it's propagated to the actual database at a later time. You need both the database file and the transaction log to have a functional database.

The Most Basic Database Object: Table

Databases are made up of many things, but none are more central to the make-up of a database than tables. A table can be thought of as equating to an accountant's ledger or an Excel spreadsheet. It is made up of what is called *domain* data (columns) and *entity* data (rows). The actual data for the database is stored in the tables.

Each table definition also contains the *metadata* (descriptive information about data) that describes the nature of the data it is to contain. Each column has its own set of rules about what can be stored in that column. A violation of the rules of any one column can cause the system to reject an inserted row or an update to an existing row, or prevent the deletion of a row.

Indexes

An *index* is an object that exists only within the framework of a particular table or view. An index works much like the index does in the back of an encyclopedia; there is some sort of lookup (or "key") value that is sorted in a particular way, and, once you have that, you are provided another key with which you can look up the actual information you are after.

Chapter 1

An index provides us ways of speeding the lookup of our information. Indexes fall into two categories:

- ❑ **Clustered**— You can have only one of these per table. If an index is clustered, it means that the table on which the clustered index is based is physically sorted according to that index. If you were indexing an encyclopedia, the clustered index would be the page numbers; the information in the encyclopedia is stored in the order of the page numbers.
- ❑ **Non-clustered**— You can have many of these for every table. This is more along the lines of what you probably think of when you hear the word *index*. This kind of index points to some other value that will let you find the data. For our encyclopedia, this would be the keyword index at the back of the book.

Note that views that have indexes—or *indexed views*—must have at least one clustered index before they can have any non-clustered indexes.

Triggers

A *trigger* is an object that exists only within the framework of a table. Triggers are pieces of logical code that are automatically executed when certain things, such as inserts, updates, or deletes, happen to your table.

Triggers can be used for a great variety of things but are mainly used for either copying data as it is entered or checking the update to make sure that it meets some criteria.

Constraints

A *constraint* is yet another object that exists only within the confines of a table. Constraints are much like they sound; they confine the data in your table to meet certain conditions. Constraints, in a way, compete with triggers as possible solutions to data integrity issues. They are not, however, the same thing; each has its own distinct advantages.

Schemas

Schemas provide an intermediate namespace between your database and the other objects it contains. The default namespace in any database is “dbo” (which stands for database owner). Every user has a default schema, and SQL Server will search for objects within that user’s default schema automatically. If, however, the object is within a namespace that is not the default for that user, then the object must be referred with two parts in the form of <schema name>. <object name>.

Schemas replace the concept of “owner” that was used in prior versions of SQL Server. While Microsoft seems to be featuring their use in this release (the idea is that you’ll be able to refer to a group of tables by the schema they are in rather than listing them all), I remain dubious at best. In short, I believe they create far more problems than they solve, and I generally recommend against their use (I have made my exceptions, but they are very situational).

Filegroups

By default, all your tables and everything else about your database (except the log) are stored in a single file. That file is a member of what's called the *primary filegroup*. However, you are not stuck with this arrangement.

SQL Server allows you to define a little over 32,000 *secondary files*. (If you need more than that, perhaps it isn't SQL Server that has the problem.) These secondary files can be added to the primary filegroup or created as part of one or more *secondary filegroups*. While there is only one primary filegroup (and it is actually called "Primary"), you can have up to 255 secondary filegroups. A secondary filegroup is created as an option to a `CREATE DATABASE` or `ALTER DATABASE` command.

Diagrams

We will discuss database diagramming in some detail when we discuss database design, but for now, suffice it to say that a database diagram is a visual representation of the database design, including the various tables, the column names in each table, and the relationships between tables. In your travels as a developer, you may have heard of an *entity-relationship* diagram—or ERD. In an ERD the database is divided into two parts: entities (such as "supplier" and "product") and relations (such as "supplies" and "purchases").

Although they have been entirely redesigned with SQL Server 2005, the included database design tools remain a bit sparse. Indeed, the diagramming methodology the tools use doesn't adhere to any of the accepted standards in ER diagramming.

Still, these diagramming tools really do provide all the "necessary" things; they are at least something of a start.

Figure 1-1 is a diagram that shows some of the various tables in the AdventureWorks database. The diagram also (though it may be a bit subtle since this is new to you) describes many other properties about the database. Notice the tiny icons for keys and the infinity sign. These depict the nature of the relationship between two tables.

Views

A view is something of a virtual table. A view, for the most part, is used just like a table, except that it doesn't contain any data of its own. Instead, a view is merely a preplanned mapping and representation of the data stored in tables. The plan is stored in the database in the form of a query. This query calls for data from some, but not necessarily all, columns to be retrieved from one or more tables. The data retrieved may or may not (depending on the view definition) have to meet special criteria in order to be shown as data in that view.

Until SQL Server 2000, the primary purpose of views was to control what the user of the view saw. This has two major impacts: security and ease of use. With views you can control what the users see, so if there is a section of a table that should be accessed by only a few users (for example, salary details), you can create a view that includes only those columns to which everyone is allowed access. In addition, the view can be tailored so that the user doesn't have to search through any unneeded information.

Chapter 1

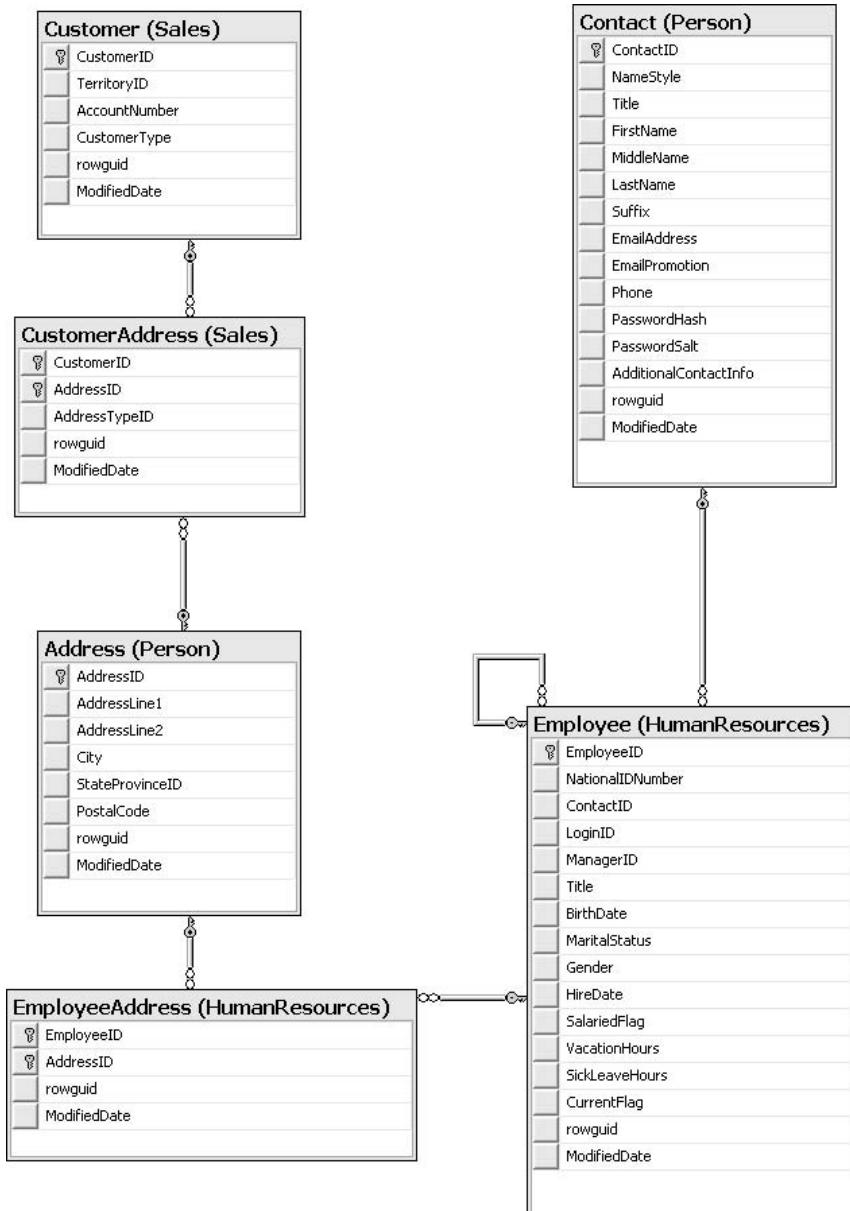


Figure 1-1

In addition to these most basic uses for view, we also have the ability to create what is called an *indexed view*. This is the same as any other view, except that we can now create an index against the view. This results in a few performance impacts (some positive, one negative):

Being Objective: Re-Examining Objects in SQL Server

- ❑ Views that reference multiple tables generally perform *much* faster with an indexed view because the join between the tables is preconstructed.
- ❑ Aggregations performed in the view are precalculated and stored as part of the index; again, this means that the aggregation is performed one time (when the row is inserted or updated), and then can be read directly from the index information.
- ❑ Inserts and deletes have higher overhead because the index on the view has to be updated immediately; updates also have higher overhead if the key column of the index is affected by the update.

We will look into these performance issues more deeply in Chapter 9.

Stored Procedures

Stored procedures (or *sprocs*) are historically and, in the .NET era, even more likely to be the bread and butter of programmatic functionality in SQL Server. Stored procedures are generally an ordered series of Transact-SQL (the language used to query Microsoft SQL Server) statements bundled up into a single logical unit. They allow for variables and parameters as well as selection and looping constructs. Sprocs offer several advantages over just sending individual statements to the server in the sense that they:

- ❑ Are referred to using short names, rather than a long string of text; as such, less network traffic is required in order to run the code within the sproc.
- ❑ Are pre-optimized and precompiled, saving a small amount of time each time the sproc is run.
- ❑ Encapsulate a process, usually for security reasons or just to hide the complexity of the database.
- ❑ Can be called from other sprocs, making them reusable in a somewhat limited sense.

In addition, you can utilize any .NET language to add program constructs, beyond those native to T-SQL, to your stored procedures.

User-Defined Functions

User-defined functions (or *UDFs*) have a tremendous number of similarities to sprocs, except that they:

- ❑ Can return a value of most SQL Server data types. Excluded return types include `text`, `ntext`, `image`, `cursor`, and `timestamp`.
- ❑ Can't have "side effects." Basically, they can't do anything that reaches outside the scope of the function, such as changing tables, sending e-mails, or making system or database parameter changes.

UDFs are similar to the functions that you would use in a standard programming language such as VB.NET or C++. You can pass more than one variable in, and get a value out. SQL Server's UDFs vary from the functions found in many procedural languages, however, in that *all* variables passed into the function are passed in by value. If you're familiar with passing in variables `By Ref` in VB, or passing in pointers in C++, sorry, there is no equivalent here. There is, however, some good news in that you can return a special data type called a table. We'll examine the impact of this in Chapter 11.

Users and Roles

These two go hand in hand. *Users* are pretty much the equivalent of logins. In short, this object represents an identifier for someone to log in to the SQL Server. Anyone logging into SQL Server has to map (directly or indirectly depending on the security model in use) to a user. Users, in turn, belong to one or more *roles*. Rights to perform certain actions in SQL Server can then be granted directly to a user or to a role to which one or more users belong.

Rules

Rules and constraints provide restriction information about what can go into a table. If an updated or inserted record violates a rule, then that insertion or update will be rejected. In addition, a rule can be used to define a restriction on a *user-defined data type*. Unlike rules, constraints aren't really objects unto themselves but rather pieces of metadata describing a particular table.

While Microsoft has not stated a particular version for doing so, they have warned that rules will be removed in a future release. Rules should be considered for backward compatibility only and should be avoided in new development. You may also want to consider phasing out any you already have in use in your database.

Defaults

There are two types of defaults. There is the default that is an object unto itself and the default that is not really an object, but rather metadata describing a particular column in a table (in much the same way that we have constraints which are objects, and rules, which are not objects but metadata). They both serve the same purpose. If, when inserting a record, you don't provide the value of a column and that column has a default defined, a value will be inserted automatically as defined in the default. We will examine both types of defaults in Chapter 5.

Much like rules, the form of default that is its own object should be treated as a legacy object and avoided in new development. Use of default constraints is, however, still very valid. See Chapter 5 for more information.

User-Defined Data Types

User-defined data types are extensions to the system-defined data types. Beginning with this version of SQL Server, the possibilities here are almost endless. Although SQL Server 2000 and earlier had the idea of user-defined data types, they were really limited to different filtering of existing data types. With SQL Server 2005, you have the ability to bind .NET assemblies to your own data types, meaning you can have a data type that stores (within reason) about anything you can store in a .NET object.

Careful with this! The data type that you're working with is pretty fundamental to your data and its storage. Although being able to define your own thing is very cool, recognize that it will almost certainly come with a large performance cost. Consider it carefully, be sure it's something you need, and then, as with everything like this, TEST, TEST, TEST!!!

Full-Text Catalogs

Full-text catalogs are mappings of data that speed the search for specific blocks of text within columns that have full-text searching enabled. Although these objects are joined at the hip to the tables and columns that they map, they are separate objects and are as such, not automatically updated when changes happen in the database.

SQL Server Data Types

Now that you're familiar with the base objects of a SQL Server database, let's take a look at the options that SQL Server has for one of the fundamental items of any environment that handles data: data types. Note that since this book is intended for developers and that no developer could survive for 60 seconds without an understanding of data types, I'm going to assume that you already know how data types work and just need to know the particulars of SQL Server data types.

SQL Server 2005 has the intrinsic data types shown in the following table:

Data Type Name	Class	Size in Bytes	Nature of the Data
Bit	Integer	1	The size is somewhat misleading. The first bit data type in a table takes up one byte; the next seven make use of the same byte. Allowing nulls causes an additional byte to be used.
Bigint	Integer	8	This just deals with the fact that we use larger and larger numbers on a more frequent basis. This one allows you to use whole numbers from -2^{63} to $2^{63}-1$. That's plus or minus about 92 quintillion.
Int	Integer	4	Whole numbers from $-2,147,483,648$ to $2,147,483,647$.
SmallInt	Integer	2	Whole numbers from $-32,768$ to $32,767$.
TinyInt	Integer	1	Whole numbers from 0 to 255.
Decimal or Numeric	Decimal/ Numeric	Varies	Fixed precision and scale from $-10^{38}-1$ to $10^{38}-1$. The two names are synonymous.
Money	Money	8	Monetary units from -2^{63} to 2^{63} plus precision to four decimal places. Note that this could be any monetary unit, not just dollars.
SmallMoney	Money	4	Monetary units from $-214,748.3648$ to $+214,748.3647$.
Float (also a synonym for ANSI Real)	Approximate Numerics	Varies	Accepts an argument (for example, <code>Float(20)</code>) that determines size and precision. Note that the argument is in bits, not bytes. Ranges from $-1.79E + 308$ to $1.79E + 308$.

Table continued on following page

Chapter 1

Data Type Name	Class	Size in Bytes	Nature of the Data
DateTime	Date/Time	8	Date and time data from January 1, 1753, to December 31, 9999, with an accuracy of three-hundredths of a second.
SmallDateTime	Date/Time	4	Date and time data from January 1, 1900, to June 6, 2079, with an accuracy of one minute.
Cursor	Special Numeric	1	Pointer to a cursor. While the pointer takes up only a byte, keep in mind that the result set that makes up the actual cursor also takes up memory — exactly how much will vary depending on the result set.
Timestamp/rowversion	Special Numeric (binary)	8	Special value that is unique within a given database. Value is set by the database itself automatically every time the record is inserted or updated, even though the timestamp column wasn't referred to by the UPDATE statement. (You're actually not allowed to update the timestamp field directly.)
Unique Identifier	Special Numeric (binary)	16	Special Globally Unique Identifier (GUID). Is guaranteed to be unique across space and time.
Char	Character	Varies	Fixed-length character data. Values shorter than the set length are padded with spaces to the set length. Data is non-Unicode. Maximum specified length is 8,000 characters.
VARCHAR	Character	Varies	Variable-length character data. Values are not padded with spaces. Data is non-Unicode. Maximum specified length is 8,000 characters, but you can use the <code>max</code> keyword to indicate it as essentially a very large character field (up to 2^{31} bytes of data).
Text	Character	Varies	Legacy support as of SQL Server 2005. Use <code>varchar(max)</code> instead.
NChar	Unicode	Varies	Fixed-length Unicode character data. Values shorter than the set length are padded with spaces. Maximum specified length is 4,000 characters.
NVARCHAR	Unicode	Varies	Variable-length Unicode character data. Values aren't padded. Maximum specified length is 4,000 characters, but you can use the <code>max</code> keyword to indicate it as essentially a very large character field (up to 2^{31} bytes of data).

Being Objective: Re-Examining Objects in SQL Server

Data Type Name	Class	Size in Bytes	Nature of the Data
Ntext	Unicode	Varies	Like the Text data type, this is legacy support only. In this case, use nvarchar(max). Variable-length Unicode character data.
Binary	Binary	Varies	Fixed-length binary data with a maximum length of 8,000 bytes.
VarBinary	Binary	Varies	Variable-length binary data with a maximum specified length of 8,000 bytes, but you can use the max keyword to indicate it as essentially a LOB field (up to 2^{31} bytes of data).
Image	Binary	Varies	Legacy support only as of SQL Server 2005. Use varbinary(max) instead.
Table	Other	Special	This is primarily for use in working with result sets, typically passing one out of a user-defined function. Not usable as a data type within a table definition. (You can't nest tables.)
Sql_variant	Other	Special	This is loosely related to the Variant in VB and C++. Essentially it's a container that enables you to hold most other SQL Server data types in it. That means you can use this when one column or function needs to be able to deal with multiple data types. Unlike VB, using this data type forces you to cast it <i>explicitly</i> to convert it to a more specific data type.
XML	Character	Varies	Defines a character field as being for XML data. Provides for the validation of data against an XML schema and the use of special XML-oriented functions.

Most of these have equivalent data types in other programming languages. For example, an `int` in SQL Server is equivalent to a `Long` in Visual Basic, and for most systems and compiler combinations in C++, is equivalent to an `int`.

SQL Server has no concept of unsigned numeric data types. If you need to allow for larger numbers than the signed data type allows, consider using a larger signed data type. If you need to prevent the use of negative numbers, consider using a CHECK constraint that restricts valid data to greater than or equal to zero.

In general, SQL Server data types work much as you would expect given experience in most other modern programming languages. Adding numbers yields a sum, but adding strings concatenates them.

Chapter 1

When you mix the usage or assignment of variables or fields of different data types, a number of types convert implicitly (or automatically). Most other types can be converted explicitly. (You say specifically what type you want to convert to.) A few can't be converted between at all. Figure 1-2 contains a chart that shows the various possible conversions.

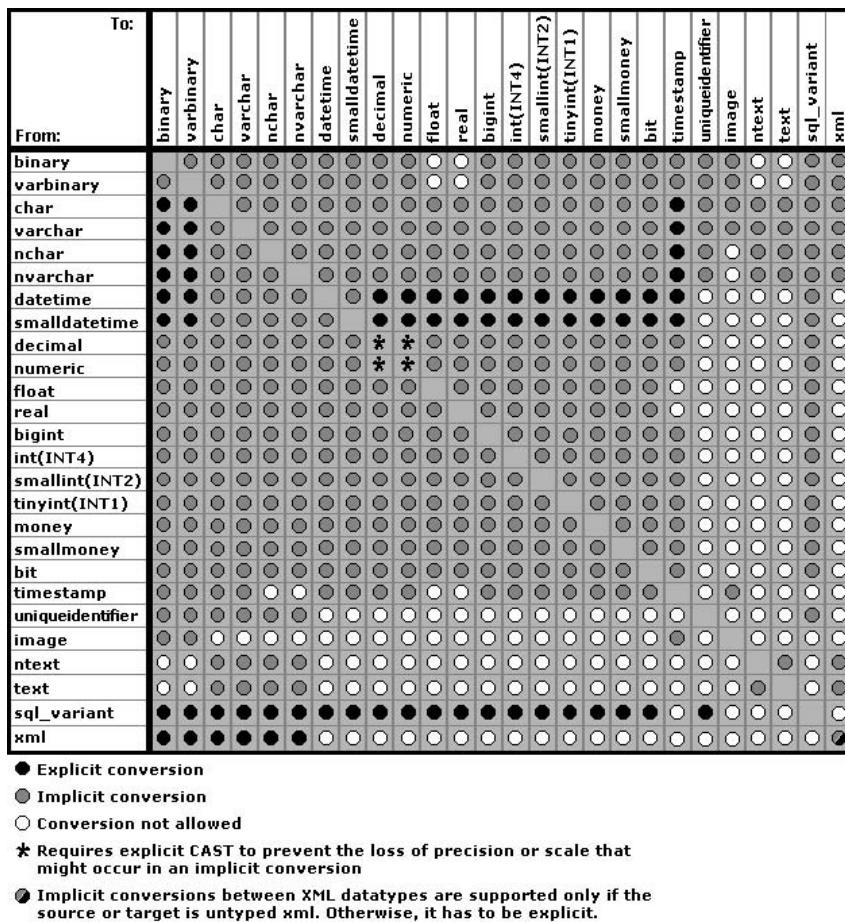


Figure 1-2

In short, data types in SQL Server perform much the same function that they do in other programming environments. They help prevent programming bugs by ensuring that the data supplied is of the same nature that the data is supposed to be (remember 1/1/1980 means something different as a date than as a number) and ensures that the kind of operation performed is what you expect.

NULL Data

What if you have a row that doesn't have any data for a particular column—that is, what if you simply don't know the value? For example, let's say that we have a record that is trying to store the company performance information for a given year. Now, imagine that one of the fields is a percentage growth

Being Objective: Re-Examining Objects in SQL Server

over the prior year, but you don't have records for the year before the first record in your database. You might be tempted to just enter a zero in the PercentGrowth column. Would that provide the right information though? People who didn't know better might think that meant you had zero percent growth, when the fact is that you simply don't know the value for that year.

Values that are indeterminate are said to be `NULL`. It seems that every time I teach a class in programming, at least one student asks me to define the value of `NULL`. Well, that's a tough one, because, by definition, a `NULL` value means that you don't know what the value is. It could be 1; it could be 347; it could be -294 for all we know. In short, it means *undefined* or perhaps *not applicable*.

SQL Server Identifiers for Objects

Now you've heard all sorts of things about objects in SQL Server. But let's take a closer look at naming objects in SQL Server.

What Gets Named?

Basically, everything has a name in SQL Server. Here's a partial list:

Stored procedures	Tables	Columns
Views	Rules	Constraints
Defaults	Indexes	Filegroups
Triggers	Databases	Servers
User-defined functions	Logins	Roles
Full-text catalogs	Files	User-defined types
Schemas		

And the list goes on. Most things I can think of except rows (which aren't really objects) have a name. The trick is to make every name both useful and practical.

Rules for Naming

The rules for naming in SQL Server are fairly relaxed, allowing things like embedded spaces and even keywords in names. Like most freedoms, however, it's easy to make some bad choices and get yourself into trouble.

Here are the main rules:

- ❑ The name of your object must start with any letter as defined by the specification for Unicode 2.0. This includes the letters most westerners are used to—A–Z and a–z. Whether “A” is different from “a” depends on the way your server is configured, but either makes for a valid beginning to an object name. After that first letter, you're pretty much free to run wild; almost any character will do.

Chapter 1

- ❑ The name can be up to 128 characters for normal objects and 116 for temporary objects.
- ❑ Any names that are the same as SQL Server keywords or contain embedded spaces must be enclosed in double quotes (" ") or square brackets ([]). Which words are considered keywords varies, depending on the compatibility level to which you have set your database.

Note that double quotes are acceptable as a delimiter for column names only if you have SET QUOTED_IDENTIFIER ON. Using square brackets ([and]) eliminates the chance that your users will have the wrong setting but is not as platform independent as double quotes are.

These rules are generally referred to as the rules for identifiers and are in force for any objects you name in SQL Server. Additional rules may exist for specific object types.

Again, I can't stress enough the importance of avoiding the use of SQL Server keywords or embedded spaces in names. Although both are technically legal as long as you qualify them, naming things this way will cause you no end of grief.

Summary

Like most things in life, the little things do matter when thinking about an RDBMS. Sure, almost anyone who knows enough to even think about picking up this book has an idea of the *concept* of storing data in columns and rows, even if they don't know that these groupings of columns and rows should be called tables, but a few tables seldom make a real database. The things that make today's RDBMSs great are the extra things—the objects that enable you to place functionality and business rules that are associated with the data right into the database with the data.

Database data has *type*, just as most other programming environments do. Most things that you do in SQL Server are going to have at least some consideration of type. Review the types that are available, and think about how these types map to the data types in any programming environment with which you are familiar.

2

Tool Time

Okay, so if you took the time to read Chapter 1, then you should now have a feel for how fast we're moving here. Again, if you're new to all of this, I would suggest swallowing your pride and starting with the *Beginning SQL Server 2005 Programming* title—it covers the basics in far more detail. For this book, our purpose in covering the stuff in the first few chapters is really more about providing a reference than anything else, so we really are just taking a whirlwind tour. With that in mind, it's time to move on to the toolset. If SQL Server 2005 was not your first experience with SQL Server, then this is a place where you'll want to pay particular attention. With SQL Server 2005, the toolset has changed—*a lot*.

Virtually everything to do with the SQL Server toolset has seen a complete overhaul for SQL Server 2005. Simplifying the “where do I find things?” question was a major design goal for the tools team in this release. For old fogies such as me, the new tools are a rather nasty shock to the system. For people new to SQL Server, I would say that the team has largely met that simplification goal. In general, there are far fewer places to look for things, and most of the toolset is grouped far more logically.

The tools we will look at in this chapter are:

- ❑ SQL Server Books Online
- ❑ The SQL Server Configuration Manager
- ❑ SQL Server Management Studio
- ❑ SQL Server Business Intelligence Development Studio
- ❑ SQL Server Integration Services (SSIS): including the Import/Export Wizard
- ❑ Reporting Services
- ❑ The Bulk Copy Program (BCP)
- ❑ Profiler
- ❑ sqlcmd

Books Online

Is *Books Online* a tool? I think so. Let's face it: It doesn't matter how many times you read this or any other book on SQL Server; you're not going to remember everything you'll ever need to know about SQL Server. SQL Server is one of my mainstay products, and I still can't remember it all. Books Online is simply one of the most important tools you're going to find in SQL Server.

Here's a simple piece of advice: Don't even try to remember it all. Remember that what you've seen is possible. Remember what is an integral foundation to what you're doing. Remember what you work with every day. Then remember to build a good reference library (starting with this book) for the rest.

As you see in Figure 2-1, Books Online in SQL Server uses the updated .NET online help interface, which is replacing the older standard online help interface used among the Microsoft technical product line (Back Office, MSDN, Visual Studio).

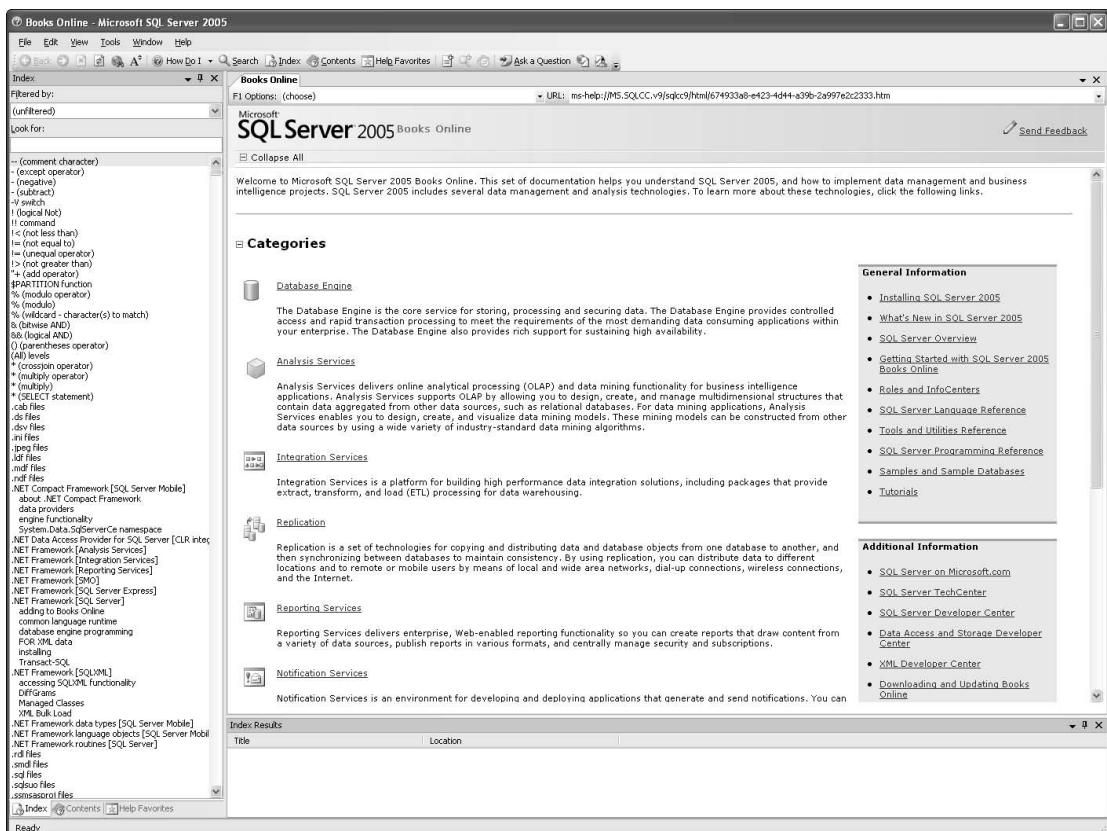


Figure 2-1

Everything works pretty much as one would expect here, so I'm not going to go into the details of how to operate a help system. Suffice it to say that SQL Server Books Online is a great quick reference that follows you to whatever machine you're working on at the time. Books Online also has the added benefit of often having information that is more up to date than the printed documentation.

Technically speaking, it's quite possible that not every system you move to will have the Books Online (BOL) installed. This is because you can manually de-select BOL at the time of installation. Even in tight space situations, however, I strongly recommend that you always install the BOL. It really doesn't take up all that much space when you consider cost per megabyte these days, and it can save you a fortune in time by having that quick reference available wherever you are running SQL Server. (On my machine, Books Online takes up 100MB of space.)

The SQL Server Configuration Manager

Administrators who configure computers for database access are the main users of this tool, but it is still important for us to understand what this tool is about.

The SQL Server Configuration Manager is a new tool with SQL Server 2005 but is really an effort to combine some settings that were spread across multiple tools into one spot. The items managed in the Computer Manager fall into two areas:

- Service Management
- Network Configuration

Service Management

Let's cut to the chase—the services available for management here include:

- Integration Services**—This powers the Integration Services.
- Analysis Services**—This powers the Analysis Services engine.
- Full-Text Search**—Again, just what it sounds like—powers the Full-Text Search Engine.
- Reporting Services**—The underlying engine for Report Services.
- SQL Server Agent**—The main engine behind anything in SQL Server that is scheduled. Utilizing this service, you can schedule jobs to run on a variety of different schedules. These jobs can have multiple tasks assigned to them and can even branch into doing different tasks depending on the outcome of some previous task. Examples of things run by the SQL Server Agent include backups as well as routine import and export tasks.
- SQL Server**—The core database engine that works on data storage, queries, and system configuration for SQL Server.
- SQL Server Browser**—Supports advertising your server so those browsing your local network can identify your system has SQL Server installed.

Network Configuration

A fair percentage of the time, any of the connectivity issues discovered are the result of client network configuration or how that configuration matches with that of the server.

SQL Server provides several of what are referred to as *Net-Libraries* (network libraries), or *NetLibs*. NetLibs serve as something of an insulator between your client application and the network protocol that is to be used—they serve the same function at the server end, too. The NetLibs supplied with SQL Server 2005 include:

- Named Pipes
- TCP/IP (the default)
- Shared Memory
- VIA (a proprietary virtual adaptor generally used for server to server communication based on special hardware)

The same NetLib must be available on both the client and server computers so that they can communicate with each other via the network protocol. Choosing a client NetLib that is not also supported on the server will result in your connection attempt failing (with a Specified SQL Server Not Found error).

Regardless of the data access method and kind of driver used (SQL Native Client, ODBC, OLE DB, or DB-Lib), it will always be the driver that talks to the NetLib. The process works as shown in Figure 2-2. The steps in order are:

1. The client app talks to the driver (SQL Native Client, ODBC, OLE DB, or DB-Lib).
2. The driver calls the client NetLib.
3. This NetLib calls the appropriate network protocol and transmits the data to a server NetLib.
4. The server NetLib then passes the requests from the client to the SQL Server.

Replies from SQL Server to the client follow the same sequence, only in reverse.

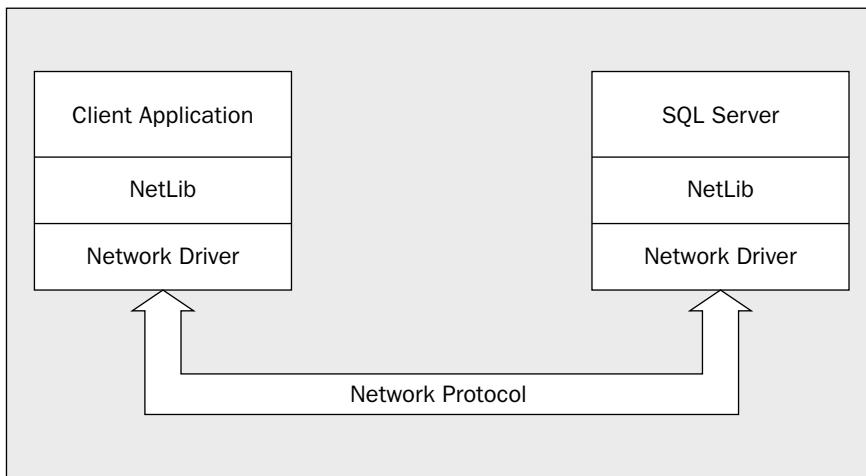


Figure 2-2

In case you're familiar with TCP/IP, the default port that the IP NetLib will listen on is 1433.

The Protocols

Let's start off with that "What are the available choices?" question. We can see what our server *could* be listening for by starting the SQL Server Computer Manager and expanding the Protocols for MSSQLSERVER tree under Server Network Configuration, as shown in Figure 2-3.

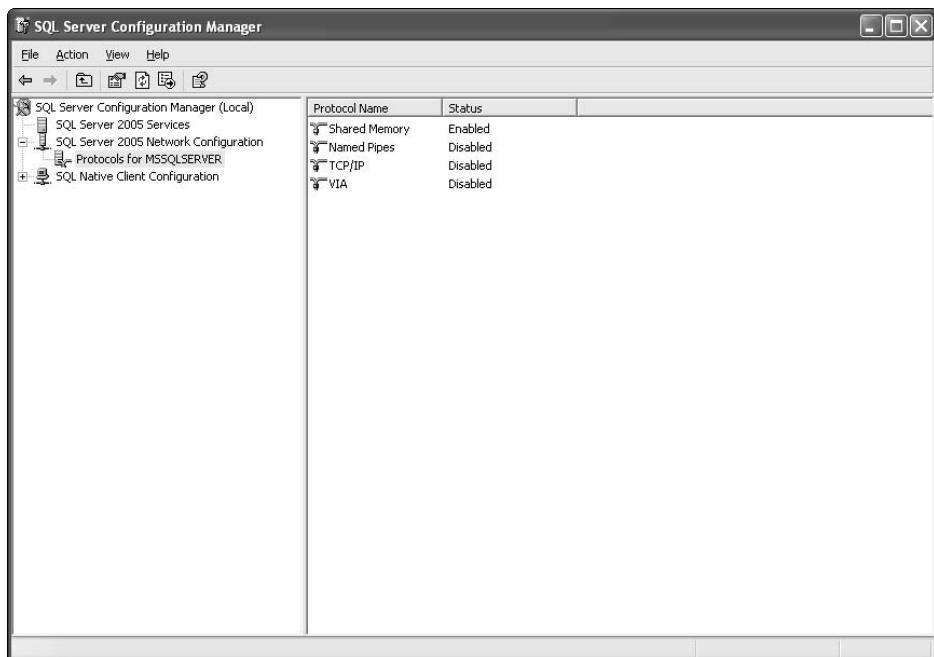


Figure 2-3

By default, only Shared Memory will be enabled. Older versions of the product had different NetLibs enabled by default depending on the version of SQL Server and the O/S.

You will need to enable at least one other NetLib if you want to be able to remotely contact your SQL Server (say, from a Web server or from different clients on your network).

Keep in mind that, in order for your client to gain a connection to the server, the server has to be listening for the protocol with which the client is trying to communicate and on the same port. Therefore, if we were in a Named Pipes environment, we might need to add a new library. To do that, we would go back to the Protocols tree, right-click on the Named Pipes protocol, and choose enable.

Chapter 2

At this point, you might be tempted to say, “Hey, why don’t I just enable every NetLib? Then I won’t have to worry about it.” This situation is like anything you add onto your server—more overhead. In this case, it would both slow down your server (not terribly, but every little bit counts) and expose you to unnecessary openings in your security (why leave an extra door open if nobody is supposed to be using that door?).

Okay, now let’s take a look at what we can support and why we would want to choose a particular protocol.

Named Pipes

Named Pipes can be very useful in situations where either TCP/IP is not available or there is no Domain Name Service (DNS) server to allow the naming of servers under TCP/IP. Named Pipes is decreasing in use, and, to make a long story short, I like it that way. The short rendition of why is that you’re going to have TCP/IP active anyway, so why add another protocol to the mix (especially since it opens another way that hackers could potentially infiltrate your system).

Technically speaking, you can connect to a SQL Server running TCP/IP by using its IP address in the place of the name. This will work all the time as long as you have a route from the client to the server—even if there is no DNS service (if it has the IP address, then it doesn’t need the name).

TCP/IP

TCP/IP has become something of the de facto standard networking protocol and has been a default with SQL Server since SQL Server 2000. It is also the only option if you want to connect directly to your SQL Server via the Internet (which, of course, uses IP only).

Don’t confuse the need to have your database server available to a Web server with the need to have your database server directly accessible to the Internet. You can have a Web server that is exposed to the Internet but that also has access to a database server that is not directly exposed to the Internet (the only way for an Internet connection to see the data server is through the Web server).

Connecting your data server directly to the Internet is a security hazard in a big way. If you insist on doing it (and there can be valid reasons for doing so), then pay particular attention to security precautions.

Shared Memory

Shared Memory removes the need for interprocess marshaling (which is a way of packaging information before transferring it across process boundaries) between the client and the server if they are running on the same box. The client has direct access to the same memory-mapped file where the server is storing data. This removes a substantial amount of overhead and is *very* fast. It is useful only when accessing the server locally (say, from a Web server installed on the same server as the database), but it can be quite a boon performance-wise.

VIA

VIA stands for *Virtual Interface Adapter*, and the specific implementation will vary from vendor to vendor. In general, it is usually a network kind of interface but is usually a very high-performance, dedicated connection between two systems. Part of that high performance comes from specialized, dedicated hardware that knows that it has a dedicated connection and therefore doesn't have to deal with normal network addressing issues.

On to the Client

Once we know what our server is offering, we can go and configure the client. Most of the time, the defaults are going to work just fine. In the Computer Manager, expand the Client Network Configuration tree and select the Client Protocols node, as shown in Figure 2-4.

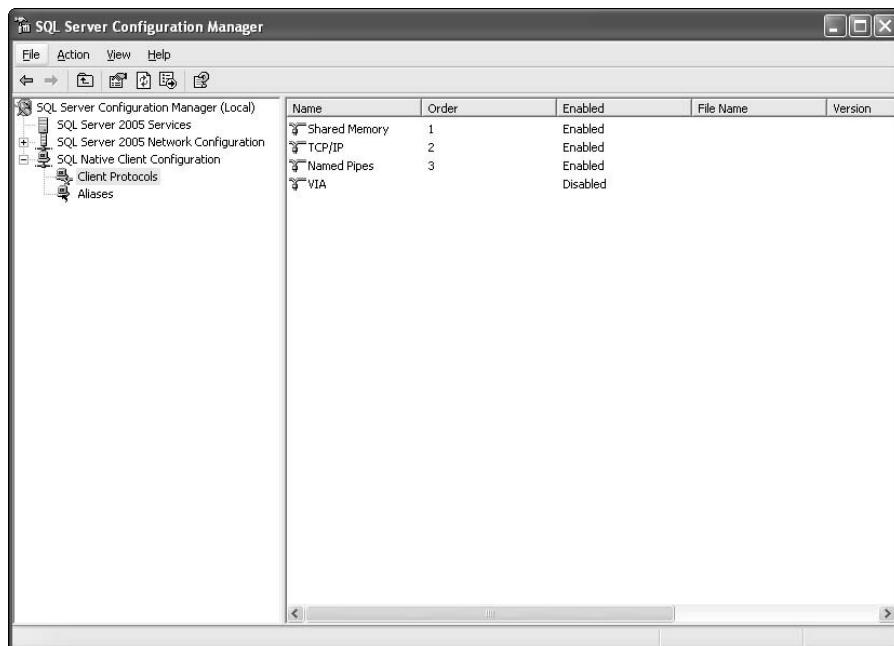


Figure 2-4

Beginning with SQL Server 2000, Microsoft added the ability for the client to start with one protocol, and then, if that didn't work, move on to another. In the preceding dialog, we are first using Shared Memory, then trying TCP/IP, and finally going to Named Pipes if TCP/IP doesn't work as defined by the "Order" column. Unless you change the default (changing the priority by using the up and down arrows), Shared Memory is the NetLib that will be used first for connections to any server not listed in the aliases list (the next node under Client Network Configuration), followed by TCP/IP, and so on.

If you have TCP/IP support on your network, leave your server configured to use it. IP has less overhead and just plain runs faster—there is no reason not to use it unless your network doesn't support it. It's worth noting, however, that for local servers (where the server is on the same physical system as the client), the Shared Memory NetLib will be quicker, as you do not need to go across the network to view your local SQL server.

The Aliases list is a listing of all the servers where you have defined a specific NetLib to be used when contacting that particular server. This means that you can contact one server using IP and another using Named Pipes—whatever you need to get to that particular server. In this case, shown in Figure 2-5, we've configured our client to use the Named Pipes NetLib for requests from the server named ARISTOTLE, and to use whatever we've set up as our default for contact with any other SQL Server.

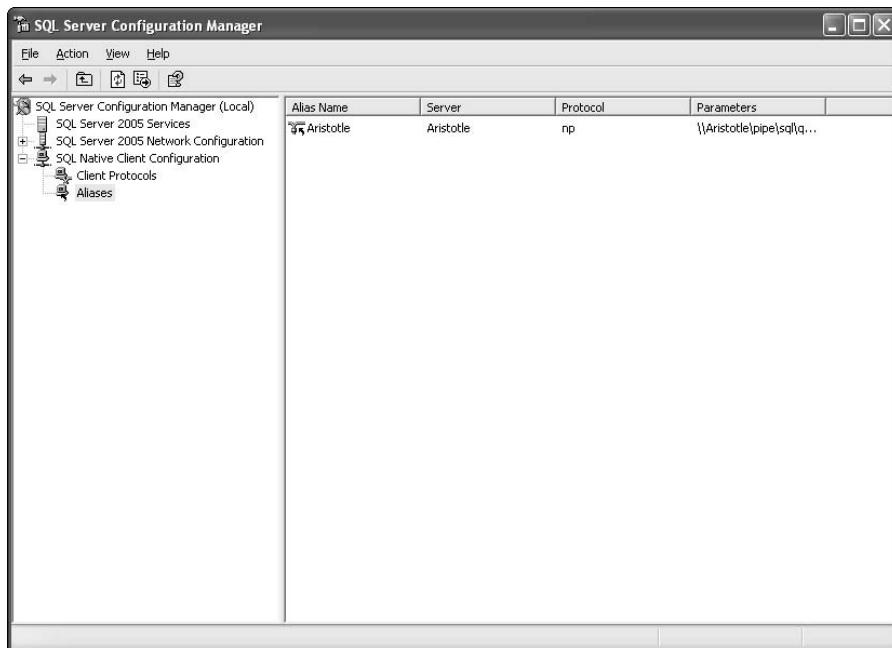


Figure 2-5

Again, remember that the Client Network Configuration setting on the network machine must either have a default protocol that matches one supported by the server or have an entry in the Aliases list to specifically choose a NetLib supported by that server.

The SQL Server Management Studio

The *SQL Server Management Studio* is pretty much home base when administering a SQL Server. It provides a variety of functionality for managing your server using a relatively easy-to-use graphical user

interface. Management Studio is completely new with SQL Server 2005. Patterned loosely after the DevStudio IDE environment, it combines a myriad of functionality that used to be in separate tools.

For the purposes of this book, we're not going to cover everything that the Management Studio has to offer, but let's make a quick rundown of the things you can do:

- ❑ Create, edit, and delete databases and database objects
- ❑ Manage scheduled tasks such as backups and the execution of SSIS package runs
- ❑ Display current activity, such as who is logged on, what objects are locked, and from which client they are running
- ❑ Manage security, including such items as roles, logins, and remote and linked servers
- ❑ Initiate and manage the Database Mail Service
- ❑ Create and manage Full-Text Search Catalogs
- ❑ Manage configuration settings for the server
- ❑ Create and manage both publishing and subscribing databases for replication

We will be seeing a great deal of Management Studio throughout this book, so let's take a closer look at some of the more key functions Management Studio serves.

Getting Started

When you first start Management Studio, you will be presented in a connection dialog similar to the one in Figure 2-6.



Figure 2-6

Again, since we're assuming that you have prior knowledge of SQL Server (if you don't, you may want to check out the *Beginning SQL Server 2005 Programming* book first), we're not going to dwell much on basic connections. That said, there are a couple that are handled a bit differently than they were in prior versions, so let's look at those in depth.

Server Type

This relates to which of the various sub-systems of SQL Server you are logging into (the normal database server, Analysis Services, Report Server, or Integration Services). Since these different types of “servers” can share the same name, pay attention to this to make sure you’re logging into what you think you’re logging into.

SQL Server

As you might guess, this is the SQL Server into which you’re asking to be logged.

SQL Server allows multiple “instances” of SQL Server to run at one time. These are just separate loads into memory of the SQL Server engine running independently from each other.

Note that the default instance of your server will have the same name as your machine on the network. There are ways to change the server name after the time of installation, but they are problematic at best, and deadly to your server at worst. Additional instances of SQL Server will be named the same as the default (SCHWEITZER in many of the examples in this book), followed by a dollar sign, then the instance name — for example, ARISTOTLE\$POMPEII.

If you select (local), then your system will connect to the SQL Server on the same computer as you are trying to connect from and will use the Shared Memory NetLib regardless of what NetLib you have selected for contacting other servers. This is a bad news/good news story. The bad news is that you give up a little bit of control (SQL Server will always use Shared Memory to connect — you can’t choose anything else). The good news is that you don’t have to remember what server you’re on, and you get a high-performance option for work on the same machine. If you use your local PC’s actual server name, then your communications will still go through the network stack and incur the overhead associated with that just as if you were communicating with another system, regardless of the fact that it is on the same machine.

Authentication Type

You can choose between *Windows authentication* (formerly NT authentication) and *SQL Server authentication*. No matter how you configure your server, Windows authentication will always be available (even if you configured it as “SQL Server Authentication”). Logins using usernames and passwords that are local to SQL Server (not part of a larger Windows network) are acceptable to the system only if you have specifically turned SQL Server Authentication on.

Windows Authentication

Windows authentication is just as it sounds. You have users and groups defined in Windows 2000 or later. Those Windows users are mapped into SQL Server “Logins” in their Windows user profile. When they attempt to log into SQL Server, they are validated through the Windows domain and mapped to “roles” according to the Login. These roles identify what the user is allowed to do.

The best part of this model is that you have only one password (if you change it in the Windows domain, then it’s changed for your SQL Server logins, too); you pretty much don’t have to fill in anything to log in (it just takes the login information from how you’re currently logged in to the Windows network).

Additionally, the administrator has to administer users in only one place. The downside is that mapping out this process can get complex and, to administer the Windows user side of things, it requires that you be a domain administrator.

SQL Server Authentication

The security does not care at all about what the user's rights to the network are, but rather what you have explicitly set up in SQL Server. The authentication process does not take into account the current network login at all—instead, the user provides a SQL Server-specific login and password.

This can be nice, since the administrator for a given SQL Server does not need to be a domain administrator (or even have a username on your network for that matter) in order to give rights to users on the SQL Server. The process also tends to be somewhat simpler than under Windows authentication. Finally, it means that one user can have multiple logins that give different rights to different things.

Query Window

This part of the Management Studio takes the place of a separate tool in previous versions that was called *Query Analyzer*. It is your tool for interactive sessions with a given SQL Server. It is where you can execute statements using *Transact-SQL* (*T-SQL*)—I lovingly pronounce it “Tee-Squeal,” but it’s supposed to be “Tee-Sequel”). *T-SQL* is the native language of SQL Server. It is a dialect of Structured Query Language (*SQL*) and is entry-level ANSI SQL 92 compliant. Entry-level compliant means that SQL Server meets a first tier of requirements needed to classify a product as ANSI compliant. You’ll find that most RDBMS products support ANSI only to entry level.

Personally, I’m not all that thrilled with this new version of the tool—I find that, because of how many things are done with the one tool, the user interface is rather cluttered, and it can be hard to find what you’re looking for. That said, for those without the expectations of the past, Microsoft is hoping it will actually prove to be more intuitive for you to use it within the larger Management Studio, too.

Since the Query window is where we will spend a fair amount of time in this book, let’s take a quick look at this tool and get familiar with how to use it.

Getting Started

Open a new Query window by clicking the New Query button toward the top left of Management Studio or choosing File→New→New Query With Current Connection from the File menu. When the Query window is up, we’ll get menus that largely match what we had in Query Analyzer back when that was a separate tool. We’ll look at the specifics, but let’s get our simple query out of the way.

Let’s start with:

```
SELECT * FROM INFORMATION_SCHEMA.TABLES
```

Statement keywords should appear in blue; unidentifiable items, such as column and table names (these vary with every table in every database on every server), are in black; and statement arguments and connectors are in red. Pay attention to how these work and learn them—they can help you catch many bugs before you’ve even run the statement (and seen the resulting error). The check mark icon on the toolbar quickly parses the query for you without the need to actually attempt to run the statement. If there are any syntax errors, this should catch them before you see error messages.

Chapter 2

Now click on the execute button in the toolbar (with the red exclamation point next to it). The Query window changes a bit, as shown in Figure 2-7.

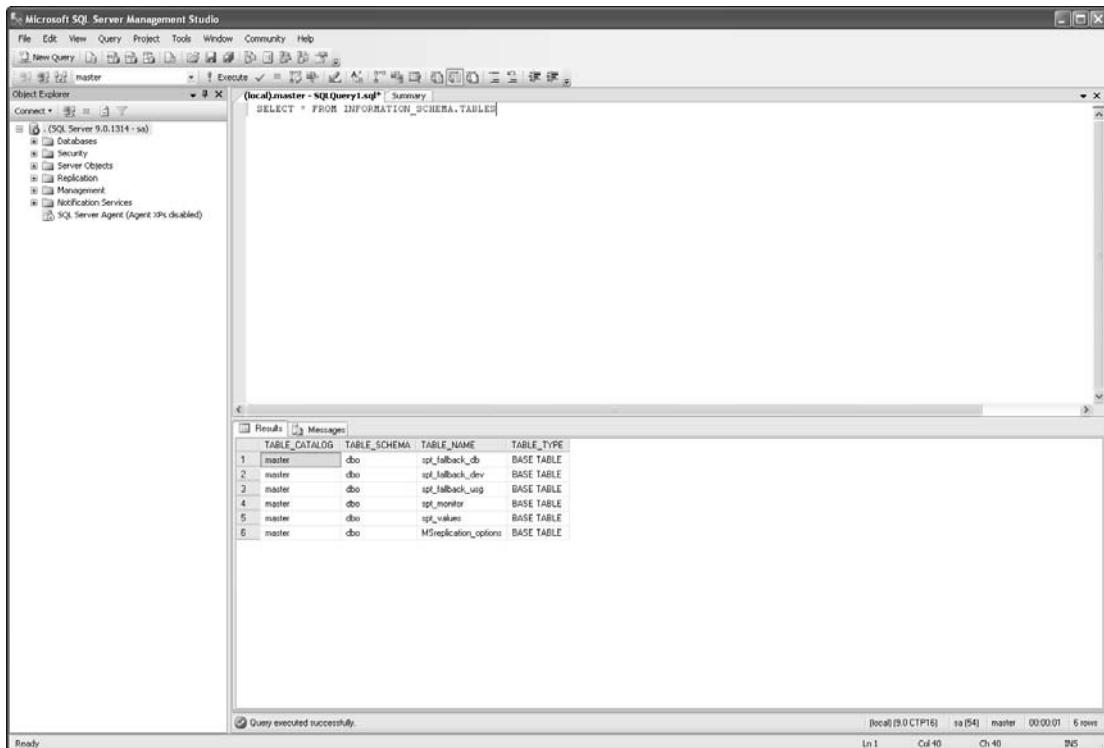


Figure 2-7

Just as it was in the old Query Analyzer (if you're familiar with that tool from SQL Server 2000), the main window is divided into two panes. The top is your original query text; the bottom contains the results.

Now, let's change a setting or two and see how what we get varies. Take a look at the toolbar above the Query window, and check out the set of three icons, highlighted in Figure 2-8.

These control the way you receive output. In order, they are Results to Text, Results to Grid, and Results to File. The same choices can also be made from the Query menu under the Results To sub-menu.

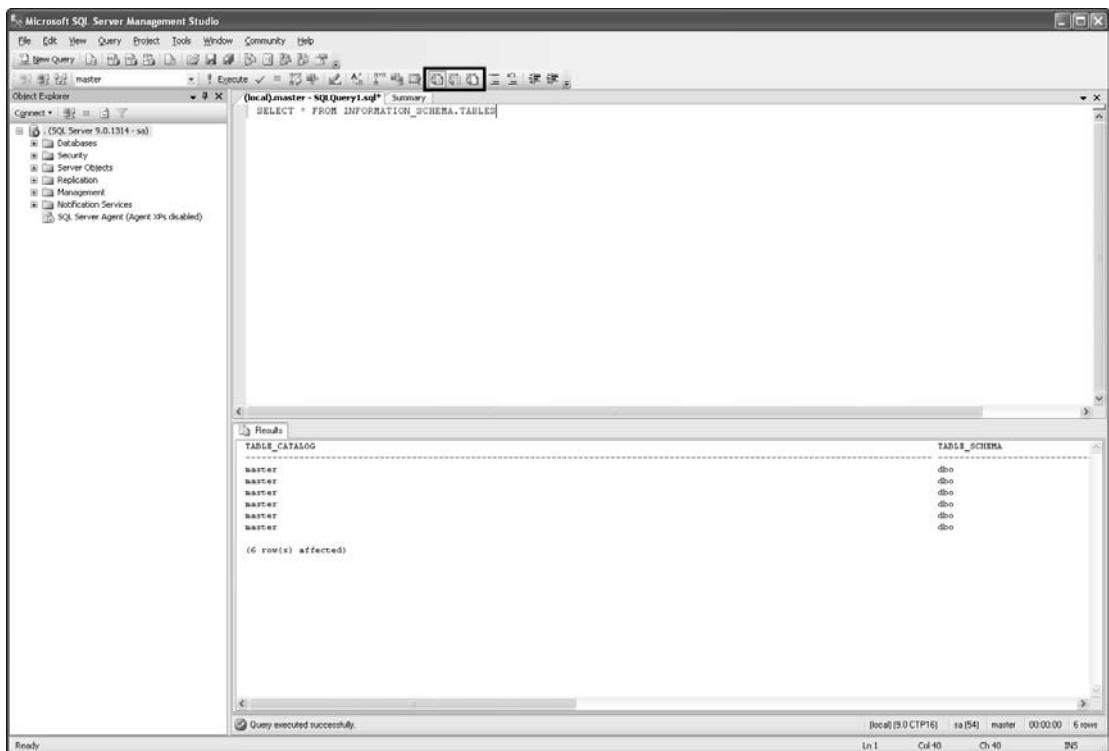


Figure 2-8

Results in Text

The Results in Text option takes all the output from your query and puts it into one page of text results. The page can be of virtually infinite length (limited by the available memory in your system).

I use this output method in several different scenarios:

- When I'm getting only one result set and the results have only fairly narrow columns
- When I want to be able to easily save my results in a single text file
- When I'm going to have multiple result sets, but the results are expected to be small, and I want to be able to see more than one result set on the same page without dealing with multiple scroll bars

Results in Grid

This one divides up the columns and rows into a grid arrangement. Specific things that this option gives us that the Results in Text doesn't include:

- You can resize the column by hovering your mouse pointer on the right border of the column header and then clicking and dragging the column border to its new size. Double-clicking on the right border will result in the auto-fit for the column.
- If you select several cells, and then cut and paste them into another grid (say, Microsoft Excel), they will be treated as individual cells (under the Results in Text option, the cut data would have been pasted all into one cell).
- You can select just one or two columns of multiple rows (under Results in Text, if you select several rows all of the inner rows have every column selected — you can only select in the middle of the row for the first and last row selected).

I use this option for almost everything, since I find that I usually want one of the benefits I just listed.

Results to File

Think of this one as largely the same as Results to Text, but instead of to screen, it routes the output directly to a file. I use this one to generate files I intend to parse using some utility or that I want to easily e-mail.

Show Execution Plan

Every time you run a query, SQL Server parses your query into its component parts and then sends it to the *Query Optimizer*. The Query Optimizer is the part of SQL Server that figures out what is the best way to run your query to balance fast results with minimum impact on other users. When you use the Show Estimated Execution Plan option, you receive a graphical representation and additional information on how SQL Server plans to run your query. Similarly, you can turn on the Include Actual Execution Plan option. Most of the time, this will be the same as the estimated execution plan, but it's possible you'll see differences here for changes that the optimizer decided to make while it was running the query and for changes in the actual cost of running the query versus what the optimizer *thought* was going to happen.

Let's see what a query plan looked like in our simple query. Click on the Include Actual Execution Plan option, and re-execute the query, as shown in Figure 2-9.

Note that you have to actually click on the Execution Plan tab for it to come up, and that your query results are still displayed in whichever way you had selected. The Show Estimated Execution plan option will give you the same output as an Include Actual Execution Plan with two exceptions:

- You get the plan immediately rather than after your query executes.
- While what you see is the actual "plan" for the query, all the cost information is estimated, and the query is not actually run. Under Show Query Plan, the query was physically executed, and the cost information you get is actual rather than estimated.

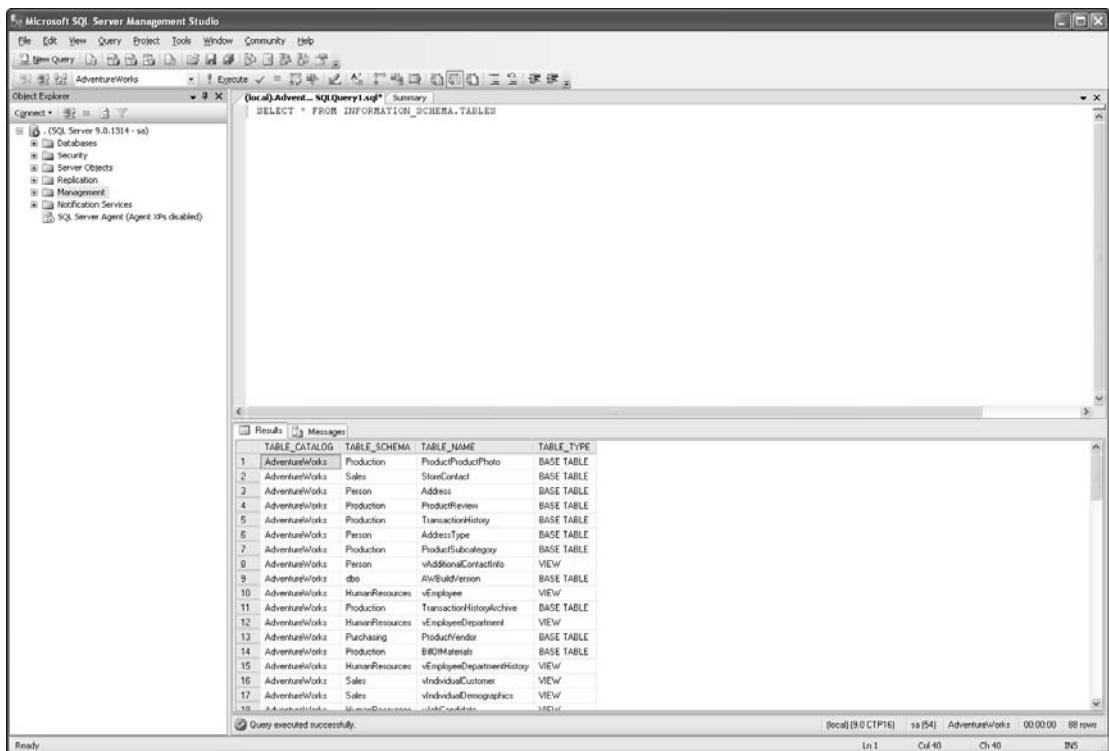


Figure 2-9

The DB Combo Box

Finally, let's take a look at the *DB combo box*. In short, this is where you select the default database that you want your queries to run against for the current window. Initially, the Query window will start with whatever the default database is for the user that's logged in (for sa, that is the master database unless someone has changed it on your system). You can then change it to any other database that the current login has permission to access.

The Object Explorer

This useful little tool allows us to navigate the database, look up object names, and even perform actions such as scripting and looking at the underlying data.

In the example in Figure 2-10, I've expanded the database node all the way down to the listing of tables in the AdventureWorks Database. You can drill down even further to see individual columns (including data type and similar properties) of the tables—a very handy tool for browsing your database.

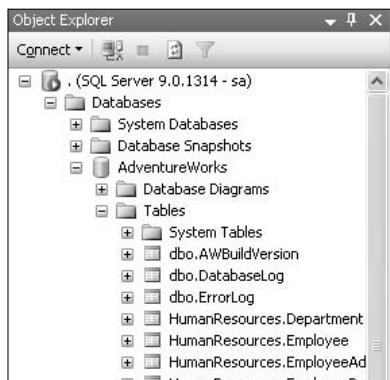


Figure 2-10

SQL Server Business Intelligence Development Studio

The SQL Server Business Intelligence Development Studio is just an incredibly long-winded name for what amounts to a special version of Visual Studio. Indeed, what this tool looks like when it comes up will vary depending on whether you have Visual Studio 2005 installed on your system or not. If you do, it will mix an array of Visual Studio menus in with your SQL Server Analysis Services-related menus and templates.

Business Intelligence Development Studio is yet another tool that received a complete makeover in SQL Server 2005. Indeed, the Analysis Services tools that were there in SQL Server 2000 have been totally replaced. What exists now is far more of a true development environment rather than the relatively narrow cube builder that we had in SQL Server 2000.

From Business Intelligence Development Studio, we can design Integration Services packages (we'll touch on these next), design reports for reporting services, and, of course, work directly with Analysis Services projects. We will be looking at the various services that Business Intelligence Development Studio supports in separate chapters later in this book.

SQL Server Integration Services (SSIS)

Your friend and mine — that's what SSIS (formerly known as Data Transformation Services — or DTS) is. I simply sit back in amazement every time I look at this feature of SQL Server. With SSIS, a tremendous amount of the coding (usually in some client-side language) that had to be done to perform complex data extracts or imports. SSIS allows you to take data from any data source that has an OLE DB or .NET data provider and pump it into a SQL Server table.

While transferring our data, we can also apply what are referred to as *transformations* to that data. Transformations essentially alter the data according to some logical rule(s). The alteration can be as

simple as changing a column name, or as complex as an analysis of the integrity of the data and application of rules to change it if necessary. To think about how this is applied, consider the problem of taking data from a field that allows nulls and moving it to a table that does not allow nulls. With SSIS, you can automatically change out any null values to some other value you choose during the transfer process (for a number, that might be zero, or, for a character, it might be something like “unknown”).

Reporting Services

Reporting Services was first introduced as a Web release after SQL Server 2000 was already out and before SQL Server 2005 hit the market. It has received some major improvements as part of the main SQL Server 2005 product.

Reporting Services provides both a framework and an engine for generating reports. It works in conjunction with the built-in Windows Web server to produce reports in a Web environment. The reports are defined using an XML-based definition language called Report Definition Language (or RDL). The Business Intelligence Development Studio provides a set of templates for generating both simple and complex reports. The reports are written to a RDL file, which is processed on demand by the Reporting Services engine. We will look more fully into Reporting Services in Chapter 17.

Bulk Copy Program (BCP)

If SSIS is “your friend and mine,” then The Bulk Copy Program, or BCP, would be that old friend that we may not see that much any more, but we really appreciate when we do see them.

BCP is a command-line program, and its sole purpose in life is to move formatted data in and out of SQL Server en masse. It was around long before what has now become SSIS was thought of, and while SSIS is replacing BCP for most import/export activity, BCP still has a certain appeal to people who like command-line utilities. In addition, you’ll find an awful lot of SQL Server installations out there that still depend on BCP to move data around fast. We discuss BCP fully in Chapter 18.

SQL Server Profiler

I can’t tell you how many times this one has saved my bacon by telling me what was going on with my server when nothing else would. It’s not something a developer (or even a DBA for that matter) will tend to use every day, but it’s extremely powerful and can be your salvation when you’re sure nothing can save you.

SQL Server Profiler is, in short, a real-time tracing tool. Whereas the Performance Monitor is all about tracking what’s happening at the macro level—system configuration stuff—the Profiler is concerned with tracking specifics. This is both a blessing and a curse. The Profiler can, depending on how you configure your trace, give you the specific syntax of every statement executed on your server. Now, imagine that you are doing performance tuning on a system with 1,000 users. I’m sure you can imagine the reams of paper that would be used to print out the statements executed by so many people in just a minute or two. Fortunately, the Profiler has a vast array of filters to help you narrow things down and track more

specific problems—for example: long-running queries or the exact syntax of a query being run within a stored procedure (which is nice when your procedure has conditional statements that cause it to run different things under different circumstances).

sqlcmd

You won't see *sqlcmd* in your SQL Server program group. Indeed, it's amazing how many people don't even know that this utility (or its older brothers—*osql* and *isql*) is around; that's because it's a console rather than a Windows program.

There are occasionally items that you want to script into a larger command-line process. *sqlcmd* gives you that capability. *sqlcmd* can be very handy—particularly if you make use of files that contain the scripts you want to run under *sqlcmd*. Keep in mind, however, that there are usually tools that can do what you're after from *sqlcmd* much more effectively, and with a user interface that is more consistent with the other things you're doing with your SQL Server.

Once again, for history and being able to understand if people you talk SQL Server with use a different lingo, sqlcmd is yet another new name for this tool of many names. Originally, it was referred to as ISQL. In SQL Server 2000 and 7.0, it was known as osql.

Summary

Most of the tools that you've been exposed to here are not ones you'll use every day. Indeed, for the average developer, only the SQL Server Management Studio will get daily use. Nevertheless, it is important to have some idea of the role that each one can play. Each has something significant to offer you. We will see each of these tools again in our journey through this book.

Note that there are some other utilities available that don't have shortcuts on your Start menu (connectivity tools, server diagnostics, and maintenance utilities), which are mostly admin-related.

3

Basic T-SQL

Well, here we are still in review time. In this chapter, we will take a whirlwind tour of the most fundamental *Transact-SQL* (or *T-SQL*) statements. As with all of the first few chapters of this book, we will be mostly assuming that you know a lot of this story already—so the goal here is to provide review material plus set you up to “fill in the blanks” of your learning.

T-SQL is SQL Server’s own dialect of *Structured Query Language* (or *SQL*). T-SQL received a bit of an overhaul for this release, with many new programming constructs added. Among other things, it was converted to be a Common Language Runtime (CLR) compliant language—in short, it is a .NET language now. While, for SQL Server 2005 we can use any .NET language to access the database, in the end we’re always going to be using some SQL for the root data access, and T-SQL remains our core language for doing things in SQL Server. For purposes of this chapter though, things are pretty much as they always have been—there is very little in the way of changes in this most fundamental statements.

The T-SQL statements that we will learn in this chapter are:

- SELECT
- INSERT
- UPDATE
- DELETE

These four statements are the bread and butter of T-SQL. We’ll learn plenty of other statements as we go along, but these statements make up the basis of T-SQL’s *Data Manipulation Language*—or *DML*. Since you’ll, generally, issue far more commands meant to manipulate (that is, read and modify) data than other types of commands (such as those to grant user rights or create a table), there really isn’t anything more fundamental than these.

In addition, SQL provides for many operators and keywords that help refine your queries. We’ll review some of the most common of these, including *JOINS*, in this chapter.

While T-SQL is unique to SQL Server, the statements you use most of the time are not. T-SQL is entry-level ANSI SQL-92 compliant, which means that it complies up to a certain level of a very wide open standard. What this means to you as a developer is that much of the SQL you're going to learn in this book is directly transferable to other SQL-based database servers such as Sybase (which, long ago, used to share the same code base as SQL Server), Oracle, DB2, and MySQL. Be aware, however, that every RDBMS has different extensions and performance enhancements that it uses above and beyond the ANSI standard. I will try to point out the ANSI versus non-ANSI ways of doing things where applicable. In some cases, you'll have a choice to make—performance vs. portability to other RDBMS systems. Most of the time, however, the ANSI way is as fast as any other option. In such a case, the choice should be clear—stay ANSI compliant.

The Basic SELECT Statement

The SELECT statement and the structures used within it form the basis for the lion's share of all the commands we will perform with SQL Server. Let's look at the basic syntax rules for a SELECT statement:

```
SELECT <column list>
[FROM <source table(s)> [[AS] <table alias>]
[{{FULL|INNER|{LEFT|RIGHT} OUTER|CROSS}} JOIN <next table>
[ON <join condition>] [<additional JOIN clause> ...]]
[WHERE <restrictive condition>]
[GROUP BY <column name or expression using a column in the SELECT list>]
[HAVING <restrictive condition based on the GROUP BY results>]
[ORDER BY <column list>]
[[FOR XML {RAW|AUTO|EXPLICIT|PATH [(<element>)]}{, XMLDATA}{, ELEMENTS}{, BINARY
base 64}]]
[OPTION (<query hint>, [, ...n])]
[;]
```

The SELECT Statement and FROM Clause

The “verb”—in this case a SELECT—is the part of the overall statement that tells SQL Server what we are doing. A SELECT indicates that we are merely reading information, as opposed to modifying it.

The FROM statement specifies the name of the table or tables from which we are getting our data. With these, we have enough to create a basic SELECT statement. Fire up the SQL Server Management Studio and let's take another look at the SELECT statement we ran during the last chapter:

```
SELECT * FROM INFORMATION_SCHEMA.TABLES;
```

Let's look at what we've asked for here. We've asked to SELECT information—you can also think of this as requesting to display information. The * works pretty much as * does everywhere—it's a wildcard. When we say SELECT *, we're saying we want to select every column from the table. Next, the FROM indicates that we've finished saying what items to output, and that we're about to say what the source of the information is supposed to be—in this case, INFORMATION_SCHEMA.TABLES.

INFORMATION_SCHEMA is a special access path that is used for displaying metadata about your system's databases and their contents. **INFORMATION_SCHEMA** has several parts that can be specified after a period, such as **INFORMATION_SCHEMA.SCHEMATA** or **INFORMATION_SCHEMA.VIEWS**. These special access paths to the metadata of your system have been put there so you won't have to use what are called "system tables." **INFORMATION_SCHEMA** is ANSI compliant in most cases (though SQL Server's implementation of it occasionally has its own twists of things). SQL Server also has a set of "sys" functions for things that are a superset of what **INFORMATION_SCHEMA** might give you. These "sys" functions are far more robust than **INFORMATION_SCHEMA** but are not ANSI compliant.

Now let's try taking a little bit more specific information. Let's say all we want is a list of all our customers by last name:

```
USE AdventureWorks
SELECT LastName FROM Person.Contact;
```

The USE AdventureWorks command here is an instruction that changes what database is considered "current." I've put it here to make sure you are in the correct database, the one that has this particular schema.table combination already installed.

Your results would look something like:

```
Achong
Abel
Abercrombie
...
He
Zheng
Hu
```

Note that I've snipped the rows out of the middle for brevity—you should have 19972 rows there. Since the name of each customer is all that we want, that's all that we've selected.

This query is pretty simple, but as our queries grow longer, we may want to "alias" our table—we can do this by adding the alias we want to use after the table name.

```
SELECT LastName FROM Person.Contact c;
```

In this case, we've used an alias, but not really done anything with it. In practice, we would actually use the alias to refer to the table elsewhere in the query:

```
SELECT c.LastName FROM Person.Contact c;
```

In this version, we've given the alias "c" to **Person.Contact**. Back in the select list, we've explicitly said that we want the last name from the **Person.Contact** table by using the alias. This probably doesn't seem worth a lot with a simple query like this, but, as we'll see shortly when we look at joins, aliases can make your code much more readable. Indeed, in some more advanced query structures, they are actually required.

*Many SQL writers have the habit of cutting their queries short and always selecting every column by using a * in their selection criteria. This is another one of those habits to resist. While typing in a * saves you a few moments of typing out the column names that you want, it also means that more data has to be retrieved than is really necessary. In addition, SQL Server must go and figure out just how many columns * amounts to and what specifically they are. You would be surprised at just how much this can drag down your application's performance and that of your network. In short, a good rule to live by is to select what you need—that is, exactly what you need. No more, no less.*

The JOIN Clause

In my previous books, this one has received its own chapter, so I do want to take some time and be careful with this one.

A *normalized* database is one where the data has been broken out from larger tables into many smaller tables for the purpose of eliminating repeating data, saving space, improving performance, and increasing data integrity. It's great stuff and vital to relational databases; however, it also means that you wind up getting your data from here, there, and everywhere. **JOINS** are all about defining how to take multiple tables and combine them into one result set. There are multiple forms to the **JOIN** clause, and they change the way that the tables you are joining interact. In addition, there is an older syntax for performing **JOINS**—you'll take a look at that as one of the last things in the chapter (you'll need a few more concepts before you deal with that).

JOIN Basics

When we are operating in a normalized environment, we frequently run into situations in which not all of the information that we want is in one table. In other cases, all the information we want returned is in one table, but the information we want to place conditions on is in another table. These situations are where the **JOIN** clause comes in.

How exactly does a **JOIN** put the information from two tables into a single result set? Well, that depends on how you tell it to put the data together—that's why there are four different kinds of **JOINS**. The thing that all **JOINS** have in common is that they match one record up with one or more other records to make a record that is a superset created by the combined columns of both records.

For example, let's take a record from a table we'll call `Films`:

FilmID	FilmName	YearMade
1	My Fair Lady	1964

Now let's follow that up with a record from a table called `Actors`:

FilmID	FirstName	LastName
1	Rex	Harrison
1	Audrey	Hepburn

With a `JOIN`, we could create one record from two records found in totally separate tables:

FilmID	FilmName	YearMade	FirstName	LastName
1	My Fair Lady	1964	Rex	Harrison
1	My Fair Lady	1964	Audrey	Hepburn

Indeed, in this case, we have created two records from a total of three (one in one table and two in another table). We have used the single record in the `Films` table multiple times — once for each actor it is associated with (as “joined” by the `FilmID`).

The examples we have used here with such a limited data set would actually yield the same results no matter what kind of `JOIN` we have. Let’s move on now and look at the specifics of the different `JOIN` types.

INNER JOINS

`INNER JOINs` are far and away the most common kind of `JOIN`. They match records together based on one or more common fields, as do most `JOINS`, but an `INNER JOIN` returns only the records where there are matches for whatever field(s) you have said are to be used for the `JOIN`. In our previous examples, every record has been included in the result set at least once, but this situation is rarely the case in the real world.

Let’s modify our tables and see what we would get with an `INNER JOIN`. Here’s our `Films` table:

FilmID	FilmName	YearMade
1	My Fair Lady	1964
2	Unforgiven	1992

And our `Actors` table:

FilmID	FirstName	LastName
1	Rex	Harrison
1	Audrey	Hepburn
2	Clint	Eastwood
5	Humphrey	Bogart

Chapter 3

Using an `INNER JOIN`, our result set would look like this:

FilmID	FilmName	YearMade	FirstName	LastName
1	My Fair Lady	1964	Rex	Harrison
1	My Fair Lady	1964	Audrey	Hepburn
2	Unforgiven	1992	Clint	Eastwood

Notice that Bogey was left out of this result set. That's because he didn't have a matching record in the `Films` table. If there isn't a match in both tables, then the record isn't returned.

INNER JOINS are *exclusive* in nature—that is, any row that doesn't match in both tables is inherently excluded from the final result set.

Enough theory—let's try this out in code.

First, let's look at a simplified version of the syntax:

```
SELECT <select list>
FROM <first_table> [<alias>]
<join_type> <second_table> [<alias>]
[ON <join_condition>] [,]
```

This is the ANSI syntax, and you'll have much better luck with it on non-SQL Server database systems than you will if you use the older syntax I mentioned earlier (which still used by many developers today—again, we'll discuss it later in the chapter).

Fire up Management Studio and take a test drive of `INNER JOINs` using the following code against `AdventureWorks`:

```
SELECT *
FROM HumanResources.Employee e
INNER JOIN HumanResources.Employee m
    ON e.ManagerID = m.EmployeeID;
```

The results of this query are far too wide to print in this book, but you should get back a little less than 300 rows. Notice several things about the results:

- ❑ Not only did we join two tables, but we are actually able to join a table to itself—this allows us to compare different rows within the same table (in this case, we're treating one look at the table as being a table of managers, and another look at the table as being all employees). To do this, we *must* use table aliases (I chose `e` and `m` in this case).
- ❑ All columns were returned from both references to the table. If we had been referencing separate tables, we would have seen the column names from both. While we see both sets of columns, we have no way of delineating which is which (you'll see the same column name twice, but often with different values).

Let's cut right to the kind of results we might be looking for, and fully expand our `JOIN` logic:

```
SELECT e.EmployeeID,
       ce.FirstName,
       ce.LastName,
       m.EmployeeID AS ManagerID,
       cm.FirstName AS ManagerFirst,
       cm.LastName AS ManagerLast
  FROM HumanResources.Employee e
 INNER JOIN HumanResources.Employee m
    ON e.ManagerID = m.EmployeeID
 INNER JOIN Person.Contact ce
    ON e.ContactID = ce.ContactID
 INNER JOIN Person.Contact cm
    ON m.ContactID = cm.ContactID;
```

This is pretty heady stuff—pulling together most every concept there is surrounding `INNER JOINS`, so hang with me as we walk through some of what's going on here by looking at individual parts.

```
FROM HumanResources.Employee e
INNER JOIN HumanResources.Employee m
  ON e.ManagerID = m.EmployeeID
```

This is the original join—one where we are joining a table to itself. When we have a table that joins back to itself for some reason, we refer to that table as *self-referencing*. Since there are two instances of this table, we *must* use a table alias, so we know which instance of this table we're referring to at every point in our query.

```
INNER JOIN Person.Contact ce
  ON e.ContactID = ce.ContactID
INNER JOIN Person.Contact cm
  ON m.ContactID = cm.ContactID
```

I'm joining to the contact table to get the names of the employees. In this particular database design, the architect designed to split contact information into its own table since an employee may, for example, also be a customer. Notice, however, that I needed to join to the Contact table *twice*—once to join to the employee's name and once to join to the manager's name.

```
SELECT e.EmployeeID,
       ce.FirstName,
       ce.LastName,
       m.EmployeeID AS ManagerID,
       cm.FirstName AS ManagerFirst,
       cm.LastName AS ManagerLast
```

Notice several changes to the select list. First, I pared it down to the columns we are likely interested in (what employees are reporting to which managers). Since we have more than one instance of both tables, it was critical that I alias not only the table name but also which instance of the columns I'm outputting.

Under the heading of “one more thing”: `INNER JOINS` are the default. As such, the `INNER` keyword is optional. Indeed, you will find that most SQL developers leave it off to make the query less wordy. So, we would have made our query look like this:

```
SELECT e.EmployeeID,
       ce.FirstName,
       ce.LastName,
```

Chapter 3

```
m.EmployeeID AS ManagerID,  
cm.FirstName AS ManagerFirst,  
cm.LastName AS ManagerLast  
FROM HumanResources.Employee e  
JOIN HumanResources.Employee m  
ON e.ManagerID = m.EmployeeID  
JOIN Person.Contact ce  
ON e.ContactID = ce.ContactID  
JOIN Person.Contact cm  
ON m.ContactID = cm.ContactID;
```

And it would be considered just as proper and produce the same result.

OUTER JOINS

This type of JOIN is something of the exception rather than the rule. This is definitely not because they don't have their uses, but rather because:

- You, more often than not, want the kind of exclusiveness that an INNER JOIN provides.
- Many SQL writers learn INNER JOINS and never go any further—they simply don't understand the OUTER variety.
- There are often other ways to accomplish the same thing.
- They are often simply forgotten about as an option.

Whereas INNER JOINS are exclusive in nature, OUTER and, as we'll see later in this chapter, FULL JOINS are inclusive. It's a tragedy that people don't get to know how to make use of OUTER JOINS, because they make seemingly difficult questions simple. They can also often speed performance when used instead of other options that might produce the same result.

JOINS have the concept of sides—a left and a right. The first named table is considered as being on the left, and the second named table is considered to be on the right. With INNER JOINS these are a passing thought at most because both sides are always treated equally. With OUTER JOINS, however, understanding your left from your right is absolutely critical. When you look at it, it seems very simple because it is very simple, yet many query mistakes involving OUTER JOINS stem from not thinking through your left from your right.

Let's look back at a variant of the earliest version of our employee/manager query.

```
SELECT e.EmployeeID, m.EmployeeID AS ManagerID  
FROM HumanResources.Employee e  
INNER JOIN HumanResources.Employee m  
ON e.ManagerID = m.EmployeeID;
```

There isn't a tremendous amount to see here yet, but note the number of rows you get back—for me, it's 289 rows (this with the default install of AdventureWorks).

Now, alter the query just slightly:

```
SELECT e.EmployeeID, m.EmployeeID AS ManagerID  
FROM HumanResources.Employee e  
LEFT OUTER JOIN HumanResources.Employee m  
ON e.ManagerID = m.EmployeeID;
```

What we're saying here is to include *all* records from the LEFT side regardless of whether there is a match on the right side or not. To be more specific, we're saying include *all* employee records, and also include those manager records where there is a match.

Check the results on this and you should come up with one more row than we had before (290 now in my database). Scan the result and see if you can find the "new" record.

EmployeeID	ManagerID
109	NULL

It seems that Employee 109 doesn't report to anyone (probably the CEO, eh?). Pay particular attention to how SQL Server addressed the issue of listing a column from the right side table even though there was no data—it supplied a NULL value. Later on, we'll see how we can look for NULL values on one side of the join to look for things like missing rows in one side of the join.

If we turned this around to a RIGHT JOIN, then, for this particular recordset, we would get the same result as an INNER JOIN because there are no records on the RIGHT (the manager side) that do not exist on the left (the employee side). The concept is, however, just the same—with a RIGHT join you want to be *inclusive* of all rows on the right, and include only those rows on the left where there is a match.

FULL JOINS

Think of this one as something of a LEFT and a RIGHT join coming together. With a FULL join, you are telling SQL Server to include all rows on *both* sides of the join. AdventureWorks doesn't give me any really great examples to show you this one, but since the concept is fairly easy once you already understand outer joins, we'll just toss together a pretty simple demonstration:

```

CREATE TABLE Film
(FilmID          int            PRIMARY KEY,
 FilmName        varchar(20)    NOT NULL,
 YearMade        smallint       NOT NULL
);

CREATE TABLE Actors
(FilmID          int            NOT NULL,
 FirstName       varchar(15)    NOT NULL,
 LastName        varchar(15)    NOT NULL,
 CONSTRAINT PKActors PRIMARY KEY(FilmID, FirstName, LastName)
);

INSERT INTO Film
VALUES
    (1, 'My Fair Lady', 1964);
INSERT INTO Film
VALUES
    (2, 'Unforgiven', 1992);

INSERT INTO Actors
VALUES
    (1, 'Rex', 'Harrison');
INSERT INTO Actors
VALUES
    (1, 'Audrey', 'Hepburn');

```

Chapter 3

```
INSERT INTO Actors
VALUES
(3, 'Anthony', 'Hopkins');
```

Don't worry too much that we haven't discussed some of these statements yet. Beyond the fact that you likely have a basic level of knowledge already (if you need a refresher on the rudimentary, consider checking out *Beginning SQL Server 2005 Programming*), you're really just building a sample to work with for your `FULL JOIN` for now — we'll get into the details of some of the other statements later in this chapter and on into the next.

Okay, now let's run a `FULL JOIN` and see what we get:

```
SELECT *
FROM Film f
FULL JOIN Actors a
ON f.FilmID = a.FilmID;
```

When you check the results, you'll see data that has been joined where they match, data from just the left if that's all there is (and nulls for the columns on the right), and data from just the right if that happens to be all there is (and, of course, nulls for the columns on the left).

FilmID	FilmName	YearMade	FilmID	FirstName	LastName
1	My Fair Lady	1964	1	Audrey	Hepburn
1	My Fair Lady	1964	1	Rex	Harrison
2	Unforgiven	1992	NULL	NULL	NULL
NULL	NULL	NULL	3	Anthony	Hopkins

(4 row(s) affected)

CROSS JOIN

Our last joins, `CROSS JOINS`, are very strange critters indeed. A `CROSS JOIN` differs from other `JOINS` in that there is no `ON` operator and in that it joins every record on one side of the `JOIN` with every record on the other side of the `JOIN`. In short, you wind up with a Cartesian product of all the records on both sides of the `JOIN`. The syntax is the same as any other `JOIN` except that it uses the keyword `CROSS` (instead of `INNER`, `OUTER`, or `FULL`), and that it has no `ON` operator.

So, let's say we were just playing games and wanted to mix every film with every actor:

```
SELECT *
FROM Film f
CROSS JOIN Actors a;
```

We get every record in the `Film` table matched with *every* actor in the `actors` table:

FilmID	FilmName	YearMade	FilmID	FirstName	LastName
1	My Fair Lady	1964	1	Audrey	Hepburn
1	My Fair Lady	1964	1	Rex	Harrison
1	My Fair Lady	1964	3	Anthony	Hopkins
2	Unforgiven	1992	1	Audrey	Hepburn
2	Unforgiven	1992	1	Rex	Harrison
2	Unforgiven	1992	3	Anthony	Hopkins

(6 row(s) affected)

Now, this has to bring out the question of “Why would you ever want this?” Good question. The answer is difficult because it’s very situational. To date, I’ve seen CROSS JOINS used in just two situations:

- ❑ **Sample Data**—CROSS JOINS are good for putting together small sets of data and then mixing the two sets of data together in every possible way so that you get a much larger sample set to work with.
- ❑ **Scientific Data**—I believe this, again, has to do with samples, but I know there are a number of scientific calculations that make use of Cartesians. I’m told that doing CROSS JOINS is a way of “preparing” data for some types of analysis. I’m not going to pretend to understand the statistics of it, but I know that it’s out there.

The end story is—they are only very, very rarely used, but keep them in mind in case you need them!

The WHERE Clause

The WHERE clause allows you to place conditions on what is returned to you. What we have seen thus far is unrestricted information in the sense that every row in the table(s) specified has been included in our results unless it was filtered out by the nature of a JOIN. For now, we’re going to stick with the single table scenario—we’ll reintroduce JOINS later in this section. Unrestricted queries are very useful for populating things like list boxes and combo boxes, and in other scenarios where you are trying to provide a *domain listing*.

For our purposes, don’t confuse a domain with that of a Windows domain. A domain listing is an exclusive list of choices. For example, if you want someone to provide you with information about a state in the U.S., you might provide them with a list that limits the domain of choices to just the 50 states.

With unrestricted joins, we also can get full listings of associated data.

Now, however, we want to try looking for more specific information. See if you can come up with a query that returns just the name, product number, and reorder point for a product with the ProductID 356.

Let’s break it down and build a query one piece at a time. First, we’re asking for information to be returned, so we know that we’re looking at a SELECT statement. Our statement of what we want indicates that we would like the product name, product number, and reorder point, so we’re going to have to know what the column names are for these pieces of information. We’re also going to need to know out of which table or tables we can retrieve these columns.

I’m going to go ahead and point you at the Production.Product table this time around (later on in the chapter, we’ll take a look at how we could find out what tables are available if we didn’t already know). The Production.Product table has several columns. To give us a quick listing of our column options, we can study the Object Explorer tree of the Production.Product table from Management Studio. To open this screen in the Management Studio, click on the Tables member underneath the AdventureWorks database. Then expand the Production.Product and Columns nodes. As in Figure 3-1, you will see all of the columns along with its data type and nullability options. Again, we’ll see some other methods of finding this information a little later in the chapter.

Chapter 3

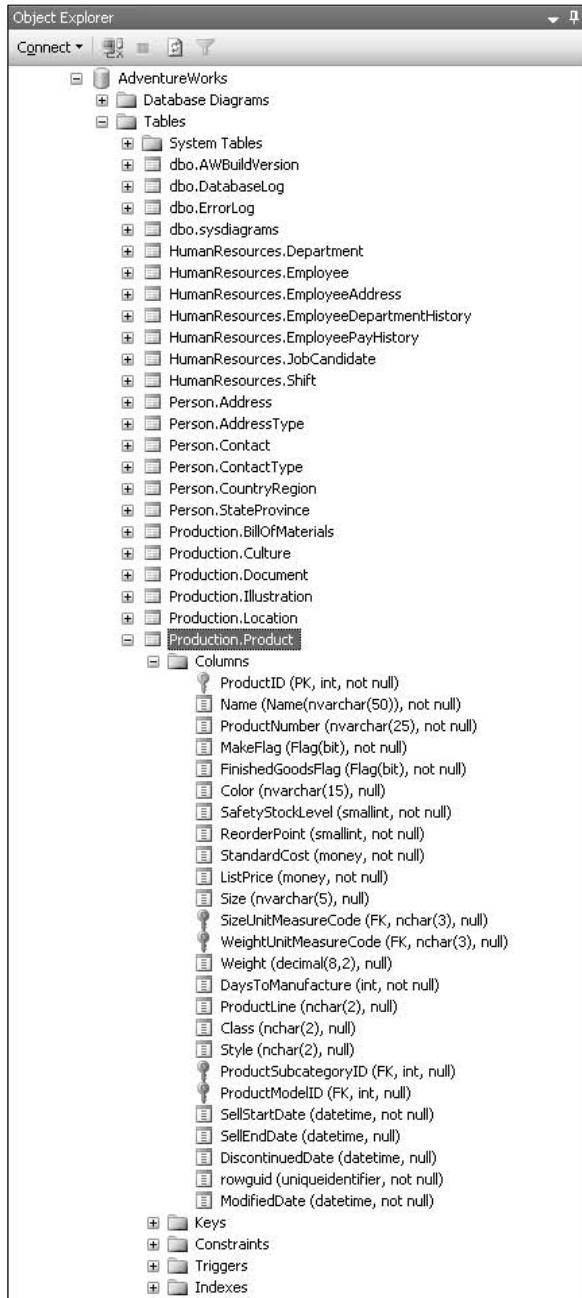


Figure 3-1

We don't have a column called product name, but we do have one that's probably what we're looking for: Name (original, eh?). The other two columns are, except for the missing space in between the two words, just as easy to identify.

As Such, our `Products` table is going to be the place we get our information `FROM`, and the `Name`, `ProductNumber`, and `ReorderPoint` columns will be the specific columns from which we'll get our information:

```
SELECT Name, ProductNumber, ReorderPoint  
FROM Production.Product;
```

This query, however, still won't give us the results that we're after — it will still return too much information. Run it — you'll see that it still returns every record in the table rather than just the one we want.

If the table has only a few records and all we want to do is take a quick look at it, this might be fine. But what if you had 100,000 or 1,000,000 records? What we're after is a conditional statement that will limit the results of our query to just one product identifier — 356. That's where the `WHERE` clause comes in. The `WHERE` clause immediately follows the `FROM` clause and defines what conditions a record has to meet before it will be shown. For our query, we would want the `ProductID` to be equal to 356, so let's finish our query:

```
SELECT Name, ProductNumber, ReorderPoint  
FROM Production.Product  
WHERE ProductID = 356;
```

Run this query against the AdventureWorks database, and you should come up with:

Name	ProductNumber	ReorderPoint
LL Grip Tape	GT-0820	600

(1 row(s) affected)

This time we've gotten back precisely what we wanted — nothing more, nothing less. In addition, this query runs much faster than the first query.

Let's take a look at all the operators we can use with the `WHERE` clause:

Operator	Example Usage	Effect
=, >, <, >=, <=, <>, !=, !>, !<	<Column Name> = <Other Column Name> <Column Name> = 'Bob'	Standard comparison operators — these work as they do in pretty much any programming language with a couple of notable points: <ol style="list-style-type: none">1. What constitutes "greater than," "less than," and "equal to" can change, depending on the collation order you have selected. For example, "ROMEY" = "romeY" in places where case-insensitive sort order has been selected, but "ROMEY" < > "romeY" in a case-sensitive situation.2. != and <> both mean "not equal." !< and !> mean "not less than" and "not greater than," respectively.

Table continued on following page

Chapter 3

Operator	Example Usage	Effect
AND, OR, NOT	<Column1> = <Column2> AND <Column3> >= <Column 4> <Column1> != "MyLiteral" OR <Column2> = "MyOtherLiteral"	Standard Boolean logic. You can use these to combine multiple conditions into one WHERE clause. NOT is evaluated first, then AND, then OR. If you need to change the evaluation order, you can use parentheses. Note that XOR is not supported.
BETWEEN	<Column1> BETWEEN 1 AND 5	Comparison is TRUE if the first value is between the second and third values inclusive. It is the functional equivalent of A>=B AND A<=C. Any of the specified values can be column names, variables, or literals.
LIKE	<Column1> LIKE "ROM%"	Uses the % and _ characters for wildcarding. % indicates that a value of any length can replace the % character. _ indicates that any one character can replace the _ character. Enclosing characters in [] symbols indicates that any single character within the [] is okay ([a-c] means a, b, and c are okay. [ab] indicates a or b is okay). ^ operates as a NOT operator — indicating that the next character is to be excluded.
IN	<Column1> IN (List of Numbers) <Column1> IN ("A", "b", "345")	Returns TRUE if the value to the left of the IN keyword matches any of the values in the list provided after the IN keyword. This is frequently used in subqueries, which we will look at in Chapter 6.
ALL, ANY, SOME	<column expression> (comparison operator) <ANY SOME> (subquery)	These return TRUE if any or all (depending on which you choose) values in a subquery meet the comparison operator (e.g., <, >, =, >=) condition. ALL indicates that the value must match all the values in the set. ANY and SOME are functional equivalents and will evaluate to TRUE if the expression matches any value in the set.
EXISTS	EXISTS (subquery)	Returns TRUE if at least one row is returned by the subquery. Again, we'll look into this one further in Chapter 6.

Now, as promised, let's mix this back in with JOINS. This really doesn't work any differently from the WHERE we've already looked at—just add it to the end of the statement. You can use the table name or table alias to specify what table a specific is coming from if there are redundant column names (if a column exists only once across all tables, you can use the column name without a table qualifier—this is the way that things will work a large percentage of the time, but I prefer to table prefix my WHERE clause restrictions just to be clear where the source of the data is).

Let's look at complex JOIN we did showing all employees and their managers. This time, we want to limit the list to those employees managed by Jo Brown. Since we know we are restricting the list to just employees of Jo Brown, we do not need to include her name in the result set, so we'll edit that part out of the select list:

```
SELECT e.EmployeeID,
       ce.FirstName,
       ce.LastName
  FROM HumanResources.Employee e
  JOIN HumanResources.Employee m
    ON e.ManagerID = m.EmployeeID
  JOIN Person.Contact ce
    ON e.ContactID = ce.ContactID
  JOIN Person.Contact cm
    ON m.ContactID = cm.ContactID
 WHERE cm.FirstName = 'Jo'
   AND cm.LastName = 'Brown';
```

We've mixed several concepts here. First, notice that we are placing WHERE clause restrictions on columns that we are not using in our select list—indeed, we do not utilize that particular table for anything other than a source for filtering our data!

Check out what we get:

EmployeeID	FirstName	LastName
1	Guy	Gilbert
57	Annik	Stahl
80	Rebecca	Laszlo
89	Margie	Shoop
129	Mark	McArthur
137	Britta	Simon
157	Brandon	Heidepriem
162	Jose	Lugo
175	Suchitra	Mohan
213	Chris	Okelberry
235	Kim	Abercrombie
247	Ed	Dudenhoefer

(12 row(s) affected)

So, out of what were originally 289 rows, we are down to just 12.

ORDER BY

Most of the queries that you've run thus far have come out in something resembling alphabetical or numerical order. Is this an accident? It will probably come as somewhat of a surprise to you, but the answer to that is yes. If you don't say that you want a specific sorting on the results of a query, then you get the data in the order that SQL Server decides to give it to you. This will always be based on what SQL Server decided was the lowest cost way to gather the data. It will usually be based either on the physical order of a table or on one of the indexes that SQL Server used to find your data.

Chapter 3

Think of an `ORDER BY` clause as being a “sort by.” It gives you the opportunity to define the order in which you want your data to come back. You can use any combination of columns in your `ORDER BY` clause as long as they are columns (or derivations of columns) found in the tables within your `FROM` clause.

Let’s look at this query:

```
SELECT Name, ProductNumber, ReorderPoint  
FROM Production.Product;
```

This will produce the following results:

Name	ProductNumber	ReorderPoint
Adjustable Race	AR-5381	750
Bearing Ball	BA-8327	750
...		
...		
Road-750 Black, 48	BK-R19B-48	75
Road-750 Black, 52	BK-R19B-52	75
(504 row(s) affected)		

As it happened, our query result set was sorted in `ProductID` order. Why? Because SQL Server decided that the best way to look at this data was by using an index that sorts the data by `ProductID`. That just happened to be what created the lowest cost (in terms of CPU and I/O) query. Had we run this exact query when the table had grown to a much larger size, SQL Server might have chosen an entirely different execution plan, and as such might have sorted the data differently. We could force this sort order by changing our query to this:

```
SELECT Name, ProductNumber, ReorderPoint  
FROM Production.Product  
ORDER BY Name;
```

Note that the `WHERE` clause isn’t required. It can either be there or not, depending on what you’re trying to accomplish—just remember that, if you do have a `WHERE` clause, it goes before the `ORDER BY` clause.

Unfortunately, that last query doesn’t really give us anything different, so we don’t see what’s actually happening. Let’s change the query to sort the data differently—by the `ProductNumber`:

```
SELECT Name, ProductNumber, ReorderPoint  
FROM Production.Product  
ORDER BY ProductNumber;
```

Now our results are quite different. It’s the same data, but it’s been substantially rearranged:

Name	ProductNumber	ReorderPoint
Adjustable Race	AR-5381	750
Bearing Ball	BA-8327	750
LL Bottom Bracket	BB-7421	375
ML Bottom Bracket	BB-8107	375
...		

```

...
Classic Vest, L           VE-C304-L      3
Classic Vest, M           VE-C304-M      3
Classic Vest, S           VE-C304-S      3
Water Bottle - 30 oz.     WB-H098       3

(504 row(s) affected)

```

SQL Server still chose the least cost method of giving us our desired results, but the particular set of tasks it actually needed to perform changed somewhat because the nature of the query changed.

It's worth noting that we could also do our sorting using numeric fields.

Don't get fooled by the preceding query. In the subset of data I've shown there, it's easy to get the impression that the data is sorted by the reorder point field, since it happens to be steadily declining—that is pure coincidence, and, if you look at the data you actually get back (I only show some of the data here in the book for the sake of brevity), you'll find reorder point to be unsorted.

We'll make a slight change to what columns we're after, changing to `Weight` this time as an additional column:

```

SELECT Name, ProductNumber, Weight
FROM Production.Product
WHERE Weight > 800
ORDER BY Weight DESC;

```

This one results in:

Name	ProductNumber	Weight
LL Road Rear Wheel	RW-R623	1050.00
ML Road Rear Wheel	RW-R762	1000.00
LL Road Front Wheel	FW-R623	900.00
HL Road Rear Wheel	RW-R820	890.00
ML Road Front Wheel	FW-R762	850.00

(5 row(s) affected)

Notice several things in this query. First, we've made use of many of the things that we've talked about up to this point. We've combined a `WHERE` clause condition with an `ORDER BY`. Second, we've added something new in our `ORDER BY` clause—the `DESC` keyword. This tells SQL Server that our `ORDER BY` should work in descending order, rather than the default of ascending. (If you want to explicitly state that you want it to be ascending, use `ASC`.)

Okay, let's do one more, but this time, let's sort based on multiple columns. To do this, all we have to do is add a comma, followed by the next column by which we want to sort.

Suppose, for example, that we want to get a listing of every order that was placed between July 8th and 9th in 1996. To add a little bit to this, though, let's further say that we want the orders sorted by date, and we want a secondary sort based on the `CustomerID`. Just for grins, we'll toss in yet another little twist: we want the most recent orders first (descending date).

Chapter 3

Our query would look like this:

```
SELECT OrderDate, CustomerID
FROM Sales.SalesOrderHeader
WHERE OrderDate BETWEEN '2001-07-08' AND '2001-07-09'
ORDER BY OrderDate DESC, CustomerID;
```

This time, we get data that is sorted two ways:

OrderDate	CustomerID
2001-07-09 00:00:00.000	11025
2001-07-09 00:00:00.000	11238
2001-07-09 00:00:00.000	16629
2001-07-09 00:00:00.000	25861
2001-07-09 00:00:00.000	27577
2001-07-09 00:00:00.000	27666
2001-07-08 00:00:00.000	13258
2001-07-08 00:00:00.000	14560
2001-07-08 00:00:00.000	16607

(9 row(s) affected)

Our Customer IDs, since we didn't say anything to the contrary, were still sorted in ascending order (the default), but you can see that we got the orders dated the 9th before those dated the 8th—descending order.

While we usually sort the results based on one of the columns that we are returning, it's worth noting that the ORDER BY clause can be based on any column in any table used in the query, regardless of whether it is included in the SELECT list.

Aggregating Data Using the GROUP BY Clause

With ORDER BY, we have kind of taken things out of order compared with how the SELECT statement reads at the top of the chapter. Let's review the overall statement structure:

```
SELECT <column list>
[FROM <source table(s)>]
[WHERE <restrictive condition>]
[GROUP BY <column name or expression using a column in the SELECT list>]
[HAVING <restrictive condition based on the GROUP BY results>]
[ORDER BY <column list>]
[[FOR XML] [RAW, AUTO, EXPLICIT][, XMLDATA][, ELEMENTS][, BINARY base 64]]
[OPTION (<query hint>, [, ...n])]
```

Why, if ORDER BY comes last, did we look at it before the GROUP BY? There are two reasons:

- ❑ ORDER BY is used far more often than GROUP BY, so I want you to have more practice with it.
- ❑ I want to make sure that you understand that you can mix and match all of the clauses after the FROM clause as long as you keep them in the order in which SQL Server expects them (as defined in the syntax definition).

The GROUP BY clause is used to aggregate information. Let's look at a simple query without a GROUP BY. Let's say that we want to know how many parts were ordered on a given set of orders:

```
SELECT SalesOrderID, OrderQty  
FROM Sales.SalesOrderDetail  
WHERE SalesOrderID BETWEEN 43684 AND 43686;
```

This yields a result set of:

SalesOrderID	OrderQty
43684	2
43684	2
43684	1
43684	2
43684	1
43684	1
43685	3
43685	1
43685	1
43685	1
43686	3
43686	1
43686	1

(13 row(s) affected)

Even though we've only asked for three orders, we're seeing each individual line of detail from the order. We can either get out our adding machine or we can make use of the GROUP BY clause with an aggregator—in this case, we'll use SUM().

```
SELECT SalesOrderID, SUM(OrderQty)  
FROM Sales.SalesOrderDetail  
WHERE SalesOrderID BETWEEN 43684 AND 43686  
GROUP BY SalesOrderID;
```

This gets us what we were looking for:

SalesOrderID
43684
43685
43686

(3 row(s) affected)

Chapter 3

As you would expect, the `SUM` function returns totals—but totals of what? If we didn't supply the `GROUP BY` clause, the `SUM` would have been of all the values in all of the rows for the named column. In this case, however, we did supply a `GROUP BY`, so the total provided by the `SUM` function is the total in each group.

We can also group based on multiple columns. To do this, we just add a comma and the next column name much the same as we do the `ORDER BY` clause.

Note that, once we use a `GROUP BY`, every column in the `SELECT` list has to either be part of the `GROUP BY` or be an aggregate. So, what does this mean? Let's find out.

Aggregates

When you consider that they usually get used with a `GROUP BY` clause, it's probably not surprising that *aggregates* are functions that work on groups of data. For example, in one of the preceding queries, we got the sum of the `Quantity` column. The sum is calculated and returned on the selected column for each group defined in the `GROUP BY` clause—in this case, just `OrderID`. There are a wide range of aggregates available, but let's play with the most common.

While aggregates show their power when used with a `GROUP BY` clause, they are not limited to grouped queries—if you include an aggregate without a `GROUP BY`, then the aggregate will work against the entire result set (all the rows that match the `WHERE` clause). The catch here is that, when not working with a `GROUP BY`, some aggregates can only be in the `SELECT` list with other aggregates—that is, they can't be paired with a column name in the `SELECT` list unless you have a `GROUP BY`. For example, unless there is a `GROUP BY`, `AVG` can be paired with `SUM`, but not with a specific column.

AVG

This one is for computing averages. 'Nuff said.

MIN/MAX

Bet you can guess these two. Yes, these grab the minimum and maximum amounts for each grouping for a selected column. Again, nothing special.

COUNT(Expression | *)

The `COUNT (*)` function is about counting the rows in a query. To begin with, let's go with one of the most common varieties of queries:

```
SELECT COUNT(*)  
FROM HumanResources.Employee  
WHERE EmployeeID = 3
```

The recordset you get back looks a little different from what you're used to from earlier queries:

```
-----  
1
```

```
(1 row(s) affected)
```

Let's look at the differences. First, as with all columns that are returned as a result of a function call, there is no default column name—if you want there to be a column name, then you need to supply an alias. Next, you'll notice that we haven't really returned much of anything. So, what does this recordset represent? It is the number of rows that matched the WHERE condition in the query for the table(s) in the FROM clause.

Keep this query in mind. This is a basic query that you can use to verify that the exact number of rows that you expect to be in a table and match your WHERE condition are indeed in there.

Just for fun, try running the query without the WHERE clause:

```
SELECT COUNT(*)  
FROM HumanResources.Employee
```

If you haven't done any deletions or insertions into the Employees table, then you should get a recordset that looks something like this:

```
-----  
290  
(1 row(s) affected)
```

What is that number? It's the total number of rows in the Employee table. This is another one to keep in mind for future use.

Now it's time to look at it with an expression—usually a column name. First, try running the COUNT the old way, but against a new table:

```
SELECT COUNT(*)  
FROM HumanResources.JobCandidate;
```

This is a smaller table, so you get a lower count:

```
-----  
13  
(1 row(s) affected)
```

Now alter your query to select the count for a specific column:

```
SELECT COUNT(EmployeeID)  
FROM HumanResources.JobCandidate;
```

Chapter 3

You'll get a result that is a bit different from the one before:

```
-----  
2
```

```
(1 row(s) affected)
```

```
Warning: Null value is eliminated by an aggregate or other SET operation.
```

Why the difference? The answer is fairly obvious when you stop to think about it — there isn't a value, as such, for the EmployeeID column in every row. In short, the COUNT, when used in any form other than COUNT(*), ignores NULL values.

Actually, all aggregate functions ignore NULLs except for COUNT (*). Think about this for a minute — it can have a very significant impact on your results. Many users expect NULL values in numeric fields to be treated as zero when performing averages, but a NULL does not equal zero, and as such, shouldn't be used as one. If you perform an AVG or other aggregate function on a column with NULLs, the NULL values will not be part of the aggregation unless you manipulate them into a non-NULL value inside the function (using COALESCE() or ISNULL() for example). We'll explore this further in Chapter 6, but beware of this when coding in T-SQL and when designing your database.

Why does it matter in your database design? Well, it can have a bearing on whether you decide to allow NULL values in a field or not when you think about the way that queries are likely to be run against the database and how you want your aggregates to work.

Now that we've seen how to operate with groups, let's move on to one of the concepts that a lot of people have problems with. Of course, after reading the next section, you'll think it's a snap.

Placing Conditions on Groups with the HAVING Clause

The HAVING clause is used only if there is also a GROUP BY in your query; whereas the WHERE clause is applied to each row before it even has a chance to become part of a group, the HAVING clause is applied to the aggregated value for that group.

To demonstrate this, let's run a quick example using the same query we used in our final GROUP BY clause example, but adding a HAVING clause.

```
SELECT SalesOrderID, SUM(OrderQty)  
FROM Sales.SalesOrderDetail  
WHERE SalesOrderID BETWEEN 43684 AND 43686  
GROUP BY SalesOrderID  
HAVING SUM(OrderQty) > 5;
```

Remember that we had three rows returned from the original query, but this time the HAVING clause reduces us to just two (the row with a total OrderQty of 5 has been filtered out):

```
SalesOrderID
-----
43684      9
43685      6

(2 row(s) affected)
```

Outputting XML Using the FOR XML Clause

Back in 2000, when the previous edition of SQL Server came out, XML was new on the scene but quickly proving itself as a key way of making data available. As part of that version of SQL Server—SQL Server 2000—Microsoft added the ability to have your results sent back in an XML format rather than the traditional result set. This was pretty powerful—particularly in web or cross-platform environments.

Microsoft has since reworked the way it delivers XML output a bit, but the foundations and importance are still the same. I’m going to shy away from the details of this clause for now since XML is a discussion unto itself—but we’ll spend extra time with XML in Chapter 16. So, for now just trust me that it’s better to learn the basics first.

Making Use of Hints Using the OPTION Clause

The OPTION clause is a way of overriding some of SQL Server’s ideas of how to best run your query. Since SQL Server really does usually know what’s best for your query, using the OPTION clause will more often hurt you rather than help you. Still, it’s nice to know that it’s there just in case.

This is another one of those “I’ll get there later” subjects. We talk about query hints extensively when we talk about locking (Chapter 12) later in the book, but, until you understand what you’re affecting with your hints, there is little basis for understanding the OPTION clause—as such, we’ll defer discussion of it for now.

DISTINCT

There’s just one more major concept to get through and we’ll be ready to move from the SELECT statement on to action statements. It has to do with repeated data.

The DISTINCT clause is all about eliminating duplicate data. If a value is the same, it appears only once (and, if used in conjunction with a COUNT(), is counted only once). It is applied either at the beginning of the SELECT list (if you want your result set to not have duplicate rows) or inside the COUNT() statement if using it in conjunction with COUNT().

To demonstrate, we will run four quick queries, and largely let them speak for themselves.

To start, let’s select from the SalesOrderDetail table all the rows from SalesOrderID 43685 through 43687.

Chapter 3

```
SELECT SalesOrderID  
FROM Sales.SalesOrderDetail  
WHERE SalesOrderID BETWEEN 43685 AND 43687;
```

This yields us:

```
SalesOrderID  
-----  
43685  
43685  
43685  
43685  
43686  
43686  
43686  
43687  
43687
```

(9 row(s) affected)

But by adding DISTINCT:

```
SELECT DISTINCT SalesOrderID  
FROM Sales.SalesOrderDetail  
WHERE SalesOrderID BETWEEN 43685 AND 43687;
```

we get something much different:

```
SalesOrderID  
-----  
43685  
43686  
43687
```

(3 row(s) affected)

Now, let's change this around to using counts. Going back to our first query:

```
SELECT COUNT(SalesOrderID)  
FROM Sales.SalesOrderDetail  
WHERE SalesOrderID BETWEEN 43685 AND 43687;
```

gets us back a count of nine:

```
-----  
9
```

(1 row(s) affected)

But . . .

```
SELECT COUNT(DISTINCT SalesOrderID)
FROM Sales.SalesOrderDetail
WHERE SalesOrderID BETWEEN 43685 AND 43687;
```

gets us just three . . .

3

(1 row(s) affected)

Note that you can use DISTINCT with any aggregate function, though I question whether many of the functions have any practical use for it. For example, I can't imagine why you would want an average of just the DISTINCT rows.

Adding Data with the INSERT Statement

The basic syntax for an `INSERT` statement looks like this:

```
INSERT [INTO] <table> [(column_list)] VALUES (data_values)
```

Let's look at the parts:

- ❑ `INSERT` is the action statement. It tells SQL Server what it is that we're going to be doing with this statement, and everything that comes after this keyword is merely spelling out the details of that action.
- ❑ The `INTO` keyword is pretty much just fluff. Its sole purpose in life is to make the overall statement more readable. It is completely optional, but I highly recommend its use for the very reason that they added it to the statement—it makes things much easier to read.
- ❑ Next comes the table into which you are inserting.
- ❑ Until this point, things have been pretty straightforward—now comes the part that's a little more difficult: the column list. An explicit column list (where you specifically state the columns to receive values) is optional, but not supplying one means that you have to be extremely careful. If you don't provide an explicit column list, then each value in your `INSERT` statement will be assumed to match up with a column in the same ordinal position of the table in order (first value to first column, second value to second column, etc.). Additionally, a value must be supplied for every column, in order, until you reach the last column that both does not accept nulls and has no default (you'll see more about what I mean shortly). In summary, this will be a list of one or more columns that you are going to be providing data for in the next part of the statement.
- ❑ Finally, you'll supply the values to be inserted. There are two ways of doing this: explicitly supplied values and values derived from a `SELECT` statement.

To supply the values, we'll start with the `VALUES` keyword and then follow that with a list of values, separated by commas and enclosed in parentheses. The number of items in the value list

Chapter 3

must exactly match the number of columns in the column list. The data type of each value must match or be implicitly convertible to the type of the column with which it corresponds (they are taken in order).

On the issue of “Should I specifically state a value for all columns or not?” I really recommend naming every column every time—even if you just use the DEFAULT keyword or explicitly state NULL. DEFAULT will tell SQL Server to use whatever the default value is for that column (if there isn’t one, you’ll get an error).

What’s nice about this is the readability of code—this way it’s really clear what you are doing. In addition, I find that explicitly addressing every column leads to fewer bugs, and also gives you a better chance of your statement still being compatible even if you make changes to the table structure later.

So, for example, we might have an `INSERT` statement that looks something like:

```
INSERT INTO HumanResources.JobCandidate  
VALUES  
(1, NULL, DEFAULT);
```

As stated earlier, unless we provide a different column list (we’ll cover how to provide a column list shortly), all the values have to be supplied in the same order as the columns are defined in the table. The exception to this is if you have an identity column, in which case that column is assumed to be skipped.

If you check the definition of the `HumanResources.JobCandidate` table, you will indeed see that it starts with an identity column called `JobCandidateID`. Since it is an identity column, we know that the system is going to automatically generate a value for that column, so we can skip it.

Since, in theory (because you’re reading this book instead of the beginning one), we already mostly know this stuff and are just reviewing, I’ve crammed several concepts into this.

- I’ve skipped the identity column entirely (the system will fill that in for us).
- I’ve supplied actual values (the number 1 for `EmployeeID`, and an explicit declaration of `NULL` as a value for `Resume`).
- I’ve used the `DEFAULT` keyword (for `ModifiedDate`) to explicitly tell the server to use whatever the default value is for that column.

Now let’s try it again with modifications for inserting into specific columns:

```
INSERT INTO HumanResources.JobCandidate  
(EmployeeID, Resume, ModifiedDate)  
VALUES  
(1, NULL, DEFAULT);
```

Note that we are still skipping the identity column. Other than that, we are just explicitly supplying names—nothing else has changed.

The **INSERT INTO . . . SELECT Statement**

Well, this “one row at a time” business is all fine and dandy, but what if we have a block of data that we want inserted? Over the course of the book, we’ll look at lots of different scenarios where that can happen, but for now, we’re going to focus on the case where what we want to insert into our table can be obtained by selecting it from another source such as:

- Another table in our database
- A totally different database on the same server
- A heterogeneous query from another SQL Server or other data
- The same table (usually, you’re doing some sort of math or other adjustment in your `SELECT` statement in this case)

The `INSERT INTO . . . SELECT` statement can do all of these. The syntax for this statement comes from a combination of the two statements we’ve seen thus far—the `INSERT` statement and the `SELECT` statement. It looks something like this:

```
INSERT INTO <table name>
[<column list>]
<SELECT statement>
```

The result set created from the `SELECT` statement becomes the data that is added in your `INSERT` statement.

Let’s check this out by doing something that, if you get into advanced coding, you’ll find yourself doing all too often—selecting some data into some form of temporary table. In this case, we’re going to declare a variable of type table and fill it with rows of data from our `Orders` table:

This next block of code is what is called a script. This particular script is made up of one batch. We will be examining batches at length in Chapter 10.

```
/*
 * This next statement is going to use code to change the "current" database
 ** to AdventureWorks. This makes certain, right in the code, that we are going
 ** to the correct database.
 */

USE AdventureWorks;

/*
 * This next statement declares our working table.
 ** This particular table is table variable we are creating on the fly.
 */

DECLARE @MyTable Table
(
    SalesOrderID      int,
    CustomerID       int
);
```

Chapter 3

```
/* Now that we have our table variable, we're ready to populate it with data
** from our SELECT statement. Note that we could just as easily insert the
** data into a permanent table (instead of a table variable).
*/
INSERT INTO @MyTable
    SELECT SalesOrderID, CustomerID
    FROM AdventureWorks.Sales.SalesOrderHeader
    WHERE SalesOrderID BETWEEN 50222 AND 50225;

-- Finally, let's make sure that the data was inserted like we think
SELECT *
FROM @MyTable;
```

This should yield you results that look like this:

```
(4 row(s) affected)
SalesOrderID CustomerID
-----
50222      638
50223      677
50224      247
50225      175

(4 row(s) affected)
```

The first (4 row[s] affected) we see is the effect of the `INSERT...SELECT` statement at work — our `SELECT` statement returns four rows, so that's what got inserted into our table. We then use a straight `SELECT` statement to verify the insert.

Note that if you try running a `SELECT` against `@MyTable` by itself (that is, outside this script), you're going to get an error. `@MyTable` is a variable that we have declared, and it exists only as long as our batch is running. After that, it is automatically destroyed.

It's also worth noting that we could have used what's called a "temporary table" — this is similar in nature, but doesn't work in quite the same way. We will revisit temp tables and table variables in Chapters 10 and 11.

Changing What You've Got with the UPDATE Statement

The `UPDATE` statement, like most SQL statements, does pretty much what it sounds like it does — it updates existing data. The structure is a little bit different from a `SELECT`, though you'll notice definite similarities. Let's look at the syntax:

```
UPDATE <table name>
SET <column> = <value> [,<column> = <value>]
[FROM <source table(s)>]
[WHERE <restrictive condition>]
```

An UPDATE can be created from multiple tables, but can affect only one table. What do I mean by that? Well, we can build a condition, or retrieve values from any number of different tables (via a join), but only one table at a time can be the subject of the update action. For now, we'll look at a simple update.

Let's start off by looking at our old friend Jo Brown (you may remember her from our look at joins earlier in the chapter). It seems that Jo has recently gotten married, and we need to make sure that her data is accurate. Let's run a query to look at one row of data:

```
SELECT e.EmployeeID,
       e.MaritalStatus,
       ce.FirstName,
       ce.LastName
  FROM HumanResources.Employee e
 JOIN Person.Contact ce
    ON e.ContactID = ce.ContactID
 WHERE ce.FirstName = 'Jo'
   AND ce.LastName = 'Brown';
```

Which returns the following:

EmployeeID	MaritalStatus	FirstName	LastName
16	S	Jo	Brown

(1 row(s) affected)

Let's update the MaritalStatus value to the more proper "M":

```
UPDATE e
SET MaritalStatus = 'M'
  FROM HumanResources.Employee e
 JOIN Person.Contact ce
    ON e.ContactID = ce.ContactID
 WHERE ce.FirstName = 'Jo'
   AND ce.LastName = 'Brown';
```

Much like when we ran the INSERT statement, we don't get much back from SQL Server:

(1 row(s) affected)

Yet, when we again run our SELECT statement, we see that the value has indeed changed:

EmployeeID	MaritalStatus	FirstName	LastName
16	S	Jo	Brown

(1 row(s) affected)

Chapter 3

Note that we could have changed more than one column just by adding a comma and the additional column expression. For example, the following statement would have also given Jo a promotion:

```
UPDATE e
SET MaritalStatus = 'M', Title = 'Shift Manager'
FROM HumanResources.Employee e
JOIN Person.Contact ce
    ON e.ContactID = ce.ContactID
WHERE ce.FirstName = 'Jo'
    AND ce.LastName = 'Brown';
```

If we choose, we can use an expression for the SET clause instead of the explicit values we've used thus far. For example, if we wanted to increase Jo's vacation time by 20 percent (after all, she's got a nice promotion!), we could do something like:

```
UPDATE e
SET VacationHours = VacationHours * 1.2
FROM HumanResources.Employee e
JOIN Person.Contact ce
    ON e.ContactID = ce.ContactID
WHERE ce.FirstName = 'Jo'
    AND ce.LastName = 'Brown';
```

As you can see, a single UPDATE statement can be fairly powerful. Even so, this is really just the beginning. We'll see even more advanced updates in later chapters.

While SQL Server is nice enough to let us update pretty much any column (there are a few that we can't, such as timestamps), be very careful about updating primary keys. Doing so puts you at very high risk of "orphaning" other data (data that has a reference to the data you're changing).

The DELETE Statement

The version of the DELETE statement that we'll cover in this chapter may be one of the easiest statements of them all. There's no column list—just a table name and, usually, a WHERE clause. The syntax couldn't be much easier:

```
DELETE [TOP (<expression>) [PERCENT]
[FROM ] <table_name>
[FROM ] <table list/JOIN conditions>
[WHERE <search condition>]
```

The tricky thing on this is the *two* FROM clauses (nope, that is not a typo). You can think of this as being somewhat like the UPDATE statement in the sense that you need to say what table you actually want to delete from, and the second is a more genuine FROM clause including JOINS if you so choose (ostensibly to help you to determine what rows you want to delete).

The WHERE clause works just like all of the WHERE clauses we've seen thus far. We don't need to provide a column list because we are deleting the entire row (you can't delete half a row for example).

Since the first form of DELETE is so easy, we'll perform only a quick example. Let's utilize the sample we built for joins early in the chapter. In case you skipped that, here is the code that built those tables.

If you built the FULL JOIN sample from earlier in the chapter, you can just ignore this build script, as they are identical.

```
CREATE TABLE Film
(FilmID      int          PRIMARY KEY,
 FilmName    varchar(20) NOT NULL,
 YearMade   smallint     NOT NULL
);

CREATE TABLE Actors
(FilmID      int          NOT NULL,
 FirstName   varchar(15) NOT NULL,
 LastName    varchar(15) NOT NULL,
 CONSTRAINT PKActors PRIMARY KEY(FilmID, FirstName, LastName)
);

INSERT INTO Film
VALUES
(1, 'My Fair Lady', 1964);
INSERT INTO Film
VALUES
(2, 'Unforgiven', 1992);

INSERT INTO Actors
VALUES
(1, 'Rex', 'Harrison');
INSERT INTO Actors
VALUES
(1, 'Audrey', 'Hepburn');
INSERT INTO Actors
VALUES
(3, 'Anthony', 'Hopkins');
```

Let's start by selecting out the rows from Film just to show what we have:

```
SELECT * FROM Film;
```

This should get us back just the two rows we inserted in our build script:

FilmID	FilmName	YearMade
1	My Fair Lady	1964
2	Unforgiven	1992

(2 row(s) affected)

Chapter 3

What we're going to do is delete the title *Unforgiven* from our table.

```
DELETE Film  
WHERE FilmName = 'Unforgiven';
```

Now re-execute our SELECT:

FilmID	FilmName	YearMade
1	My Fair Lady	1964

(1 row(s) affected)

Poof! Our record was indeed deleted.

Now let's move on to the slightly more complex JOIN scenario. This time we want to delete all rows from the `Actors` table where there are no matching rows in the `Film` table. This requires a query that is aware of both tables (so a JOIN is required). In addition, however, it requires realizing that there is no match on one side of the join (that `Film` does not have a record that matches a particular actor).

You may recall that an OUTER join will return a NULL on the side where there is no match. We are going to utilize that here by actually testing for NULL.

This is fast-moving stuff, so take it slow and review if you did not already have some fundamentals before reading this section. This book assumes more knowledge and experience in its readers than the previous edition did, so it moves very fast. Take your time, and walk through the examples carefully.

```
DELETE FROM Actors  
FROM Actors a  
LEFT JOIN Film f  
    ON a.FilmID = f.FilmID  
WHERE f.FilmID IS NULL;
```

So, if we skip ahead for a moment to our second `FROM` clause, you can see that we are utilizing a LEFT JOIN. That means all actors will be returned. Films will be returned if there is a matching `FilmID`, but the film side of the columns will be NULL if no match exists. In the `DELETE` statement in the example, we are leveraging this knowledge, and testing for it—if we find a `FilmID` that is null, then we must not have found a match there (and, therefore, our actor needs to be deleted).

Exploring Alternative Syntax for Joins

What we're going to look at in this section is what many people still consider to be the “normal” way of coding joins. Until SQL Server 6.5, the alternative syntax we'll look at here was the only join syntax in SQL Server, and what is today called the “standard” way of coding joins wasn't even an option.

Until now, we have been using the ANSI syntax for all of our SQL statements. I highly recommend that you use the ANSI method since it has much better portability between systems and is also much more readable. It is worth noting that the old syntax is actually reasonably well supported across platforms at the current time, but the ANSI syntax is now also supported by every major platform out there.

The primary reason I am covering the old syntax at all is that there is absolutely no doubt that, sooner or later, you will run into it in legacy code. I don't want you staring at that code saying, "What the heck is this?" That being said, I want to reiterate my strong recommendation that you use the ANSI syntax wherever possible. Again, it is substantially more readable, and Microsoft has indicated that it may not continue to support the old syntax indefinitely. I find it very hard to believe, given the amount of legacy code out there, that Microsoft will dump the old syntax any time soon, but you never know.

Perhaps the biggest reason is that the ANSI syntax is actually more functional. Under old syntax, it was actually possible to create ambiguous query logic—where there was more than one way to interpret the query. The new syntax eliminates this problem.

Remember when I compared a JOIN to a WHERE clause earlier in this chapter? Well, there was a reason. The old syntax expresses all of the JOINS within the WHERE clause.

The old syntax supports all of the joins that we've done using ANSI with the exception of a FULL JOIN. If you need to perform a FULL JOIN, I'm afraid you'll have to stick with the ANSI version.

An Alternative INNER JOIN

Let's do a déjà vu thing and look back at the first INNER JOIN we did in this chapter:

```
SELECT *
FROM HumanResources.Employee e
INNER JOIN HumanResources.Employee m
    ON e.ManagerID = m.EmployeeID;
```

Instead of using the ANSI JOIN, however, let's rewrite it using a WHERE clause-based join syntax. It's actually quite easy—just eliminate the words INNER JOIN and add a comma, and replace the ON operator with a WHERE clause:

```
SELECT *
FROM HumanResources.Employee e, HumanResources.Employee m
WHERE e.ManagerID = m.EmployeeID;
```

It's a piece of cake, and it yields us the same rows we got with the other syntax.

This syntax is supported by virtually all major SQL systems (Oracle, DB2, MySQL, etc.) in the world today.

An Alternative OUTER JOIN

With SQL Server 2005, we do not necessarily have the alternative OUTER JOIN syntax available to us. Indeed, by default it is now turned off—you must set your database compatibility level to be 80 or lower (80 is SQL Server 2000). Thankfully, there is very little code left out there using this syntax.

Since it is being actively deprecated, I'm not going to dwell on this syntax much, but the basics work pretty much the same as the INNER JOIN, except that, because we don't have the LEFT or RIGHT keywords (and no OUTER or JOIN for that matter), we need some special operators especially built for the task. These look like this:

Chapter 3

Alternative	ANSI
* =	LEFT JOIN
= *	RIGHT JOIN

Let's pull up the first OUTER JOIN we did this chapter. It made use of the pubs database and looked something like this:

```
SELECT e.EmployeeID, m.EmployeeID AS ManagerID  
FROM HumanResources.Employee e  
LEFT OUTER JOIN HumanResources.Employee m  
    ON e.ManagerID = m.EmployeeID;
```

Again, we just lose the words LEFT OUTER JOIN, and replace the ON operator with a WHERE clause:

```
SELECT e.EmployeeID, m.EmployeeID AS ManagerID  
FROM HumanResources.Employee e, HumanResources.Employee m  
WHERE e.ManagerID *= m.EmployeeID;
```

The alternative syntax for outer joins is not available by default. You must be running 80 or lower as your compatibility mode in order to use this functionality.

An Alternative CROSS JOIN

This is far and away the easiest of the bunch. To create a CROSS JOIN using the old syntax, you just do nothing. That is, you don't put anything in the WHERE clause of the form: TableA.ColumnA = TableB.ColumnA.

So, for an ultra-quick example, let's take our first example from the CROSS JOIN section earlier in the chapter. The ANSI syntax looked like this:

```
SELECT *  
FROM Film f  
CROSS JOIN Actors a;
```

To convert it to the old syntax, we just strip out the CROSS JOIN keywords and add a comma:

```
SELECT *  
FROM Film f, Actors a;
```

As with the other examples in this section, we get back the same results that we got with the ANSI syntax.

Now we're back to being supported across most of the database management systems.

The UNION

Okay, enough with all the “old syntax” versus “new syntax” stuff—now we’re into something that’s the same regardless of what other join syntax you prefer—the UNION operator. UNION is a special operator we can use to cause two or more queries to generate one result set.

A UNION isn’t really a JOIN, like the previous options we’ve been looking at—instead it’s more of an appending of the data from one query right onto the end of another query (functionally, it works a little differently from this, but this is the easiest way to look at the concept). Where a JOIN combined information horizontally (adding more columns), a UNION combines data vertically (adding more rows), as illustrated in Figure 3-2.

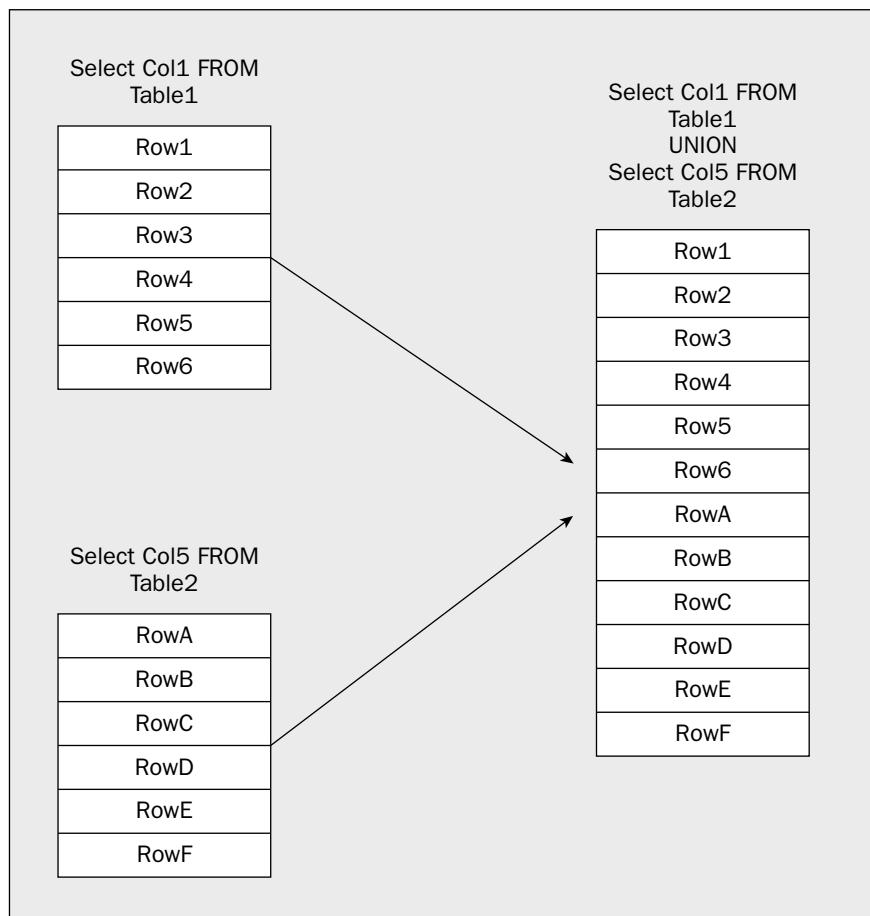


Figure 3-2

Chapter 3

When dealing with queries that use a UNION, there are just a few key points:

- ❑ All the UNIONED queries must have the same number of columns in the SELECT list. If your first query has three columns in the SELECT list, then the second (and any subsequent queries being UNIONED) must also have three columns. If the first has five, then the second must have five, too. Regardless of how many columns are in the first query, there must be the same number in the subsequent query(ies).
- ❑ The headings returned for the combined result set will be taken only from the first of the queries. If your first query has a SELECT list that looks like `SELECT Col1, Col2 AS Second, Col3 FROM...`, then regardless of how your columns are named or aliased in the subsequent queries, the headings on the columns returned from the UNION will be `Col1, Second, and Col3`, respectively.
- ❑ The datatypes of each column in a query must be implicitly compatible with the datatype in the same relative column in the other queries. Note that I'm *not* saying they have to be the same datatype—they just have to be implicitly convertible (a conversion table that shows implicit versus explicit conversions can be found in Chapter 1). If the second column in the first query is of type `char(20)`, then it would be fine that the second column in the second query is `varchar(50)`. However, because things are based on the first query, any rows longer than 20 would be truncated for data from the second result set.
- ❑ Unlike non-UNION queries, the default return option for UNIONS is `DISTINCT` rather than `ALL`. This can really be confusing to people. In our other queries, all rows were returned regardless of whether they were duplicated with another row or not, but the results of a UNION do not work that way. Unless you use the `ALL` keyword in your query, only one of any repeating rows will be returned.

As always, let's take a better look at this with an example.

In this case, we are creating two tables from which we will select. We'll then insert three rows into each table, with one row being identical between the two tables. If our query is performing an `ALL`, then every row (six of them) will show up. If the query is performing a `DISTINCT`, then it will return only five rows (tossing out one duplicate):

```
CREATE TABLE UnionTest1
(
    idcol    int         IDENTITY,
    col2     char(3),
);

CREATE TABLE UnionTest2
(
    idcol    int         IDENTITY,
    col4     char(3),
);

INSERT INTO UnionTest1
VALUES
    ('AAA');
INSERT INTO UnionTest1
VALUES
    ('BBB');
```

```
INSERT INTO UnionTest1
VALUES
    ('CCC');

SELECT *
FROM UnionTest1;

INSERT INTO UnionTest2
VALUES
    ('CCC');
INSERT INTO UnionTest2
VALUES
    ('DDD');
INSERT INTO UnionTest2
VALUES
    ('EEE');

PRINT 'Regular UNION-----'
SELECT col2
FROM UnionTest1
UNION
SELECT col4
FROM UnionTest2;

PRINT 'UNION ALL-----'

SELECT col2
FROM UnionTest1
UNION ALL
SELECT col4
FROM UnionTest2;

DROP TABLE UnionTest1;
DROP TABLE UnionTest2;
```

Now, let's look at the heart of what's returned (you'll see some "one row(s) affected" comments in there—just ignore them until you get to where the results of your query are visible):

```
Regular UNION-----
col2
-----
AAA
BBB
CCC
DDD
EEE

(5 row(s) affected)

UNION ALL-----
col2
-----
AAA
BBB
```

Chapter 3

```
CCC  
CCC  
DDD  
EEE  
  
(6 row(s) affected)
```

The first result set returned was a simple UNION statement with no additional parameters. You can see that one row was eliminated — even though we inserted “CCC” into both tables, only one makes an appearance since the duplicate record is eliminated by default. Other than that, what we have is the results of the UNION’d SELECT statements combined into one result set.

The second return changed things only slightly — this time we used a UNION ALL and the ALL keyword ensured that we get every row back. Thus, our eliminated row from the last query suddenly reappears.

Summary

T-SQL is SQL Server’s own brand of ANSI SQL or Structured Query Language. T-SQL is entry-level ANSI 92 compliant, but it also has a number of its own extensions to the language — we’ll see more of those in later chapters.

Even though, for backward compatibility, SQL Server has a number of different syntax choices that are effectively the same, wherever possible, you ought to use the ANSI form. Where there are different choices available, I will usually show you all of the choices, but again, stick with the ANSI version wherever possible. This is particularly important for situations where you think your back end — or database server — might change at some point. Your ANSI code will, more than likely, run on the new database server — however, code that is only T-SQL definitely will not. The alternative INNER join syntax will work on most SQL products today, but the OUTER join syntax will not. FULL joins are often not supported in other DBMSs at all.

In our next chapter, we’ll be reviewing how to CREATE and ALTER the tables that we want to select from. In this area, we’ll see there are far fewer “nonstandard” choices.

4

Creating and Altering Tables

In this chapter, we will be studying the syntax to create our own tables. We will also take a look at how to make use of the SQL Server Management Studio to help with this (after we know how to do it for ourselves).

However, before we get too deep in the actual statements that create tables and other objects, we need to digress far enough to deal with the convention for a fully qualified object name, and, to a lesser extent, object ownership.

Object Names in SQL Server

In all the queries that you've been performing so far in this book, you've seen simple naming at work. I've had you switch the active database in the Query Analyzer before running any queries, and that has helped your queries to work. How? Well, SQL Server looks only at a very narrow scope when trying to identify and locate the objects you name in your queries and other statements. For example, you've only been providing the names of tables without any additional information, but there are actually four levels in the naming convention for any SQL Server table (and any other SQL Server object for that matter). A fully qualified name is as follows:

[ServerName. [DatabaseName. [SchemaName.]] ObjectName

You must provide an object name whenever you are performing an operation on that object, but all parts of the name to the left of the object name are optional. Indeed, most of the time, they are not needed, and are therefore left off. Still, before you start creating objects, it's a good idea for you to get a solid handle on each part of the name. So, now move from the object name left.

Schema Name (a.k.a. Ownership)

If you're utilizing schemas (most older databases do not, but it appears that it will become more important in the future), you may need to indicate what schema your object is in. It is entirely possible to have objects with the same name, but residing in different schemas. If you're wanting to access an object that is not in your default schema (set on a login by login basis), then you'll need to specifically state the schema name of your object. For example, look at what has to be one of the worst uses of schemas I've ever seen—the AdventureWorks database—and take a look at a query get a list of employees and what city they live in:

```
SELECT e.EmployeeID, c.FirstName, c.LastName, City
FROM HumanResources.Employee AS e
JOIN Person.Contact c
    ON e.ContactID = c.ContactID
JOIN HumanResources.EmployeeAddress AS ea
    ON e.EmployeeID = ea.EmployeeID
JOIN Person.Address AS a
    ON ea.AddressID = a.AddressID
```

In this example, we're making use of four tables spread across two schemas. If one of the two schemas involved—`HumanResources` and `Person`—happened to be our default schema, then we could have left that schema name off when naming tables in that schema. In this case, we named all schema to be on the safe side.

This is another time when I have to get on the consistency soap box. If you're going to use the schema features at all, then I highly recommend using two-part naming (schema and table name) in *all* of your queries. It is far too easy for a user's default schema to get changed or for some other alias to have a change made such that your assumptions about the default are no longer valid. If you're not utilizing different schemas at all in your database design, then it's fine to leave them off (and make your code a fair amount more readable in the process), but keep in mind there may be a price to be paid if you later start using schemas.

A Little More about Schemas

The ANSI standard for SQL has had the notion of what has been called a "schema" for quite some time now. SQL Server has had that same concept in place all along but used to refer to it differently (and, indeed, have a different intent for it even if it could be used the same way). So, what you see referred to in SQL Server 2005 and other databases like Oracle as "schema" was usually referred to as "Owner" in previous versions of SQL Server.

The notion of the schema used to be a sticky one. While it is still non-trivial, Microsoft has added some new twists in SQL Server 2005 to make the problems of schema much easier to deal with. If, however, you need to deal with backward compatibility to prior versions of SQL Server, you're going to need to either avoid the new features or use pretty much every trick they have to offer—and that means ownership (as it was known in prior versions) remains a significant hassle.

There were always some people who liked using ownership in their pre-SQL Server 2005 designs, but I was definitely not one of them. For now, the main thing to know is that ownership has gone through a name change in SQL Server 2005 and is now referred to by the more ANSI-compliant term *schema*. This is somewhat important as Microsoft appears to not only be making a paradigm shift in their support of schemas but also heading in a direction where they will become rather important to your design. New functions exist to support the use of schemas in your naming, and even the new sample that ships with

SQL Server 2005 (the AdventureWorks database that we have occasionally used and will use more often in coming chapters) makes extensive use of schemas. The schema also becomes important in dealing with some other facets of SQL Server such as Notification Services.

Still, I want to focus, for now, on what a schema is and how it works.

For prior releases, ownership (as it was known then) was actually a great deal like what it sounds like—it was recognition, right within the fully qualified name, of who “owned” the object. Usually, this was either the person who created the object or the database owner (more commonly referred to as the `dbo`—I’ll get to describing the `dbo` shortly). For SQL Server 2005, things work in a similar fashion, but the object is assigned to a schema rather than an owner. Whereas an owner related to one particular login, a schema can now be shared across multiple logins, and one login can have rights to multiple schemas.

By default, only users who are members of the `sysadmin` system role, or the `db_owner` or `db_ddladmin` database roles, can create objects in a database.

The roles mentioned here are just a few of many system and database roles that are available in SQL Server 2005. Roles first appeared in SQL Server 7.0 and have a logical set of permissions granted to them according to how that role might be used. When you assign a particular role to someone, you are giving that person the ability to have all the permissions that the role has.

Individual users can also be given the right to create certain types of both database and system objects. If such individuals do indeed create an object, then, by default, that object will be assigned to whatever schema is listed as default for that login.

We'll talk much more about this in Chapter 22, but let me say that just because a feature is there doesn't mean it should be used! Giving `CREATE` authority to individual users is nothing short of nightmarish. Trying to keep track of who created what, when, and for what reason becomes near impossible. In short, keep `CREATE` access limited to the `sa` account or members of the `sysadmins` or `db_owner` security roles.

The Default Schema: `dbo`

Whoever creates the database is considered to be the *database owner*, or `dbo`. Any objects that they create within that database shall be listed with a schema of `dbo` rather than their individual username.

For example, say that I am an everyday user of a database, my login name is `MySchema`, and I have been granted `CREATE TABLE` authority to a given database. If I create a table called `MyTable`, the owner-qualified object name would be `MySchema.MyTable`. Note that, since the table has a specific owner, any user other than me (remember, I'm `MySchema` here) of `MySchema.MyTable` would need to provide the owner qualified name in order for SQL Server to resolve the table name.

Now, say that there is also a user with a login name of `Fred`. `Fred` is the database owner (as opposed to just any member of `db_owner`). If `Fred` creates a table called `MyTable` using a `CREATE` statement identical to that used by `MySchema`, the owner-qualified table name will be `dbo.MyTable`. In addition, as `dbo` also happens to be the default owner, any user could just refer to the table as `MyTable`.

It's worth pointing out that `sa` (or members of the `sysadmin` role) always aliases to the `dbo`. That is, no matter who actually owns the database, `sa` will always have full access as if it were the `dbo`, and any objects created by the `sa` login will show ownership belonging to the `dbo`. In contrast, objects created by members of the `db_owner` database role do *not* default to `dbo` as the default schema—they will be assigned to whatever that particular user has set as their default schema (it could be anything). Weird but true!

The Database Name

The next item in the fully qualified naming convention is the database name. Sometimes you want to retrieve data from a database other than the default, or current, database. Indeed, you may actually want to `JOIN` data from across databases. A database-qualified name gives you that ability. For example, if you were logged in with `AdventureWorks` as your current database, and you wanted to refer to the `Orders` table in a database called `Northwind`, then you could refer to it by `Northwind.dbo.Orders`. Since `dbo` is the default schema, you could also use `Northwind..Orders`. If a schema named `MySchema` owns a table named `MyTable` in `MyDatabase`, then you could refer to that table as `MyDatabase.MySchema.MyTable`. Remember that the current database (as determined by the `USE` command or in the dropdown box if you're using the SQL Server Management Console) is always the default, so, if you only want data from the current database, then you do not need to include the database name in your fully qualified name.

dbo is something of the default default (yes, I said that twice). That is, every user has a default schema in which SQL Server will look for objects if a schema name is not explicitly stated. Prior to SQL Server 2005, that default was the user's login name, but, if it wasn't found there, it also looked under `dbo`. In SQL Server 2005, the default schema for a user starts off as `dbo` but can be changed to some other value. Be careful about using leveraging the default as was done in the previous `Northwind..Orders` example—you never know when a configuration change is going to change the default and potentially break your query.

Naming by Server

In addition to naming other databases on the server you're connected to, you can also "link" to another server. Linked servers give you the capability to perform a `JOIN` across multiple servers—even different types of servers (SQL Server, Oracle, DB2, Access—just about anything with an OLE DB provider). Linked servers are largely an administration-related topic—just realize that there is one more level in your naming hierarchy, that it lets you access different servers, and that it works pretty much like the database and ownership levels work.

Now, just add to our previous example. If we wanted to retrieve information from a server we have created a link with called `MyServer`, a database called `MyDatabase`, and a table called `MyTable` owned by `MySchema`, then the fully qualified name would be `MyServer.MyDatabase.MySchema.MyTable`.

The CREATE Statement

In the Bible, God said, “Let there be light!” And there was light! Unfortunately, creating things isn’t quite as simple for us mere mortals. We need to provide a well-defined syntax in order to create the objects in our database. To do that, we make use of the `CREATE` statement.

Take a look at the full structure of a `CREATE` statement, starting with the utmost in generality. You’ll find that all the `CREATE` statements start out the same and then get into the specifics. The first part of the `CREATE` will always look like:

```
CREATE <object type> <object name>
```

This will be followed by the details that will vary by the nature of the object that you’re creating.

CREATE DATABASE

The most basic syntax for the `CREATE DATABASE` statement looks like the previous example:

```
CREATE DATABASE <database name>
```

It's worth pointing out that when you create a new object, no one can access it except for the person who created it, the system administrator and the database owner (which, if the object created was a database, is the same as the person who created it). This allows you to create things and make whatever adjustments you need to make before you explicitly allow access to your object. We will look further into how to “grant” access and security in general in Chapter 22.

It's also worth noting that you can only use the `CREATE` statement to create objects on the local server (adding in a specific server name doesn't work).

This will yield a database that looks exactly like your `model` database (you discussed the `model` database in Chapter 1). The reality of what you want is almost always different, so look at a more full syntax listing:

```
CREATE DATABASE <database name>
[ON [PRIMARY]
  ([NAME = <'logical file name'>, ]
   FILENAME = <'file name'>
   [, SIZE = <size in kilobytes, megabytes, gigabytes, or terrabytes>]
   [, MAXSIZE = size in kilobytes, megabytes, gigabytes, or terrabytes>]
   [, FILEGROWTH = <kilobytes, megabytes, gigabytes, or terrabytes/percentage>])
  [LOG ON
    ([NAME = <'logical file name'>, ]
     FILENAME = <'file name'>
     [, SIZE = <size in kilobytes, megabytes, gigabytes, or terrabytes>]
     [, MAXSIZE = size in kilobytes, megabytes, gigabytes, or terrabytes>]
     [, FILEGROWTH = <kilobytes, megabytes, gigabytes, or terrabytes/percentage>])
  [ COLLATE <collation name> ]
  [ FOR ATTACH [WITH <service broker>] | FOR ATTACH_REBUILD_LOG | WITH DB_CHAINING {
    ON|OFF } | TRUSTWORTHY { ON|OFF } ]
  [AS SNAPSHOT OF <source database name>]
  [, ]]
```

Chapter 4

Keep in mind that some of the preceding options are mutually exclusive (for example, if you’re creating for attaching, most of the options other than file locations are invalid). There’s a lot there, so take a look at a breakdown of the parts.

ON

ON is used in two places: to define the location of the file where the data is stored and to define the same information for where the log is stored. You’ll notice the PRIMARY keyword there — this means that what follows is the primary (or main) filegroup in which to physically store the data. You can also store data in what are called secondary filegroups — the use of which is outside the scope of this title. For now, stick with the default notion that you want everything in one file.

SQL Server allows you to store your database in multiple files; furthermore, it allows you to collect those files into logical groupings called filegroups. Filegroups are an unnecessary complication for *most* databases, but for Very Large Databases (VLDBs, as they are known), filegroups can offer a way to split up the backup and restore operations or just split I/O — very handy.

NAME

This one isn’t quite what it sounds like. It is a name for the file you are defining, but only a logical name — that is, the name that SQL Server will use internally to refer to that file. You will use this name when you want to resize (expand or shrink) the database and/or file.

FILENAME

This one *is* what it sounds like — the physical name on the disk of the actual operating system file in which the data and log (depending on what section you’re defining) will be stored. The default here (assuming you used the simple syntax we looked at first) depends on whether you are dealing with the database itself or the log. Your file will be located in the `MSSQL.1\MSQL\DATA` subdirectory under your main `Program Files\Microsoft SQL Server` directory (or whatever you called your main SQL Server directory if you changed it at install). If you’re dealing with the physical database file, it will be named the same as your database with an `.mdf` extension. If you’re dealing with the log, it will be named the same as the database file only with a suffix of `_Log` and an `.ldf` extension. You are allowed to specify other extensions if you explicitly name the files, but I strongly encourage you to stick with the defaults of `mdf` (database) and `ldf` (log file). As a sidenote, secondary files have a default extension of `.ndf`.

Keep in mind that, while `FILENAME` is an optional parameter, it is optional only as long as you go with the extremely simple syntax (the one that creates a new database based on the `model` database) that I introduced first. If you provide any of the additional information, then you must include an explicit file name — be sure to provide a full path.

SIZE

No mystery here. It is what it says — the size of the database. By default, the size is in megabytes, but you can make it kilobytes by using a `KB` instead of `MB` after the numeric value for the size, or go bigger by using `GB` (gigabytes) or even `TB` (terabytes). Keep in mind that this value must be at least as large as the `model` database is and must be a whole number (no decimals) or you will receive an error. If you do not supply a value for `SIZE`, then the database will initially be the same size as the `model` database.

MAXSIZE

This one is still pretty much what it sounds like, with only a slight twist to the `SIZE` parameter. SQL Server has a mechanism to allow your database to automatically allocate additional disk space (to grow) when necessary. `MAXSIZE` is the maximum size to which the database can grow. Again, the number is, by default, in megabytes, but like `SIZE`, you can use `KB`, `GB`, or `TB` to use different increment amounts. The slight twist is that there is no firm default. If you don't supply a value for this parameter, then there is considered to be no maximum — the practical maximum becomes when your disk drive is full.

If your database reaches the value set in the `MAXSIZE` parameter, your users will start getting errors back saying that their inserts can't be performed. If your log reaches its maximum size, you will not be able to perform any logged activity (which is most activities) in the database. Personally, I recommend setting up what is called an *alert*. You can use alerts to tell you when certain conditions exist (such as a database or log that's almost full). You'll see how to create alerts in Chapter 24.

I recommend that you always include a value for `MAXSIZE`, and that you make it at least several megabytes smaller than would fill up the disk. I suggest this because a completely full disk can cause situations where you can't commit any information to permanent storage. If the log was trying to expand, the results could potentially be disastrous. In addition, even the operating system can occasionally have problems if it runs completely out of disk space.

One more thing — if you decide to follow my advice on this issue, be sure to keep in mind that you may have multiple databases on the same system. If you size each of them to be able to take up the full size of the disk less a few megabytes, then you will still have the possibility of a full disk (if they all expand).

FILEGROWTH

Whereas `SIZE` set the initial size of the database, and `MAXSIZE` determined just how large the database file could get, `FILEGROWTH` essentially determines just how fast it gets to that maximum. You provide a value that indicates by how many bytes (in `KB`/`MB`/`GB` or `TB`) at a time you want the file to be enlarged. Alternatively, you can provide a percentage value by which you want the database file to increase. With this option, the size will go up by the stated percentage of the current database file size. Therefore, if you set a database file to start out at 1GB with a `FILEGROWTH` of 20 percent, then the first time it expands it will grow to 1.2GB, the second time to 1.44GB, and so on.

LOG ON

The `LOG ON` option allows you to establish that you want your log to go to a specific set of files and where exactly those files are to be located. If this option is not provided, then SQL Server will create the log in a single file and default it to a size equal to 25 percent of the data file size. In most other respects, it has the same file specification parameters as the main database file does.

It is highly recommended that you store your log files on a different drive than your main data files. Doing so avoids having the log and main data files competing for I/O off the disk as well as providing additional safety should one hard drive fail.

COLLATE

This one has to do with the issue of sort order, case sensitivity, and sensitivity to accents. When you installed your SQL Server, you decided on a default collation, but you can override this at the database level (and, as you'll see later in the chapter, also at the column level).

FOR ATTACH

You can use this option to attach an existing set of database files to the current server. The files in question must be part of a database that was, at some point, properly detached using `sp_detach_db`. Normally, you would use `sp_attach_db` for this functionality, but the `CREATE DATABASE` command with `FOR ATTACH` has the advantage of being able to access as many as 32,000+ files—`sp_attach_db` is limited to just 16.

If you use `FOR ATTACH`, you must complete the `ON PRIMARY` portion of the file location information. Other parts of the `CREATE DATABASE` parameter list can be left off as long as you are attaching the database to the same file path they were in when they were originally detached.

WITH DB CHAINING ON|OFF

As previously mentioned, the concept of “schemas” didn’t really exist in prior versions of SQL Server. Instead, we had the notion of “ownership.” One of the bad things that could happen with ownership was “ownership chains.” This was a situation where person A was the owner of an object, and then person B became the owner of an object that depended on person A’s object. You could have person after person create objects that depended on other people’s objects, and there became a complex weave of permission issues based on this.

This switch is about respecting such ownership chains when they cross databases (person A’s object is in DB1, and person B’s object is in DB2). Turn it on, and cross-database ownership chains work—turn it off, and they don’t. Avoid such ownership chains as if they were the plague—they are a database equivalent to a plague, believe me!

TRUSTWORTHY

This is set to off by default, and for the vast majority of databases that will be the right option. What does it do? Well, `TRUSTWORTHY` indicates that your database has objects within it (most likely .NET assemblies, which we will discuss in Chapter 14) that need access to objects outside your SQL Server environment (such as the file system), and that you want that access granted subject to an *impersonation context*. We will discuss impersonation contexts somewhat in Chapter 22, but, for now, think of it as a way of changing the security for a given object to let it have special access that goes beyond what the connected user would normally have as long as that access is within the context of that object. (For example, access to a certain network resource).

Building a Database

The following is the statement to create a sample database. The database itself is only one of many objects that we will create on our way to a fully functional database:

```
CREATE DATABASE Accounting
ON
    (NAME = 'Accounting',
     FILENAME = 'c:\Program Files\Microsoft SQL Server\
MSSQL.1\mssql\data\AccountingData.mdf',
```

```
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 5)
LOG ON
(NAME = 'AccountingLog',
FILENAME = 'c:\Program Files\Microsoft SQL Server\
MSSQL.1\mssql\data\AccountingLog.ldf',
SIZE = 5MB,
MAXSIZE = 25MB,
FILEGROWTH = 5MB)

GO
```

Try running a command called `sp_helpdb`. This system stored procedure is especially tailored for database structure information, and often provides better information if you're more interested in the database itself than the objects it contains. `sp_helpdb` takes one parameter — the database name:

```
EXEC sp_helpdb 'Accounting'
```

This actually yields you two separate result sets. The first is based on the combined (data and log) information about your database:

name	db_size	owner	dbid	created	status	compatibility_level
Accounting	15.00 MB	sa	9	May 28 2005	Status=ONLINE, Updateability= READ_WRITE, UserAccess= MULTI_USER, Recovery=FULL, Version=598, Collation=SQL_ Latin1_General_ CP1_CI_AS, SQLSortOrder=52, IsAutoCreate Statistics, IsAuto UpdateStatistics, IsFullTextEnabled	90

The actual values you receive for each of these fields may vary somewhat from mine. For example, the DBID value will vary depending on how many databases you've created and in what order you've created them. The various status messages will vary depending on what server options were in place at the time you created the database as well as any options you changed for the database along the way.

Note that the `db_size` property is the *total* of the size of the database and the size of the log.

The second provides specifics about the various files that make up your database — including their current size and growth settings:

Chapter 4

name	fileid	filename	filegroup	size	maxsize	growth	usage
Accounting	1	C:\Program Files\Microsoft SQL Server\mssql\data\Accounting Data.mdf	PRIMARY	10240 KB	51200 KB	5120 KB	data only
AccountingLog	2	C:\Program Files\Microsoft SQL Server\mssql\data\Accounting Log.ldf	NULL	5120 KB	25600 KB	5120 KB	log only

After you create tables and insert data, the database will begin to automatically grow on an as-needed basis.

CREATE TABLE

The first part of creating a table is pretty much the same as creating any object — remember that line I showed you? Well, here it is again:

```
CREATE <object type> <object name>
```

Since a table is what we want, we can be more specific:

```
CREATE TABLE Customers
```

With `CREATE DATABASE`, we could have stopped with just these first three keywords, and it would have built the database based on the guidelines established in the `model` database. With tables, however, there is no `model`, so we need to provide some more specifics in the form of columns, data types, and special operators.

Let's look at more extended syntax:

```
CREATE TABLE [database_name.] [owner]. [table_name]
(<column name> <data type>
 [[DEFAULT <constant expression>]
 | [IDENTITY [(seed, increment) [NOT FOR REPLICATION]]]
 | [ROWGUIDCOL]
 | [COLLATE <collation name>]
 | [NULL|NOT NULL]
 | [<column constraints>]
 | [column_name AS computed_column_expression]
 | [<table constraint>]
 [, . . . n]
)
[ON {<filegroup>| DEFAULT} ]
[TEXTIMAGE_ON {<filegroup>| DEFAULT} ]
```

Now that's a handful—and it still has sections taken out of it for simplicity's sake! As usual, let's look at the parts, starting with the second line (we've already seen the top line).

Table and Column Names

The rules for naming tables and columns are, in general, the same rules that apply to all database objects. The SQL Server documentation will refer to these as the *rules for identifiers*, and they are the same rules we observed at the end of Chapter 1. The rules are actually pretty simple; what we want to touch on here though, are some notions about how exactly to name our objects—not specific rules of what SQL Server will and won't accept for names, but how we want to go about naming our tables and columns so that they are useful and make sense.

There are a ton of different “standards” out there for naming database objects—particularly tables and columns. My rules are pretty simple:

- ❑ For each word in the name, capitalize the first letter and use lowercase for the remaining letters.
- ❑ Keep the name short, but make it long enough to be descriptive.
- ❑ Limit the use of abbreviations. The only acceptable use of abbreviations is when the chosen abbreviation will be recognized by anyone. Examples of abbreviations I use include “ID” to take the place of identification, “No” to take the place of number, and “Org” to take the place of organization. Keeping your names of reasonable length will require you to be more cavalier about your abbreviations sometimes, but keep in mind that, first and foremost, you want clarity in your names.
- ❑ When building tables based on other tables (usually called linking or associate tables), you should include the names of all of the parent tables in your new table name. For example, say you have a movie database where many stars can appear in many movies. If you have a `Movies` table and a `Stars` table, you may want to tie them together using a table called `MovieStars`.
- ❑ When you have two words in the name, do not use any separators (run the words together)—use the fact that you capitalize the first letter of each new word to figure out how to separate words.

I can't begin to tell you the battles I've had with other database people about naming issues. For example, you will find that a good many people believe that you should separate the words in your names with an underscore (`_`). Why don't I do it that way? Well, it's an ease-of-use issue. Underscores present a couple of different problems:

- ❑ First, many people have a difficult time typing an underscore without taking their hand away from the proper keyboard position—this leads to lots of typos.
- ❑ Second, in documentation it is not uncommon to run into situations where the table or column name is underlined. Underscores are, depending on the font, impossible to see when the text is underlined—this leads to confusion and more errors.
- ❑ Finally (and this is a nitpick), it's just more typing.

Beginning with SQL Server 7.0, it also became an option to separate the words in the name using a regular space. Do not go there! It is extremely bad practice and creates an unbelievable number of errors. It was added to facilitate Access upsizing, and I continue to curse the person(s) who decided to put it in—I'm sure they were well-meaning, but they are now part of the cause of much grief in the database world.

Chapter 4

This list is certainly not set in stone, rather it is just a *Reader's Digest* version of the rules I use when naming tables. I find that they save me a great deal of grief. I hope they'll do the same for you.

Consistency, consistency, consistency. Every time I teach, I always warn my class that it's a word I'm going to repeat over and over, and in no place is it more important than in naming. If you have to pick one rule to follow, then pick a rule that says that, whatever your standards are—make them just that: standard. If you decide to abbreviate for some reason, then abbreviate that word every time (the same way), and make sure everyone in your organization is doing the same thing (write a guide for it, and make sure people follow it). Regardless of what you're doing in your naming, make it apply to the entire database consistently. This will save a ton of mistakes, and it will save your users time in terms of how long it takes for them to get to know the database.

Data Types

There isn't much to this—the data types are as I described them in Chapter 1. You just need to provide a data type immediately following the column name—there is no default data type.

DEFAULT

You'll cover this in much more detail in Chapter 5, but for now, suffice to say that this is the value you want to be used for any rows that are inserted without a user-supplied value for this particular column. The default, if you use one, should immediately follow the data type.

IDENTITY

The *identity* value is a very important concept in database design. What is an identity column? Well, when you make a column an identity column, SQL Server automatically assigns a sequenced number to this column with every row you insert. The number that SQL Server starts counting from is called the *seed* value, and the amount that the value increases or decreases by with each row is called the *increment*. The default is for a seed of 1 and an increment of 1, and most designs call for it to be left that way. As an example though, you could have a seed of 3 and an increment of 5. In this case, you would start counting from 3, and then add 5 each time for 8, 13, 18, 23, and so on.

An identity column must be numeric, and, in practice, it is almost always implemented with an integer or bigint data type. Use an int (or even smaller) if it's big enough to meet your needs. Remember that using a larger datatype means more I/O later on.

The usage is pretty simple; you simply include the `IDENTITY` keyword right after the data type for the column. An identity option cannot be used in conjunction with a default constraint. This makes sense if you think about it—how can there be a constant default if you're counting up or down every time?

It's worth noting that an identity column works sequentially. That is, once you've set a seed (the starting point) and the increment, your values only go up (or down if you set the increment to a negative number). There is no automatic mechanism to go back and fill in the numbers for any rows you may have deleted. If you want to fill in blank spaces like that, you need to use `SET IDENTITY_INSERT ON`, which allows you to turn off (yes, turning it "on" turns it off—that is, you are turning on the ability to insert your own values, which has the effect of turning off the automatic value) the identity process for inserts from the current connection.

The most common use for an identity column is to generate a new value to be used as an identifier for each row—that is, identity columns are commonly used to create a primary key for a table. Keep in mind, however, that an `IDENTITY` column and a `PRIMARY KEY` are completely separate notions—that is, just because you have an `IDENTITY` column doesn't mean that the value is unique (for example, you can reset the seed value and count back up through values you've used before). `IDENTITY` values are *usually* used as the `PRIMARY KEY` column, but they don't *have* to be used that way.

If you've come from the Access world, you'll notice that an `IDENTITY` column is much like an `AutoNumber` column. The major difference is that you have a bit more control over it in SQL Server.

NOT FOR REPLICATION

This one is very tough to deal with at this point, so I am, at least in part, going to skip it until you come to Chapter 20.

Briefly, replication is the process of automatically doing what, in a very loose sense, amounts to copying some or all of the information in your database to some other database. The other database may be on the same physical machine as the original, or it may be located remotely.

The `NOT FOR REPLICATION` parameter determines whether a new identity value for the new database is assigned when the column is published to another database (via replication), or whether it keeps its existing value. There will be much more on this at a later time.

ROWGUIDCOL

This is also replication related and, in many ways, is the same in purpose as an identity column. You've already seen how using an identity column can provide you with an easy way to make sure that you have a value that is unique to each row and can, therefore, be used to identify that row. However, this can be a very error-prone solution when you are dealing with replicated or other distributed environments.

Think about it for a minute—while an identity column will keep counting upwards from a set value, what's to keep the values from overlapping on different databases? Now, think about when you try to replicate the values such that all the rows that were previously in separate databases now reside in one database—uh oh! You now will have duplicate values in the column that is supposed to uniquely identify each row!

Over the years, the common solution for this was to use separate seed values for each database you were replicating to and from. For example, you may have database A that starts counting at 1, database B starts at 10,000, and database C starts at 20,000. You can now publish them all into the same database safely—for a while. As soon as database A has more than 9,999 records inserted into it, you're in big trouble.

Chapter 4

“Sure,” you say, “why not just separate the values by 100,000 or 500,000?” If you have tables with a large amount of activity, you’re still just delaying the inevitable — that’s where a `ROWGUIDCOL` comes into play.

What is a `ROWGUIDCOL`? Well, it’s quite a bit like an identity column in that it is usually used to uniquely identify each row in a table. The difference is to what lengths the system goes to make sure that the value used is truly unique. Instead of using a numerical count, SQL Server instead uses what is known as a *GUID*, or a *Globally Unique Identifier*. While an identity value is usually (unless you alter something) unique across time, it is not unique across space. Therefore, you can have two copies of your table running, and have them both assigned an identical identity value. While this is just fine to start with, it causes big problems when you try to bring the rows from both tables together as one replicated table. A GUID is unique across both space and time.

GUIDs are fairly ubiquitous in computing today. For example, if you check the registry, you’ll find tons of them. A GUID is a 128-bit value — for you math types, that’s 38 zeros in decimal form. If I generated a GUID every second, it would, theoretically speaking, take me millions of years to generate a duplicate given a number of that size.

GUIDs are generated using a combination of information — each type of which is designed to be unique in either space or time. When you combine them, you come up with a value that is guaranteed, statistically speaking, to be unique across space and time.

There is a Win32 API call to generate a GUID in normal programming, but, in addition to the `ROWGUIDCOL` option on a column, SQL has a special function to return a GUID — it is called the `NEWID()` function, and can be called at any time.

COLLATE

This works pretty much just as it did for the `CREATE DATABASE` command, with the primary difference being the scope (here, you define at the column level rather than the database level).

NULL/NOT NULL

This one is pretty simple — it states whether the column in question accepts `NULL` values or not. The default, when you first install SQL Server, is to set a column to `NOT NULL` if you don’t specify nullability. There are, however, a very large number of different settings that can affect this default, and change its behavior. For example, setting a value by using the `sp_dbcmptlevel` stored procedure or setting ANSI-compliance options can change this value.

I highly recommend explicitly stating the `NULL` option for every column in every table you ever build. Why? As I mentioned before, there are a large number of different settings that can affect what the system uses for a default for the nullability of a column. If you rely on these defaults, then you may find later that your scripts don’t seem to work right (because you or someone else has changed a relevant setting without realizing its full effect).

Column Constraints

Constraints are restrictions and rules that you place on individual columns about the data that can be inserted into that column—for example, restricting a column that contained the number month of the year to numbers between 1 and 12.

Computed Columns

You can also have a column that doesn't have any data of its own but whose value is derived on the fly from other columns in the table. If you think about it, this may seem odd since you could just figure it out at query time, but really, this is something of a boon for many applications. Why? Well, for two major reasons: First, it allows you to represent what might be a difficult calculation in a standard and easy-to-retrieve manner (frankly, it can be nice even on simple calculations), but in a manner that does not have to take up physical disk space. Second, you can index this column, and, while that will mean that it will take up disk space, it also means you can perform some very fast query lookups on this computed data.

Take a look at the specific syntax:

```
<column name> AS <computed column expression>
```

The first item is a little different—you're providing a column name to go with your value. This is simply the alias that you're going to use to refer to the value that is computed, based on the expression that follows the `AS` keyword.

Next comes the computed column expression. The expression can be any normal expression that uses either literals or column values from the same tables. For example:

```
ExtendedPrice AS Price * Quantity
```

Pretty easy, eh? There are a few caveats and provisos though:

- ❑ You cannot use a subquery, and the values cannot come from a different table.
- ❑ Prior to SQL Server 2000, you could not use a computed column as any part of any key (primary, foreign, or unique) or with a default constraint. While you can now use a computed column as a primary key or unique constraint—foreign keys and defaults are still “no-no’s.”

You'll look at specific examples of how to use computed columns a little later in this chapter.

Table Constraints

Table constraints are quite similar to column constraints, in that they place restrictions on the data that can be inserted into the table. What makes them a little different is that they may be based on more than one column.

Again, you will be covering these in the constraints chapter, but examples of table-level constraints include `PRIMARY` and `FOREIGN KEY` constraints, as well as `CHECK` constraints.

OK, so why is a `CHECK` constraint a table constraint? Isn't it a column constraint, since it affects what you can place in a given column? The answer is that it's both. If it is based on solely one column, then it meets the rules for a column constraint. If, however, (as `CHECK` constraints can be) it is dependent on multiple columns, then you have what would be referred to as a table constraint.

ON

Remember when you were dealing with database creation, and I said you could create different filegroups? Well, the `ON` clause in a table definition is a way of specifically stating on which filegroup (and, therefore, physical device) you want the table located. You can place a given table on a specific physical device, or, as you will want to do in most cases, just leave the `ON` clause out, and the table will be placed on whatever the default filegroup is (which will be the `PRIMARY` unless you've set it to something else). You will be looking at this usage extensively in the chapter on performance tuning.

TEXTIMAGE_ON

This one is basically the same as the `ON` clause you just looked at, except that it lets you move a very specific part of the table to yet a different filegroup. This clause is valid only if your table definition has `text`, `ntext`, or `image` column(s) in it. When you use the `TEXTIMAGE_ON` clause, you move only the `BLOB` information into the separate filegroup — the rest of the table stays either on the default filegroup or with the filegroup chosen in the `ON` clause.

There can be some serious performance increases to be had by splitting your database up into multiple files and then storing those files on separate physical disks. When you do this, it means you get the I/O from both drives. Major discussion of this is outside the scope of this book, but keep this in mind as something to gather more information on should you run into I/O performance issues.

Creating a Table

When we started this section, we looked at the standard `CREATE` syntax of:

```
CREATE <object type> <object name>
```

And then we moved on to a more specific start (indeed, it's the first line of our statement that will create the table) on creating a table called `Customers`:

```
CREATE TABLE Customers
```

We're going to add in a `USE <database name>` line prior to our `CREATE` code so that we're sure that, when we run the script, the table is created in the proper database. We'll then follow up that first line that we've already seen with a few columns.

```
USE Accounting
CREATE TABLE Customers
(
    CustomerNo      int          IDENTITY NOT NULL,
    CustomerName   varchar(30)   NOT NULL,
    Address1        varchar(30)   NOT NULL,
    Address2        varchar(30)   NOT NULL,
    City            varchar(20)   NOT NULL,
    State           char(2)      NOT NULL,
    Zip             varchar(10)   NOT NULL,
    Phone           char(15)     NOT NULL
)
```

This is a somewhat simplified table versus what you would probably use in real life, but there's plenty of time to change it later (and we will). If we want to verify that it was indeed created, we can make use of several commands, but perhaps the best is one that will seem like an old friend before you're done with this book: `sp_help`. The syntax is simple:

```
EXEC sp_help <object name>
```

To specify the table object that we just created, try executing the following code:

```
EXEC sp_help Customers
```

The ALTER Statement

Okay, so now you have a table — isn't life grand? If only things always stayed the same, but they don't. Sometimes (actually, far more often than you would like), you get requests to *change* a table rather than recreate it. Likewise, you have needs to change the size, file locations, or some other feature of your database. That's where your `ALTER` statement comes in.

Much like the `CREATE` statement, your `ALTER` statement pretty much always starts out the same:

```
ALTER <object type> <object name>
```

This is totally boring so far, but it won't stay that way. You'll see the beginnings of issues with this statement right away, and things will get really interesting (read: convoluted and confusing!) when you deal with this even further in the next chapter (when you deal with constraints).

ALTER DATABASE

You can get right into it by taking a look at changing your database. You'll actually make a couple of changes just so you can see the effects of different things and how their syntax can vary.

Perhaps the biggest trick with the `ALTER` statement is to remember what you already have. With that in mind, take a look again at what you already have:

```
EXEC sp_helpdb Accounting
```

Chapter 4

So, the results should be just like they were when you created the database:

Name	db_size	owner	dbid	created	status	compatibility_level
Accounting	15.00 MB	sa	9	May 28 2000	Status=ONLINE, Updateability= READ_WRITE, UserAccess= MULTI_USER, Recovery=FULL, Version=598, Collation=SQL_ Latin1_General_ CP1_CI_AS, SQLSortOrder=52, IsAutoCreate Statistics, IsAuto UpdateStatistics, IsFullTextEnabled	90

And . . .

Name	fileid	filename	filegroup	size	maxsize	growth	usage
Accounting	1	c:\Program Files\ Microsoft SQL Server\MSSQL.1\mssql\data\AccountingData.mdf	PRIMARY	10240 KB	51200 KB	5120 KB	data only
AccountingLog	2	c:\Program Files\ Microsoft SQL Server\MSSQL.1\mssql\data\AccountingLog.ldf	NULL	5120 KB	25600 KB	5120 KB	log only

Say you want to change things a bit. For example, suppose that you know that you are going to be doing a large import into your database. Currently, your database is only 15MB in size—that doesn't hold much these days. Since you have autogrow turned on, you could just start your import, and SQL Server would automatically enlarge the database 5MB at a time. Keep in mind, however, that it's actually a fair amount of work to reallocate the size of the database. If you were inserting 100MB worth of data, then the server would have to deal with that reallocation at least 16 times (at 20MB, 25MB, 30MB, etc.). Since you know that you're going to be getting up to 100MB of data, why not just do it in one shot? To do this, you would use the `ALTER DATABASE` command.

The general syntax looks like this:

```
ALTER DATABASE <database name>
ADD FILE
  ([NAME = <'logical file name'>,]
   FILENAME = <'file name'>
   [, SIZE = <size in KB, MB, GB or TB>]
   [, MAXSIZE = < size in KB, MB, GB or TB >]
   [, FILEGROWTH = <No of KB, MB, GB or TB /percentage>]) [,...n]
   [ TO FILEGROUP filegroup_name]
   [, OFFLINE ]
| ADD LOG FILE
  ([NAME = <'logical file name'>,]
   FILENAME = <'file name'>
   [, SIZE = < size in KB, MB, GB or TB >]
   [, MAXSIZE = < size in KB, MB, GB or TB >]
   [, FILEGROWTH = <No KB, MB, GB or TB /percentage>])
| REMOVE FILE <logical file name> [WITH DELETE]
| ADD FILEGROUP <filegroup name>
| REMOVE FILEGROUP <filegroup name>
| MODIFY FILE <filespec>
| MODIFY NAME = <new dbname>
| MODIFY FILEGROUP <filegroup name> {<filegroup property>|NAME =
  <new filegroup name>}
| SET <optionspec> [,...n] [WITH <termination>]
| COLLATE <collation name>
```

The reality is that you will very rarely use all that stuff—sometimes I think Microsoft just puts it there for the sole purpose of confusing the heck out of you (just kidding!).

So, after looking at all that gobbledegook, just worry about what you need to expand your database out to 100MB:

```
ALTER DATABASE Accounting  
    MODIFY FILE  
        (NAME = Accounting,  
         SIZE = 100MB)
```

Note that, unlike when you created your database, you don't get any information about the allocation of space—instead, you get the rather non-verbose:

The command(s) completed successfully.

One thing worth noticing is that, even though you exceeded the previous maximum size of 51,200KB, you didn't get an error. This is because you *explicitly* increased the size. It was, therefore, implied that you must have wanted to increase the maximum, too. If you had done things your original way of just letting SQL Server expand things as necessary, your import would have blown up in the middle because of the size restriction. One other item worth noting here is that the maxsize was only increased to your new explicit value — there now isn't any room for growth left.

Things pretty much work the same for any of the more common database-level modifications you'll make. The permutations are, however, endless. The more complex filegroup modifications and the like are outside the scope of this book, but, if you need more information on them, I would recommend one of the more administrator-oriented books out there (and there are a ton of them).

Option and Termination Specs

SQL Server has a few options that can be set with an `ALTER DATABASE` statement. Among these are database-specific defaults for most of the `SET` options that are available (such as `ANSI_PADDING`, `ARITHABORT`—handy if you're dealing with indexed or partitioned views), state options (for example, single-user mode or read-only), and recovery options. The effects of the various `SET` options are discussed where they are relevant throughout the book. This new `ALTER` functionality simply gives you an additional way to change the defaults for any particular database.

SQL Server also has the ability to control the implementation of some of the changes you are trying to make on your database. Many changes require that you have exclusive control over the database—something that can be hard to deal with if other users are already in the system. SQL Server gives you the ability to gracefully force other users out of the database so that you may complete your database changes. The strength of these actions ranges from waiting a number of seconds (you decide how long) before kicking other users out, all the way up to immediate termination of any option transactions (automatically rolling them back). Relatively uncontrolled (from the client's perspective) termination of transactions is not something to be taken lightly. Such an action is usually in the realm of the database administrator. Therefore, we will consider further discussion out of the scope of this book.

ALTER TABLE

A far, far more common need is the situation where you need to change the make-up of your table. This can range from simple things like adding a new column to more complex issues such as changing a data type.

Start out by taking a look at the basic syntax for changing a table:

```
ALTER TABLE table_name
  { [ALTER COLUMN <column_name>
      { [<schema of new data type>].<new_data_type> [(precision [, scale])] max |
      <xml schema collection>
        [COLLATE <collation_name>]
        [NULL|NOT NULL]
        | [{ADD|DROP} ROWGUIDCOL] | PERSISTED]
      | ADD
        <column_name> <data_type>
        [[DEFAULT <constant_expression>]
        | [IDENTITY [(<seed>, <increment>) [NOT FOR REPLICATION]]]
        | [ROWGUIDCOL]
        | [COLLATE <collation_name>]
          [NULL|NOT NULL]
          [<column_constraints>]
          | [<column_name> AS <computed_column_expression>]
      | ADD
        [CONSTRAINT <constraint_name>]
        { {{PRIMARY KEY|UNIQUE}
          [CLUSTERED|NONCLUSTERED]
          {(<column_name>[ , . . . n ])}
          [WITH FILLFACTOR = <fillfactor>]
```

```

[ON {<filegroup> | DEFAULT}]
]
| FOREIGN KEY
  [(<column_name>[ ,...n])]
  REFERENCES <referenced_table> [(<referenced_column>[ ,...n])]
  [ON DELETE {CASCADE|NO ACTION}]
  [ON UPDATE {CASCADE|NO ACTION}]
  [NOT FOR REPLICATION]
|DEFAULT <constant_expression>
  [FOR <column_name>]
|CHECK [NOT FOR REPLICATION]
  (<search_conditions>
  [,...n][ ,...n]
  | [WITH CHECK|WITH NOCHECK]
| { ENABLE | DISABLE } TRIGGER
  { ALL | <trigger_name> [ ,...n ] }

|DROP
  {[CONSTRAINT] <constraint_name>
    |COLUMN <column_name>[ ,...n]
  |{CHECK|NOCHECK} CONSTRAINT
    {ALL|<constraint_name>[ ,...n]}
  |{ENABLE|DISABLE} TRIGGER
    {ALL|<trigger_name>[ ,...n]}
  | SWITCH [ PARTITION <source partition number expression> ]
    TO [ schema_name. ] target_table
    [ PARTITION <target partition number expression> ]
}

```

As with the `CREATE TABLE` command, there's quite a handful there to deal with.

So, start an example of using this by looking back at your `Employees` table in the `Accounting` database:

```
EXEC sp_help Customers
```

For the sake of saving a few trees, I'm going to edit the results that I show here to just the part you care about—you'll actually see much more than this:

Column_name	Type	Computed	Length	Prec	Scale	Nullable
CustomerNo	int	no	4	10	0	no
CustomerName	varchar	no	30			no
Address1	varchar	no	30			no
Address2	varchar	no	30			no
City	varchar	no	20			no
State	char	no	2			no
Zip	varchar	no	10			no
Phone	char	no	15			no

Chapter 4

Say that you've decided you'd like to keep the Federal Identification Number for your Customers. That just involves adding another column, and really isn't all that tough. The syntax looks much like it did with your CREATE TABLE statement except that it has obvious alterations to it:

```
ALTER TABLE Customers
ADD
    FedIDNo    varchar(9)    NULL
```

Not exactly rocket science — is it? Indeed, you could have added several additional columns at one time if you had wanted to. It would look something like this:

```
ALTER TABLE Customers
ADD
    Contact      varchar(25)    NULL,
    LastRaiseDate datetime      NOT NULL
        DEFAULT '2005-11-07'
```

Notice the DEFAULT I slid in here. You haven't really looked at these yet (they are in the next chapter), but I wanted to use one here to point out a special case.

If you want to add a NOT NULL column after the fact, you have the issue of what to do with rows that already have NULL values. You have shown the solution to that here by providing a default value. The default is then used to populate the new column for any row that is already in your table.

Use `sp_help` to look at the `Customers` table, and you'll see your new columns have been added. The thing to note, however, is that they all went to the end of the column list. There is no way to add a column to a specific location in SQL Server. If you want to move a column to the middle, you need to create a completely new table (with a different name), copy the data over to the new table, `DROP` the existing table, and then rename the new one.

This issue of moving columns around can get very sticky indeed. Even some of the tools that are supposed to automate this often have problems with it. Why? Well, any foreign key constraints you have that reference this table must first be dropped before you are allowed to delete the current version of the table. That means that you have to drop all your foreign keys, make the changes, and then add all your foreign keys back. It doesn't end there, however; any indexes you have defined on the old table are automatically dropped when you drop the existing table — that means that you must remember to recreate your indexes as part of the build script to create your new version of the table — yuck!

*But wait! There's more! While you haven't really looked at views yet, I feel compelled to make a reference here to what happens to your views when you add a column. You should be aware that, even if your view is built using a `SELECT *` as its base statement, your new column will not appear in your view until you rebuild the view. Column names in views are resolved at the time the view is created for performance reasons. That means any views that have already been created when you add your columns have already resolved using the previous column list — you must either `DROP` and recreate the view or use an `ALTER VIEW` statement to rebuild it.*

The DROP Statement

Performing a DROP is the same as deleting whatever object(s) you reference in your DROP statement. It's very quick and easy, and the syntax is exactly the same for all of the major SQL Server objects (tables, views, sprocs, triggers, etc.). It goes like this:

```
DROP <object type> <object name> [, ...n]
```

Actually, this is about as simple as SQL statements get.

```
USE Accounting  
DROP TABLE Customers
```

Poof—gone.

Be very careful with this command. There is no, “Are you sure?” kind of question that goes with this—it just assumes you know what you’re doing and deletes the object(s) in question.

The syntax is very much the same for dropping the entire database. Now drop the Accounting database:

```
USE master  
DROP DATABASE Accounting
```

You should see the following in the Results pane:

```
Deleting database file 'c:\Program Files\Microsoft SQL  
Server\mssql\data\AccountingLog.ldf'.  
Deleting database file 'c:\Program Files\Microsoft SQL  
Server\mssql\data\AccountingData.mdf'.
```

You may run into a situation where you get an error that says that the database cannot be deleted because it is in use. If this happens, check a couple of things:

- Make sure that the database that you have as current in the Management Studio is something other than the database you’re trying to drop (that is, make sure you’re not using the database as you’re trying to drop it).
- Ensure that you don’t have any other connections open (using the Management Studio or `sp_who`) that are showing the database you’re trying to drop as the current database.

I usually solve the first one just as you did in the code example—I switch to using the `master` database. The second you have to check manually—I usually close other sessions down entirely just to be sure.

Using the GUI Tool

You've just spent a lot of time pounding in perfect syntax for creating a database and a couple of tables—that's enough of that for a while. Take a look at the graphical tool in the Management Studio that allows you to build and relate tables. From this point on, you'll not only be dealing with code, but with the tool that can generate much of that code for us.

Since, in theory, you already know most of what is going on here, you're going to do this at a whirlwind pace—in short, I'm going to show you where to find it.

Creating or Editing the Database

To create a database using the SQL Server Management Studio, right-click the databases node and select "New Database..." as in Figure 4-1.

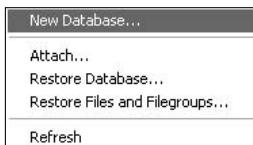


Figure 4-1

This will bring you up a fairly straightforward dialog, as shown in Figure 4-2, that gives you text boxes to fill in all the things that you had in your original SQL syntax at the beginning of the chapter.

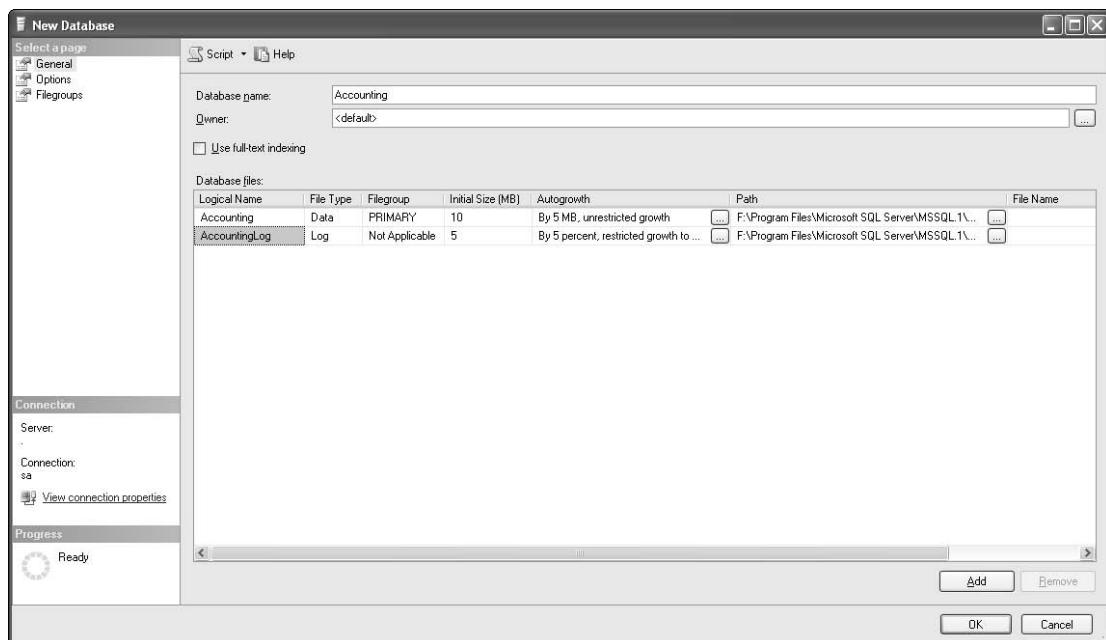


Figure 4-2

There are a few extra dialogs you can see by clicking on “Options” and “Filegroups” on the left, but these are doing nothing more than showing the SQL options that didn’t fit in the General dialog.

Likewise, you can edit a database by right-clicking the database you want to edit and selecting “Properties,” as in Figure 4-3.

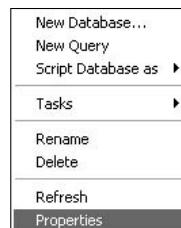


Figure 4-3

This will bring you up what is largely the same dialog you had in Figure 4-2, but with editing in mind instead of creating.

Creating and Editing Tables

Creating and editing tables using the Management Studio works similarly to creating and editing databases, but with a few extra twists. Right-click on the Tables node of the explorer tree in the Management Studio, and choose New Table... as in Figure 4-4.

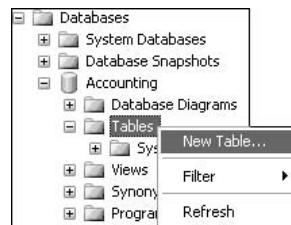


Figure 4-4

When you’re done, you have a chance to add columns and such, as shown in Figure 4-5. Most of this should be pretty explanatory, but I have two things I want to point out.

First, notice that by default the identity column is identified in the properties section on the right. You choose which column is to be the identity column for that table (there can only be one). Second, you can change additional things about the table by right-clicking it and choosing from the context menu shown in Figure 4-6.

Chapter 4

The screenshot shows the SQL Server Management Studio interface. The main window displays the schema of 'Table - dbo.Table_1'. The table has seven columns: CustomerNo, CustomerName, Address1, Address2, City, State, Zip, and Phone. The 'CustomerNo' column is defined as an int type and is marked as the Identity Column. The 'Properties' pane on the right shows the table's properties, including its name 'Table_1', database 'Accounting', schema 'dbo', and server 'schweitzer'. The 'Table Designer' section indicates it is a regular data space with no replication or row GUID features.

Column Name	Data Type	Allow Nulls
CustomerNo	int	<input type="checkbox"/>
CustomerName	varchar(30)	<input type="checkbox"/>
Address1	varchar(30)	<input type="checkbox"/>
Address2	varchar(30)	<input type="checkbox"/>
City	varchar(20)	<input type="checkbox"/>
State	char(2)	<input type="checkbox"/>
Zip	varchar(50)	<input type="checkbox"/>
Phone	char(15)	<input type="checkbox"/>

Figure 4-5

A context menu is open over the 'CustomerNo' column header in the Table Designer. The menu options include: Set Primary Key, Insert Column, Delete Column, Relationships..., Indexes/Keys..., Fulltext Index..., XML Indexes..., Check Constraints..., and Generate Change Script... .

Figure 4-6

Again, there's nothing here that you shouldn't be able to do via scripting (although you haven't seen some of it quite yet). It is also worth noting that the final item in the context menu—Generate Change Script—will take a table that you have and script out the SQL to generate it (which can be kind of handy).

At any time, just click the save icon and the Management Console will prompt you for a table name and create the table for you.

Editing works just slightly different than it did for databases. Right-click the Tables element in the explorer tree as in Figure 4-7, but this time you choose "Modify" instead of Properties.

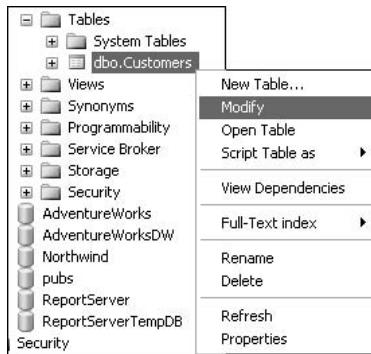


Figure 4-7

From there, you get the same dialog you saw in Figure 4-5.

Modifying the column order using this GUI tool *will* reorder your columns for you.
The SQL Server Management Studio scripts the creation of a new table, migrates the data, and then renames the new table. While this is handy for getting your column order as you would like it, keep in mind that this may be very time-consuming for tables that have large amounts of data—plan accordingly.

Summary

In this chapter, we've covered the basics of the CREATE, ALTER, and DROP statements as they relate to creating a database and tables. There are, of course, many other renditions of these that we will cover as we continue through the book. We have also taken a look at the wide variety of options that we can use in databases and tables to have full control over our data. Finally, we have begun to see the many things that you can use the Management Studio for in order to simplify our lives, and make design and scripting simpler.

At this point, we're done reviewing how to lay out your tables. We'll spend the next few chapters reviewing many of the ancillary items that go along with our tables (such as constraints) as well as some more advanced query constructs. From there, we're ready to review normalization and other basic database design as well as move into more advanced design.

5

Reviewing Keys and Constraints

You've heard me talk about them, but now it's time for a serious review of *keys* and *constraints*. SQL Server has had many changes in this area over the last few versions, and that trend has continued with SQL Server 2005.

We've talked a couple of times already about what constraints are, but let's review in case you decided to skip straight to this chapter.

A constraint is a restriction. Placed at either column or table level, a constraint ensures that your data meets certain data integrity rules.

This gets back to the notion that I talked about in Chapter 1, where ensuring data integrity is not the responsibility of the programs that use your database, but rather the responsibility of the database. If you think about it, this is really cool. Data is inserted, updated, and deleted from the database by many sources. Even in standalone applications (situations where only one program accesses the database), the same table may be accessed from many different places in the program. It doesn't stop there though. Your database administrator (that might mean you if you're a dual role kind of person) may be altering data occasionally to deal with problems that arise. In more complex scenarios, hundreds of different access paths can exist for altering just one piece of data, let alone your entire database.

Moving the responsibility for data integrity into the database itself has been revolutionary to database management. There are still many different things that can go wrong when you are attempting to insert data into your database, but your database is now *proactive* rather than *reactive*.

Chapter 5

to problems. Many problems with what programs allow into the database are now caught much earlier in the development process because, although the client program allowed the data through, the database knows to reject it. How does it do it? Primarily with constraints. (Data types and triggers are among the other worker bees of data integrity.)

In this chapter, we will look at three types of constraints at a high level:

- Entity constraints
- Domain constraints
- Referential integrity constraints

At a more specific level, we'll be looking at the specific methods of implementing each of these types of constraints:

- PRIMARY KEY constraints
- FOREIGN KEY constraints
- UNIQUE constraints (also known as alternate keys)
- CHECK constraints
- DEFAULT constraints
- Rules
- Defaults (similar to, yet different from, DEFAULT constraints)

SQL Server 2000 was the first version to support two of the most commonly requested forms of referential integrity actions: cascade updates and cascade deletes. These were common complaint areas, but Microsoft left some other areas of ANSI referential integrity support out. These were added in SQL Server 2005. You look at cascading and other ANSI referential integrity actions in detail when you look at FOREIGN KEY constraints.

We also take a cursory look at triggers and stored procedures as a ways of implementing data integrity rules.

Types of Constraints

There are several ways to implement constraints, but each of them falls into one of three categories—entity, domain, or referential integrity constraints—as illustrated in Figure 5-1.

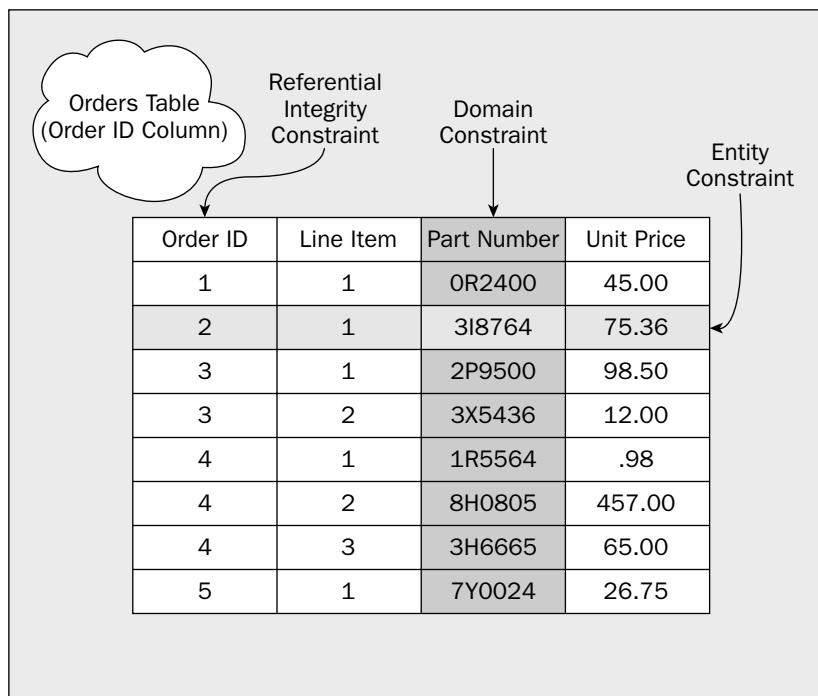


Figure 5-1

Domain Constraints

Domain constraints deal with one or more columns. They ensure that a particular column or set of columns meets particular criteria. When you insert or update a row, the constraint is applied without respect to any other row in the table; it's the column's data you're interested in.

For example, if you want to confine the `UnitPrice` column to values that are greater than or equal to zero, you would use a domain constraint. Although any row that had a `UnitPrice` that didn't meet the constraint would be rejected, you're actually enforcing integrity to make sure that entire column (no matter how many rows) meets the constraint. The domain is the column, and the constraint is a domain constraint.

You'll use this kind of constraint when dealing with `CHECK` constraints, rules, defaults, and `DEFAULT` constraints.

Entity Constraints

Entity constraints are all about individual rows. This form of constraint doesn't really care about a column as a whole. It's interested in a particular row, and would best be exemplified by a constraint that requires every row to have a unique value for a column or combination of columns.

Chapter 5

“What,” you say, “a unique column? Doesn’t that mean it’s a domain constraint?” No, it doesn’t. You’re not requiring a column to meet any particular format, nor that the value be greater or less than anything. What you’re saying is that for *this* row, the same value can’t already exist in some other row.

You’ll use this kind of constraint when dealing with PRIMARY KEY and UNIQUE constraints.

Referential Integrity Constraints

Referential integrity constraints are created when a value in one column must match the value in another column — in the same table or, more typically, a different table.

Imagine you are taking product orders, and that you accept credit cards. To be paid by the credit card company, you need to have some form of merchant agreement with that company. You don’t want employees to take credit cards from companies that aren’t going to pay you. That’s where referential integrity comes in: it enables you to build a *domain* or *lookup table*. A domain table is a table designed to provide a limited list of acceptable values. In this case, you might build a table that looks something like this:

CreditCardID	CreditCard
1	VISA
2	MasterCard
3	Discover Card
4	American Express

You can then build one or more tables that *reference* the CreditCardID column of the domain table. With referential integrity, any table, such as our Orders table, that is defined as referencing our CreditCard table will have to have a column that matches up to the CreditCardID column of our CreditCard table. Each row that we insert into the referencing table must have a value that is in our domain list. (It must have a corresponding row in the CreditCard table.)

You’ll see more of this as you learn about FOREIGN KEY constraints later in this chapter.

Constraint Naming

Before we get down to the nitty-gritty constraints, I’ll digress for a moment and address the issue of naming constraints.

For each type of constraints that you will be dealing with in this chapter, you can elect not to supply a name — that is, you can have SQL Server provide a name for you. Resist the temptation to do this. You’ll quickly find that when SQL Server creates its own name, it isn’t particularly useful.

An example of a system-generated name might be something like `PK_Customers_145C0A3F`. This is a SQL Server generated name for a primary key on the `Customers` table of the Accounting database, which you recreate (we get rid of the old one from last chapter and rebuild it) later in the chapter. The PK is for primary key, which is the major thing that makes it useful. The `Customers` is for the `Customers` table that it is on, and the rest is a randomly generated value to ensure uniqueness. You only get this type of naming if you create the primary key through script. If you created this table through Management Studio, it would have a name of `PK_Customers`.

That one isn't too bad, but you get less help on other constraints, for example, a `CHECK` constraint used later in the chapter might generate something like `CK_Customers__22AA2996`. From this, you know that it's a `CHECK` constraint, but you know nothing about the nature of the `CHECK`.

Because you can have multiple `CHECK` constraints on a table, you could wind up with all these (or other randomly generated numbers in them) as names of constraints on the same table:

`CK_Customers__22AA2996`

`CK_Customers__25869641`

`CK_Customers__267ABA7A`

Needless to say, if you needed to edit one of these constraints, it would be a pain to figure out which was which.

I use a combination of type of constraint with a phrase to indicate what it does or the name(s) of the column(s) it affects. For example, I might use `CKPriceExceedsCost` if I have a constraint to ensure that my users can't sell a product at a loss, or perhaps something as simple as `CKCustomerPhoneNo` on a column that ensures that phone numbers are formatted properly.

As with the naming of anything that you'll use in this book, how exactly you name things is really not all that important. What is important is that you adhere to these simple guidelines:

- Be consistent.
- Make it something that everyone can understand.
- Keep it as short as you can while still meeting the above rules.
- Did I mention to be consistent?

Key Constraints

There are four different types of common keys that you may hear about in your database endeavors. These are primary keys, foreign keys, alternate keys, and inversion keys. This chapter looks at the first three of these, as they provide constraints on the database.

An inversion key is basically just any index that doesn't apply some form of constraint to the table (primary key, foreign key, unique). (I cover indexes in Chapter 8.) Inversion keys, rather than enforcing data integrity, are merely an alternative way of sorting the data.

Keys are one of the cornerstone concepts of database design and management, so fasten your seatbelt and hold on tight. I'm hoping that by the time you got to a "Professional" level book, you already had the fundamentals of this, but this is one of the most important concepts you'll read about in this book, and it will become absolutely critical as you move on to design in Chapter 7.

PRIMARY KEY Constraints

Before I define what a primary key actually is, I want to digress into a brief discussion of relational databases. Relational databases are constructed on the idea of being able to "relate" data. Therefore, it becomes critical in relational databases for most tables (there are exceptions, but they are very rare) to have a unique identifier for each row. A unique identifier enables you to reference accurately a record from another table in the database; thus, forming a relation between those two tables.

This is a wildly different concept from that in the old mainframe environment or the ISAM databases (dBase, FoxPro, Clipper, etc.) of the '80s and early '90s. In those environments, we dealt with one record at a time. We would generally open the entire table, and go one record at a time until we found what we were looking for. If we needed data from a second table, we would then open that table separately and fetch that table's data, and then mix the data programmatically ourselves.

Primary keys are the unique identifiers for each row. They must contain unique values (and hence cannot be NULL). Because of their importance in relational databases, primary keys are the most fundamental of all keys and constraints.

Don't confuse the primary key, which uniquely identifies each row in a table, with a GUID, which is a more generic tool typically used to identify something (more than just rows) across all space and time. Although a GUID can certainly be used as a primary key, they incur some overhead, and are usually not called for when you're dealing with only the contents of a table. Indeed, the only common place that a GUID becomes particularly useful in a database environment is as a primary key when dealing with replicated or other distributed data.

A table can have a maximum of one primary key. As I mentioned earlier, it is rare to have a table on which you don't want a primary key.

When I say "rare" here, I mean very rare. A table that doesn't have a primary key violates the concept of relational data. It means that you can't guarantee that you can relate to a specific record. The data in your table no longer has anything that gives it distinction.

Having multiple rows that are logically identical is actually not that uncommon, but that doesn't mean that you don't want a primary key. In these instances, you'll want to take a look at fabricating some sort of key. This approach has most often been implemented using an identity column, though using a GUID now makes more sense in some situations.

A primary key ensures uniqueness within the columns declared as part of that primary key, and that unique value serves as an identifier for each row in that table. How do you create a primary key? Actually, there are two ways. You can create the primary key in your CREATE TABLE command or with an ALTER TABLE command.

Creating the Primary Key at Table Creation

Review one of the CREATE TABLE statements from the previous chapter:

```
CREATE TABLE Customers
(
    CustomerNo      int      IDENTITY   NOT NULL,
    CustomerName    varchar(30)      NOT NULL,
    Address1        varchar(30)      NOT NULL,
    City            varchar(20)      NOT NULL,
    State           char(2)        NOT NULL,
    Zip             varchar(10)      NOT NULL,
    Contact         varchar(25)      NOT NULL,
    Phone           char(15)        NOT NULL
)
```

This CREATE statement should seem old hat by now, but it's missing a very important piece—the PRIMARY KEY constraint. You want to identify CustomerNo as your primary key. Why CustomerNo? You look into what makes a good primary key in Chapter 7, but for now, just think about it a bit: Do you want two customers to have the same CustomerNo? Definitely not. It makes perfect sense for a CustomerNo to be used as an identifier for a customer. Indeed, such a system has been used for years, so there's really no sense in reinventing the wheel here.

To alter your CREATE TABLE statement to include a PRIMARY KEY constraint, you add the constraint information right after the column(s) that you want to be part of your primary key:

```
CREATE TABLE Customers
(
    CustomerNo      int      IDENTITY   NOT NULL
        PRIMARY KEY,
    CustomerName    varchar(30)      NOT NULL,
    Address1        varchar(30)      NOT NULL,
    City            varchar(20)      NOT NULL,
    State           char(2)        NOT NULL,
    Zip             varchar(10)      NOT NULL,
    Contact         varchar(25)      NOT NULL,
    Phone           char(15)        NOT NULL
)
```

Go ahead and drop the old table and create the new one defined here—you'll need it as we progress through the chapter.

Creating a Primary Key on an Existing Table

Now, what if you already have a table and you want to set the primary key? That's also easy.

Let's create a sample table called Employees. I'm creating it in the Accounting database (the original code to create that database was in the last chapter), but you could create this in the database of your choice as long as it is the same one you put the Customers table in (we will be using the two tables together later on).

Chapter 5

```
USE Accounting

CREATE TABLE Employees
(
    EmployeeID      int          IDENTITY NOT NULL,
    FirstName        varchar(25)   NOT NULL,
    MiddleInitial    char(1)       NULL,
    LastName         varchar(25)   NOT NULL,
    Title            varchar(25)   NOT NULL,
    SSN              varchar(11)   NOT NULL,
    Salary           money        NOT NULL,
    PriorSalary      money        NOT NULL,
    LastRaise AS Salary - PriorSalary,
    HireDate         smalldatetime NOT NULL,
    TerminationDate smalldatetime NULL,
    ManagerEmpID    int          NOT NULL,
    Department       varchar(25)   NOT NULL
)
```

Now, this example does not have a primary key, so let's add it:

```
USE Accounting

ALTER TABLE Employees
ADD CONSTRAINT PK_EmployeeID
PRIMARY KEY (EmployeeID)
```

The ALTER command tells SQL Server the following:

- You are adding something to the table. (You could also be dropping something from the table.)
- You're adding a constraint.
- The name of the constraint to allow you to address the constraint directly later.
- The type of constraint (`PRIMARY KEY`).
- The column(s) to which the constraint applies.

FOREIGN KEY Constraints

Foreign keys are a method of ensuring data integrity and a manifestation of the relationships between tables. When you add a foreign key to a table, you are creating a dependency between the table for which you define the foreign key (the *referencing table*) and the table your foreign key references (the *referenced table*). After adding a foreign key, any record you insert into the referencing table must have a matching record in the referenced column(s) of the referenced table, or the value of the foreign key column(s) must be set to `NULL`. This can be a little confusing, so I'll lead you through an example.

When I say that a value must be "set to `NULL`," I'm referring to the way the actual `INSERT` statement looks. As you'll learn in a moment, the data may actually look slightly different after it gets in the table depending on what options you've set in your `FOREIGN KEY` declaration.

Reviewing Keys and Constraints

Try creating another table in your Accounting database called `Orders`. One thing you'll notice in this `CREATE` script is that you're going to use a primary key and a foreign key. The foreign key is added to the script in almost exactly the same way as the primary key was, except that you must say what you are referencing. The syntax goes on the column or columns that you are placing your `FOREIGN KEY` constraint on and looks something like this:

```
<column name> <data type> <>nullability>
FOREIGN KEY REFERENCES <table name>(<column name>)
    [ON DELETE {CASCADE|NO ACTION|SET NULL|SET DEFAULT}]
    [ON UPDATE {CASCADE|NO ACTION|SET NULL|SET DEFAULT}]
```

For the moment, ignore the `ON` clause. That leaves you, for your `Orders` table, with a script that looks something like this:

```
USE Accounting

CREATE TABLE Orders
(
    OrderID      int      IDENTITY   NOT NULL
    PRIMARY KEY,
    CustomerNo   int      NOT NULL
    FOREIGN KEY REFERENCES Customers(CustomerNo),
    OrderDate    smalldatetime NOT NULL,
    EmployeeID   int      NOT NULL
)
```

Note that the actual column being referenced must have a `PRIMARY KEY` or a `UNIQUE` constraint defined on it. (I discuss `UNIQUE` constraints later in this chapter.)

It's also worth noting that primary and foreign keys can exist on the same column. You create a table later in this chapter that has a column that is a primary key and a foreign key.

After you have successfully run the preceding code, run `sp_help` and you should see your new constraint reported under the constraints section of the `sp_help` information. If you want to get even more to the point, you can run `sp_helpconstraint`; the syntax is easy:

```
EXEC sp_helpconstraint <table name>
```

Run `sp_helpconstraint` on your new `Orders` table, and you'll get information giving you the names, criteria, and status for all the constraints on the table. At this point, your `Orders` table has one `FOREIGN KEY` constraint and one `PRIMARY KEY` constraint.

When you run `sp_helpconstraint` on this table, the word (clustered) will appear right after the reporting of the `PRIMARY KEY`. This just means it has a clustered index. You explore the meaning of this further in Chapter 8.

Your new foreign key was referenced in the physical definition of your table and is now an integral part of your table. As discussed in Chapter 1, the database is in charge of its own integrity. Your foreign key enforces one constraint on your data and makes sure your database integrity remains intact.

Unlike primary keys, you aren't limited to one foreign key on a table. You can have between 0 and 253 foreign keys in each table. The only limitation is that a given column can reference only one foreign key. However, you can have more than one column participate in a single foreign key. A given column that is the target of a reference by a foreign key can also be referenced by many tables.

Adding a Foreign Key to an Existing Table

As with primary keys, or any constraint for that matter, sometimes you want to add your foreign key to a table that already exists. This process is similar to creating a primary key.

Try adding another foreign key to your `Orders` table to restrict the `EmployeeID` field, which is intended to have the ID of the employee who entered the order, to valid employees as defined in the `Employees` table. To do this, you need to be able to uniquely identify a target record in the referenced table. As I've already mentioned, you can do this by referencing a primary key or a column with a `UNIQUE` constraint. In this case, you use the existing primary key that you placed on the `Employees` table earlier in the chapter:

```
ALTER TABLE Orders
    ADD CONSTRAINT FK_EmployeeCreatesOrder
        FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID)
```

Now execute `sp_helpconstraint` against the `Orders` table and you'll see that your new constraint has been added.

The latest constraint works just as the last one did: The physical table definition is aware of the rules placed on the data it is to contain. Just as it would not allow string data to be inserted into a numeric column, it will also not allow a row to be inserted into the `Orders` table where the referenced employee in charge of that order isn't a valid `EmployeeID`. If someone attempts to add a row that doesn't match with an employee record, the insertion into `Orders` will be rejected to maintain the integrity of the database.

Note that although you added two foreign keys, there is still a line down at the bottom of the `sp_helpconstraint` results (or under the messages tab if you have Results in Grid selected) that says "No foreign keys reference this table." This tells you that, while you do have foreign keys in this table that reference other tables, there are no other tables out there that reference this table. If you want to see the difference, just run `sp_helpconstraint` on the `Customers` or `Employees` tables at this point and you'll see that each of these tables is referenced by your new `Orders` table.

Making a Table Self-Referencing

What if the column you want to refer to isn't in another table but is actually right within the table in which you are building the reference? Can a table be both the referencing and the referenced table? You bet! Indeed, while this is far from the most common of situations, it's actually used with regularity.

Before you actually create this self-referencing constraint that references a required (non-nullable) field that's based on an identity column, it's critical that you put at least one row in the table prior to adding the foreign key. Why? Well, the problem stems from the fact that the identity value is chosen and filled in after the foreign key has already been checked and enforced; that means that you don't have a value yet for that first row to reference when the check happens. The only other option is to create the foreign key but then disable it when adding the first row. You learn about disabling constraints a little later in this chapter.

Reviewing Keys and Constraints

Because this is a table that's referencing a column based on an identity column, you need to get a primer row into the table before you add your constraint:

```
INSERT INTO Employees
(
    FirstName,
    LastName,
    Title,
    SSN,
    Salary,
    PriorSalary,
    HireDate,
    ManagerEmpID,
    Department
)
VALUES
(
    'Billy Bob',
    'Boson',
    'Head Cook & Bottle Washer',
    '123-45-6789',
    100000,
    80000,
    '1990-01-01',
    1,
    'Cooking and Bottling'
)
```

Now that you have a primer row in, you can add your foreign key. In an ALTER situation, this works the same as any other foreign key definition. You can now try this out:

```
ALTER TABLE Employees
ADD CONSTRAINT FK_EmployeeHasManager
FOREIGN KEY (ManagerEmpID) REFERENCES Employees(EmployeeID)
```

There is one difference with a CREATE statement. The only trick to it is that you can (but you don't have to) leave out the FOREIGN KEY phrasing and just use the REFERENCES clause. You already have your Employees table set up at this point, but if you were creating it from scratch, you would use the following the script. (Pay particular attention to the foreign key on the ManagerEmpID column.)

```
CREATE TABLE Employees (
    EmployeeID      int      IDENTITY   NOT NULL
    PRIMARY KEY,
    FirstName       varchar (25)    NOT NULL,
    MiddleInitial   char (1)      NULL,
    LastName        varchar (25)    NOT NULL,
    Title           varchar (25)    NOT NULL,
    SSN             varchar (11)    NOT NULL,
    Salary          money        NOT NULL,
    PriorSalary     money        NOT NULL,
    LastRaise AS Salary - PriorSalary,
    HireDate        smalldatetime NOT NULL,
    TerminationDate smalldatetime NULL,
```

```
ManagerEmpID      int          NOT NULL  
    REFERENCES Employees(EmployeeID),  
Department       varchar (25)   NOT NULL  
)
```

It's worth noting that, if you try to DROP the Employees table at this point (to run the second example), you get an error. Why? Well, when you established the reference in your Orders table to the Employees table, the two tables became "schema bound"—that is, the Employees table now knows that it has a dependency on it. SQL Server won't let you drop a table that is referenced by another table. You have to drop the foreign key in the Orders table before SQL Server lets you delete the Employees table (or the Customers table for that matter).

In addition, using the self-referencing foreign key in the constraint doesn't allow you to get your primer row in, so it's important that you do it this way only when the column the foreign key constraint is placed on allows NULLS. That way you can have the first row have a NULL in that column and avoid the need for a primer row.

Cascading Actions

One important difference between foreign keys and other kinds of keys is that foreign keys are bidirectional, that is, they have effects not only in restricting the child table to values that exist in the parent, but they also check for child rows whenever you do something to the parent. (By doing so, they avoid orphans—child records that no longer have a parent to reference.) The default behavior is for SQL Server to "restrict" the parent row from being deleted if any child rows exist. Sometimes, however, you would rather automatically delete any dependent records than prevent the deletion of the referenced record. The same notion applies to updates to records where you want the dependent record to reference automatically the newly updated record. Somewhat rarer is the instance where you want to alter the referencing row to some sort of known state. For this, you have the option to set the value in the dependent row to NULL or to the default value for that column.

The process of making such automatic deletions and updates is known as *cascading*. This process, especially for deletes, can actually run through several layers of dependencies, where one record depends on another, which depends on another, and so on. So, how do you implement cascading actions in SQL Server? All you need is a modification to the syntax you use when declaring your foreign key; you add the ON clause that you skipped at the beginning of this section.

Check this out by adding a new table to your Accounting database. You make this a table to store the individual line items in an order, and you call it OrderDetails:

```
CREATE TABLE OrderDetails  
(  
    OrderID      int          NOT NULL,  
    PartNo       varchar(10)   NOT NULL,  
    Description  varchar(25)   NOT NULL,  
    UnitPrice    money        NOT NULL,  
    Qty          int          NOT NULL,  
    CONSTRAINT   PKOrderDetails  
        PRIMARY KEY    (OrderID, PartNo),
```

```
CONSTRAINT FKOrderContainsDetails
    FOREIGN KEY (OrderID)
        REFERENCES Orders(OrderID)
        ON UPDATE NO ACTION
        ON DELETE CASCADE
)
```

This time a whole lot is going on, so let's take it apart piece by piece.

Before you get too far into looking at the foreign key aspects of this, notice something about how the primary key was created here. Instead of placing the declaration immediately after the key, I decided to declare it as a separate constraint item. This facilitates the multicolumn primary key, which couldn't be declared as a column constraint, and the clarity of the overall CREATE TABLE statement. Likewise, I could have declared the foreign key immediately following the column or, as I did here, as a separate constraint item. I'll touch on this a little bit later in the chapter.

First, notice that the foreign key is also part of the primary key. This isn't at all uncommon in child tables and is actually almost always the case for associate tables (more on this in Chapter 7). Just remember that each constraint stands alone; you add, change, or delete each of them independently.

Next, look at your foreign key declaration:

```
FOREIGN KEY (OrderID)
    REFERENCES Orders(OrderID)
```

You declared `OrderID` as dependent on a "foreign" column. In this case, it's for a column (also called `OrderID`) in a separate table (`Orders`), but, as you saw earlier in the chapter, it could just as easily have been in the same table if that's what you needed.

There is something of a gotcha when creating foreign keys that reference the same table the foreign key is being defined on. Foreign keys of this nature are not allowed to have declarative CASCADE actions. The reason for this restriction is to avoid cyclical updates or deletes—that is, situations where the first update causes another, which in turn tries to update the first. The result could be a neverending loop.

To get to the heart of the cascading issue though, you need to look at the ON clauses:

```
ON UPDATE NO ACTION
ON DELETE CASCADE
```

You defined two different *referential integrity actions*. As you might guess, a *referential integrity action* is what you want to have happen whenever the referential integrity rule you've defined is invoked. When the parent record (in the `Orders` table) is updated, you've said that you don't want that update to be cascaded to the child table (`OrderDetails`). For illustration purposes, however, I've chosen a CASCADE for deletes.

Chapter 5

Note that NO ACTION is the default, and specifying this in your code is optional. The fact that this keyword wasn't supported until SQL Server 2000 caused the "typical" way of coding this to be to leave out the NO ACTION keywords. If you don't need backward support, I encourage you to include NO ACTION explicitly to make your intent clear.

Try an insert into your OrderDetails table:

```
INSERT INTO OrderDetails  
VALUES  
(1, '4X4525', 'This is a part', 25.00, 2)
```

Unless you've been playing around with your data some, this generates an error:

```
Msg 547, Level 16, State 0, Line 1  
The INSERT statement conflicted with the FOREIGN KEY constraint  
"FKOrderContainsDetails". The conflict occurred in database "Accounting", table  
"Orders", column 'OrderID'.  
The statement has been terminated.
```

Why? Well, you haven't inserted anything into your Orders table yet, so how can you refer to a record in the Orders table if there isn't anything there?

This is going to expose you to one of the hassles of relational database work—dependency chains. A dependency chain is where you have something that is, in turn, dependent on something else, which may yet be dependent on something else, and so on. There's really nothing you can do about this; it's just something that comes with database work. You have to start at the top of the chain and work your way down to what you need inserted. Fortunately, the records you need are often already there except for one or two dependency levels.

To get your row into your OrderDetails table, you must also have a record already in the Orders table. Unfortunately, getting a row into the Orders table requires that you have one in the Customers table. Remember that foreign key you built on Orders? So, take care of it a step at a time:

```
INSERT INTO Customers -- Our Customer.  
-- Remember that CustomerNo is  
-- an Identity column  
  
VALUES  
( 'Billy Bob''s Shoes',  
  '123 Main St.',  
  ' ',  
  'Vancouver',  
  'WA',  
  '98685',  
  'Billy Bob',  
  '(360) 555-1234',  
  '931234567',  
  GETDATE()  
)
```

Reviewing Keys and Constraints

You have a `CustomerID` of 1. Your number may be different depending on what experimentation you've done. You take that number and use it in your next `INSERT` (into `Orders` finally). Now insert an order for `CustomerID` 1:

```
INSERT INTO Orders
    (CustomerNo, OrderDate, EmployeeID)
VALUES
    (1, GETDATE(), 1)
```

This time, things should work fine.

The reason that you don't still get one more error here is that you already inserted that primer row in the `Employees` table; otherwise, you would need to get a row into that table before SQL Server would have allowed the insert into `Orders`. (Remember that `Employees` foreign key?)

At this point, you're ready for the insert into the `OrderDetails` table. Just to help with a `CASCADE` example you're going to be doing in the moment, you're actually going to insert not one, but two rows:

```
INSERT INTO OrderDetails
VALUES
    (1, '4X4525', 'This is a part', 25.00, 2)

INSERT INTO OrderDetails
VALUES
    (1, '0R2400', 'This is another part', 50.00, 2)
```

Now that you have your data in all the way, look at the effect a `CASCADE` has on the data. Delete a row from the `Orders` table, and then see what happens in `OrderDetails`:

```
USE Accounting

-- First, let's look at the rows in both tables
SELECT *
FROM Orders

SELECT *
FROM OrderDetails

-- Now, let's delete the Order record
DELETE Orders
WHERE OrderID = 1

-- Finally, look at both sets of data again
-- and see the CASCADE effect
SELECT *
FROM Orders

SELECT *
FROM OrderDetails
```

Chapter 5

This yields some interesting results:

OrderID	CustomerNo	OrderDate	EmployeeID
1	1	2000-07-13 22:18:00	1

(1 row(s) affected)

OrderID	PartNo	Description	UnitPrice	Qty
1	0R2400	This is another part	50.0000	2
1	4X4525	This is a part	25.0000	2

(2 row(s) affected)

OrderID	CustomerNo	OrderDate	EmployeeID

(1 row(s) affected)

OrderID	PartNo	Description	UnitPrice	Qty

(0 row(s) affected)

OrderID	PartNo	Description	UnitPrice	Qty

(0 row(s) affected)

Notice that even though you issued a `DELETE` only against the `Orders` table, the `DELETE` also cascaded to your matching records in the `OrderDetails` table. Records in both tables were deleted. If you had defined your table with a `CASCADE` update and updated a relevant record, then that too would have been propagated to the child table.

It's worth noting that there is no limit to the depth that a CASCADE action can affect. For example, if you had a `ShipmentDetails` table that referenced rows in `OrderDetails` with a CASCADE action, then those too would have been deleted just by your one `DELETE` in the `Orders` table.

This is actually one of the danger areas of cascading actions; it's very, very easy to overlook all the different things that one DELETE or UPDATE statement may do in your database. For this and other reasons, I'm not a huge fan of cascading actions. They allow people to get lazy, and that's something that's not usually a good thing when doing something like deleting data!

Those Other CASCADE Actions

So, those were examples of cascading updates and deletes, but what about the other two types of cascade actions I mentioned? What of `SET NULL` and `SET DEFAULT`?

These are new with SQL Server 2005, so avoid them if you want backward compatibility with SQL Server 2000, but their operation is very simple: If you perform an update that changes the parent values for a row, the child row is set to NULL or whatever the default value for that column is (whichever you chose—`SET NULL` or `SET DEFAULT`). It's just that simple.

What Makes Values in Foreign Keys Required versus Optional

By the nature of a foreign key itself, you have two choices on what to fill into a column or columns that have a foreign key defined for them:

- Fill the column with a value that matches the corresponding column in the referenced table
- Don't fill in a value at all and leave the value `NULL`

You can make the foreign key completely required (limit your users to just the first option) by simply defining the referencing column as `NOT NULL`. Because a `NULL` value won't be valid in the column and the foreign key requires any non-`NULL` value to have a match in the referenced table, you know that every row will have a match in your referenced table. In other words, the reference is required.

Allowing the referencing column to have `NULLS` will create the same requirement, except that the user will also have the option of supplying no value; even if there is not a match for `NULL` in the referenced table, the insert will still be allowed.

UNIQUE Constraints

These are relatively easy. `UNIQUE` constraints are essentially the younger siblings of primary keys in that they require a unique value throughout the named column (or combination of columns) in the table. You will often hear `UNIQUE` constraints referred to as *alternate keys*. The major differences are that they aren't considered to be *the* unique identifier of a record in that table—even though you could effectively use them that way—and you *can* have more than one `UNIQUE` constraint. (Remember that you can only have one primary key per table.)

After you establish a `UNIQUE` constraint, every value in the named columns must be unique. If you update or insert a row with a value that already exists in a column with a unique constraint, SQL Server raises an error and rejects the record.

Unlike a primary key, a `UNIQUE` constraint doesn't automatically prevent you from having a `NULL` value. Whether `NULLS` are allowed depends on how you set the `NULL` option for that column in the table. Keep in mind though that, if you do allow `NULLS`, you will only be able to insert one of them. Although a `NULL` doesn't equal another `NULL`, they are still considered to be duplicate from the perspective of a `UNIQUE` constraint.

Because there is nothing novel about this (you've pretty much already seen it with primary keys), you can get right to the code. Create yet another table in your `Accounting` database. This time, it will be your `Shippers` table:

```
CREATE TABLE Shippers
(
    ShipperID      int      IDENTITY      NOT NULL
        PRIMARY KEY,
    ShipperName    varchar(30)      NOT NULL,
```

```
Address      varchar(30)      NOT NULL,  
City         varchar(25)       NOT NULL,  
State        char(2)          NOT NULL,  
Zip          varchar(10)       NOT NULL,  
PhoneNo     varchar(14)       NOT NULL  
    UNIQUE  
)
```

Now run `sp_helpconstraint` against the `Shippers` table, and verify that your `Shippers` table has been created with the proper constraints.

Creating UNIQUE Constraints on Existing Tables

Again, this works pretty much the same as with primary and foreign keys. You create a `UNIQUE` constraint on your `Employees` table:

```
ALTER TABLE Employees  
ADD CONSTRAINT AK_EmployeeSSN  
UNIQUE (SSN)
```

A quick run of `sp_helpconstraint` verifies that your constraint was created as planned, and on what columns the constraint is active.

In case you're wondering, the AK I used in the constraint name here is for Alternate Key, much like we used PK and FK for Primary and Foreign Keys. You will also often see a UQ or just U prefix used for UNIQUE constraint names.

CHECK Constraints

The nice thing about `CHECK` constraints is that they are not restricted to a particular column. They can be related to a column, but they can also be essentially table related in that they can check one column against another as long as all the columns are within a single table, and the values are for the same row being updated or inserted. They may also check that any combination of column values meets a criterion.

The constraint is defined using the same rules that you would use in a `WHERE` clause. The following table gives examples of the criteria for a `CHECK` constraint:

Goal	SQL
Limit Month column to appropriate numbers	<code>BETWEEN 1 AND 12</code>
Proper SSN formatting	<code>LIKE '[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9][0-9]'</code>
Limit to a specific list of Shippers	<code>IN ('UPS', 'Fed Ex', 'USPS')</code>
Price must be positive	<code>UnitPrice >= 0</code>
Referencing another column in the same row	<code>ShipDate >= OrderDate</code>

This table only scratches the surface, and the possibilities are virtually endless. Almost anything you could put in a WHERE clause you can also put in your constraint. What's more, CHECK constraints are very fast performance-wise compared to the alternatives (rules and triggers).

Still building on your Accounting database, let's digress a moment and add a column to keep track of the date a customer is added to the system.

```
ALTER TABLE Customers  
ADD DateInSystem datetime
```

With that in place, we're ready to add an additional modification to your Customers table to check for a valid date in your DateInSystem field. You can't have a date in the system that's in the future:

```
ALTER TABLE Customers  
ADD CONSTRAINT CN_CustomerDateInSystem  
CHECK  
(DateInSystem <= GETDATE ())
```

Now try to insert a record that violates the CHECK constraint. You'll get an error:

```
INSERT INTO Customers  
(CustomerName, Address1, Address2, City, State, Zip, Contact,  
Phone, FedIDNo, DateInSystem)  
VALUES  
( 'Customer1', 'Address1', 'Add2', 'MyCity', 'NY', '55555',  
'No Contact', '553-1212', '930984954', '12-31-2049' )
```

```
Msg 547, Level 16, State 0, Line 1  
The INSERT statement conflicted with the CHECK constraint  
"CN_CustomerDateInSystem". The conflict occurred in database "Accounting", table  
"dbo.Customers", column 'DateInSystem'.  
The statement has been terminated.
```

Now if you change things to use a DateInSystem that meets the criterion used in the CHECK (anything with today's date or earlier), the INSERT works fine.

DEFAULT Constraints

This will be the first of two different types of data integrity tools that will be called something to do with "default." This is, unfortunately, very confusing, but I'll do my best to make it clear (and I think it will become so).

You'll see the other type of default when you look at rules and defaults later in the chapter.

A DEFAULT constraint, like all constraints, becomes an integral part of the table definition. It defines what to do when a new row is inserted that doesn't include data for the column on which you have defined the default constraint. You can define it as a literal value (for example, by setting a default salary to zero or UNKNOWN for a string column) or as one of several system values such as GETDATE().

Chapter 5

The following list points out the main things to understand about a DEFAULT constraint:

- ❑ Defaults are used only in `INSERT` statements; they are ignored for `UPDATE` and `DELETE` statements.
- ❑ If any value is supplied in the `INSERT`, the default isn't used.
- ❑ If no value is supplied, the default will always be used.

Under the heading of “One more thing,” it’s worth noting that there is an exception to the rule about an `UPDATE` command not using a default. The exception happens if you explicitly say that you want a default to be used. You do this by using the keyword `DEFAULT` as the value you want the column updated to.

Defining a `DEFAULT` Constraint in Your `CREATE TABLE` Statement

At the risk of sounding repetitious, this works pretty much like all the other column constraints you've dealt with thus far. You just add it to the end of the column definition.

To work an example, start by dropping the existing `Shippers` table that you created earlier in the chapter. This time, create a simpler version of that table including a default:

```
CREATE TABLE Shippers
(
    ShipperID      int      IDENTITY   NOT NULL
        PRIMARY KEY,
    ShipperName    varchar(30)     NOT NULL,
    DateInSystem  smalldatetime  NOT NULL
        DEFAULT GETDATE ()
)
```

After you run your `CREATE` script, you can again make use of `sp_helpconstraint` to show you what you have done. You can then test how your default works by inserting a new record:

```
INSERT INTO Shippers
    (ShipperName)
VALUES
    ('United Parcel Service')
```

Then run a `SELECT` statement on your `Shippers` table:

```
SELECT * FROM Shippers
```

The default value has been generated for the `DateInSystem` column because you didn't supply a value:

```
ShipperID      ShipperName          DateInSystem
-----          -----
1              United Parcel Service 2000-07-13 23:26:00
(1 row(s) affected)
```

Adding a *DEFAULT* Constraint to an Existing Table

While this one is still pretty much more of the same, there is a slight twist. You make use of the `ALTER` statement and `ADD` the constraint as before, but you add a `FOR` operator to tell SQL Server which column is the target for the `DEFAULT`:

```
ALTER TABLE Customers
    ADD CONSTRAINT CN_CustomerDefaultDateInSystem
        DEFAULT GETDATE() FOR DateInSystem
```

And an extra example:

```
ALTER TABLE Customers
    ADD CONSTRAINT CN_CustomerAddress
        DEFAULT 'UNKNOWN' FOR Address1
```

As with all constraints except for a `PRIMARY KEY`, you can add more than one per table.

You can mix and match any and all of these constraints as you choose. Just be careful not to create constraints that have mutually exclusive conditions. For example, don't have one constraint that says that `col1 > col2` and another one that says that `col2 > col1`. SQL Server lets you do this, and you wouldn't see the issues with it until runtime.

Disabling Constraints

Sometimes you want to eliminate the constraint checking, either just for a time or permanently. It probably doesn't take much thought to realize that SQL Server must give you some way of deleting constraints, but SQL Server also enables you to deactivate a `FOREIGN KEY` or `CHECK` constraint while otherwise leaving it intact.

The concept of turning off a data integrity rule might seem rather ludicrous at first. I mean, why would you want to turn off the thing that makes sure you don't have bad data? The usual reason is the situation where you already have bad data. This data usually falls into two categories:

- Data that's already in your database when you create the constraint
- Data that you want to add after the constraint is already built

You cannot disable PRIMARY KEY or UNIQUE constraints.

Ignoring Bad Data When You Create the Constraint

All this syntax has been just fine for use when you create the constraint at the same time as you create the table. Quite often, however, data rules are established after the fact. Say, for instance, that you missed something when you were designing your database, and you now have some records in an `Invoicing` table that show a negative invoice amount. You might want to add a rule that won't let any more negative invoice amounts into the database, but at the same time, you want to preserve the existing records in their original state.

To add a constraint but have it not apply to existing data, you make use of the `WITH NOCHECK` option when you perform the `ALTER TABLE` statement that adds your constraint. As always, take a look at an example.

The `Customers` table you created in the Accounting database has a field called `Phone`. The `Phone` field was created with a data type of `char` because you expected all of the phone numbers to be of the same length. You also set it with a length of 15 to ensure that you have enough room for all the formatting characters. However, you haven't done anything to make sure that the records inserted into the database do indeed match the formatting criteria that you expect. To test this, insert a record in a format that is not what you're expecting but might be a very honest mistake in terms of how someone might enter a number:

```
INSERT INTO Customers
(CustomerName,
Address1,
Address2,
City,
State,
Zip,
Contact,
Phone,
FedIDNo,
DateInSystem)
VALUES
('MyCust',
'123 Anywhere',
',
',
'Reno',
'NV',
80808,
'Joe Bob',
'555-1212',
'931234567',
GETDATE ())
```

Now add a constraint to control the formatting of the `Phone` field:

Reviewing Keys and Constraints

```
ALTER TABLE Customers
    ADD CONSTRAINT CN_CustomerPhoneNo
    CHECK
        (Phone LIKE '([0-9][0-9][0-9]) [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]')
```

When you run this, you have a problem:

```
Msg 547, Level 16, State 1, Line 1
ALTER TABLE statement conflicted with COLUMN CHECK constraint 'CN_CustomerPhoneNo'.
The conflict occurred in database 'Accounting', table 'Customers', column 'Phone'.
```

SQL Server won't create the constraint unless the existing data meets the constraint criteria. To get around this long enough to install the constraint, you need to correct the existing data or you must make use of the WITH NOCHECK option in your ALTER statement. To do this, you just add WITH NOCHECK to the statement as follows:

```
ALTER TABLE Customers
    WITH NOCHECK
    ADD CONSTRAINT CN_CustomerPhoneNo
    CHECK
        (Phone LIKE '([0-9][0-9][0-9]) [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]')
```

Now if you run your same INSERT statement again (remember it inserted without a problem last time), the constraint works and the data is rejected:

```
Msg 547, Level 16, State 0, Line 1
The ALTER TABLE statement conflicted with the CHECK constraint
"CN_CustomerPhoneNo". The conflict occurred in database "Accounting", table
"dbo.Customers", column 'Phone'.
```

However, if you modify your INSERT statement to adhere to your constraint and then re-execute it, the row will be inserted normally:

```
INSERT INTO Customers
    (CustomerName,
     Address1,
     Address2,
     City,
     State,
     Zip,
     Contact,
     Phone,
     FedIDNo,
     DateInSystem)
VALUES
    ('MyCust',
     '123 Anywhere',
     '',
     'Reno',
     'NV',
     80808,
     'Joe Bob',
     '(800)555-1212',
     '931234567',
     GETDATE ())
```

Chapter 5

Try running a `SELECT` on the `Customers` table at this point. You'll see data that does and data that doesn't adhere to your `CHECK` constraint criterion:

```
SELECT CustomerNo, CustomerName, Phone FROM Customers
```

CustomerNo	CustomerName	Phone
1	Billy Bob's Shoes	(360) 555-1234
2	Customer1	553-1212
3	MyCust	555-1212
5	MyCust	(800) 555-1212

(2 row(s) affected)

The old data is retained for backward reference, but any new data is restricted to meeting the new criteria.

Temporarily Disabling an Existing Constraint

Old data doesn't just come in the form of data that has already been added to your database. It may also be data that you are importing from a legacy database or some other system. Whatever the reason, the same issue still holds: you have some existing data that doesn't match with the rules, and you need to get it into the table.

Certainly one way to do this would be to drop the constraint, add the desired data, and then add the constraint using a `WITH NOCHECK`. But what a pain! Fortunately, you don't need to do that. Instead, you can run an `ALTER` statement with an option called `NOCHECK` that turns off the constraint in question. Here's the code that disables the `CHECK` constraint that you added in the previous section:

```
ALTER TABLE Customers
  NOCHECK
  CONSTRAINT CN_CustomerPhoneNo
```

Now you can run that `INSERT` statement again—the one that wouldn't work if the constraint was active:

```
INSERT INTO Customers
  (CustomerName,
   Address1,
   Address2,
   City,
   State,
   Zip,
   Contact,
   Phone,
   FedIDNo,
   DateInSystem)
```

```
VALUES
('MyCust',
'123 Anywhere',
',
'Reno',
'NV',
80808,
'Joe Bob',
'555-1212',
'931234567',
GETDATE())
```

Once again, you are able to insert nonconforming data to the table.

By now, you may be wondering how you know whether you have the constraint turned on. SQL Server provides a procedure to indicate the status of a constraint, and it's a procedure you've already seen, `sp_helpconstraint`. To execute it against your `Customers` table is easy:

```
EXEC sp_helpconstraint Customers
```

Object Name

Customers

constraint_type	constraint_name
DEFAULT on column Address1	CN_CustomerAddress
CHECK on column DateInSystem	CN_CustomerDateInSystem
DEFAULT on column DateInSystem	CN_CustomerDefaultDateInSystem
PRIMARY KEY (clustered)	PK_Customers_7E6CC920

Table is referenced by foreign key

Accounting.dbo.Orders: FK__Orders__Customer__03317E3D

I've truncated the right-hand side of the results for the sake of it fitting in this book (but at least you get a feel for what you should be seeing), but the second result set this procedure generates includes a column (scroll way to the right) called `status_enabled`. Whatever this column says the status is can be believed; in this case, it should be `Disabled`.

When you are ready for the constraint to be active again, you simply turn it on by issuing the same command with a `CHECK` in the place of the `NOCHECK`:

```
ALTER TABLE Customers
CHECK
CONSTRAINT CN_CustomerPhoneNo
```

If you run the `INSERT` statement to verify that the constraint is again functional, you will see a familiar error message:

```
Msg 547, Level 16, State 0, Line 1
The INSERT statement conflicted with the CHECK constraint
"CN_CustomerDateInSystem". The conflict occurred in database "Accounting", table
"dbo.Customers", column 'DateInSystem'.
The statement has been terminated.
```

The other option, of course, is to run `sp_helpconstraint` again, and check the `status_enabled` column. If it shows `Enabled`, your constraint must be functional again.

Rules and Defaults: Cousins of Constraints

Rules and *defaults* have been around much longer than `CHECK` and `DEFAULT` constraints have been. They are something of an old SQL Server standby and are definitely not without their advantages.

That being said, I'm going to digress from explaining them long enough to recommend that you look them over for backward compatibility and legacy code familiarity only. Rules and defaults aren't ANSI compliant, which creates portability issues, and they don't perform as well as constraints do.

The primary thing that sets rules and defaults apart from constraints is in their very nature; constraints are features of a table. Rules and defaults are actual objects. Whereas a constraint is defined in the table definition, rules and defaults are defined independently and are then "bound" to the table after the fact.

Rules

A rule is incredibly similar to a `CHECK` constraint. The only difference beyond those I've already described is that rules are limited to working with just one column at a time. You can bind the same rule separately to multiple columns in a table, but the rule will work independently with each column, and won't be aware of the other columns at all. A constraint defined as `(QtyShipped <= QtyOrdered)` would not work for a rule. It refers to more than one column. In contrast, `LIKE ([0-9][0-9][0-9])` applies only to the column to which the rule is bound.

Try defining a rule so that you can see the differences first hand:

```
CREATE RULE SalaryRule
AS @Salary > 0
```

Notice that what you are comparing is shown as a variable: the value is that of the column being checked and is the value used in the place of `@Salary`. Thus, in this example, you're saying that any column your rule is bound to must have a value greater than zero.

If you want to go back and see what your rule looks like, you can make use of `sp_helptext`:

```
EXEC sp_helptext SalaryRule
```

It will show you your exact rule definition:

Text

```
-----  
CREATE RULE SalaryRule  
    AS @Salary > 0
```

Now you have a rule, but it isn't doing anything. If you tried to insert a record in your Employees table, you could still insert any value right now without any restrictions beyond data type.

To activate the rule, you use a special stored procedure called `sp_bindrule`. You want to bind your `SalaryRule` to the `Salary` column of your `Employees` table. The syntax looks like this:

```
sp_bindrule <'rule'>, <'object_name'>, [<'futureonly_flag'>]
```

The `rule` part is simple enough: that's the rule you want to bind. The `object_name` is also simple enough. It's the object (column or user-defined data type) to which you want to bind the rule. The only odd parameter is the `futureonly_flag`, and it applies only when the rule is bound to a user-defined data type. The default is for this to be off. However, if you set it to `True` or pass in a `1`, the binding of the rule will apply to only new columns to which you bind the user-defined data type. Any columns that already have the data type in its old form will continue to use that form.

Because you're just binding this rule to a column, your syntax requires only the first two parameters:

```
sp_bindrule 'SalaryRule', 'Employees.Salary'
```

Take a close look at the `object_name` parameter. `Employees` and `Salary` are separated by a `"."`. Why is that? Because the rule isn't associated with any particular table until you bind it, you need to state the table and column to which the rule will be bound. If you don't use the `tablename.column` naming structure, SQL Server assumes that what you're naming must be a user-defined data type. If it doesn't find one, you'll get an error message that can be a bit confusing if you hadn't intended to bind the rule to a data type:

```
Msg 15148, Level 16, State 1, Procedure sp_bindrule, Line 190  
The data type or table column 'Salary' does not exist or you do not have  
permission.
```

In this case, trying to insert or update an `Employees` record with a negative value violates the rule and generates an error.

If you want to remove your rule from use with this column, use `sp_unbindrule`:

```
EXEC sp_unbindrule 'Employees.Salary'
```

The `futureonly_flag` parameter is again an option but doesn't apply to this particular example. If you use `sp_unbindrule` with the `futureonly_flag` turned on, and it's used against a user-defined data type (rather than a specific column), the unbinding will apply only to future uses of that data type; existing columns using that data type will still make use of the rule.

Dropping Rules

If you want to completely eliminate a rule from your database, use the same `DROP` syntax that you are already familiar with for tables:

```
DROP RULE <rule name>
```

Defaults

Defaults are even more similar to their cousin — a default constraint — than a rule is to a `CHECK` constraint. Indeed, they work identically, with the only real differences being the way that they are attached to a table and the default's (the object, not the constraint) support for a user-defined data type.

The concept of defaults versus `DEFAULT` constraints is wildly difficult for a lot of people to grasp. After all, they have almost the same name. “Default” refers to the object-based default (what I’m talking about in this section), or as shorthand to the actual default value (that will be supplied if you don’t provide an explicit value). A “`DEFAULT constraint`” refers to the nonobject-based solution, the solution that is an integral part of the table definition.

The syntax for defining a default works much as it did for a rule:

```
CREATE DEFAULT <default name>
AS <default value>
```

Therefore, use the following code to define a default of zero for `Salary`:

```
CREATE DEFAULT SalaryDefault
AS 0
```

Again, a default is worthless without being bound to something. To bind it you use `sp_bindefault`, which is, other than the procedure name, identical syntax to the `sp_bindrule` procedure:

```
EXEC sp_bindefault 'SalaryDefault', 'Employees.Salary'
```

To unbind the default from the table, use `sp_unbindefault`:

```
EXEC sp_unbindefault 'Employees.Salary'
```

Keep in mind that the `futureonly_flag` also applies to this stored procedure; it is just not used here.

Dropping Defaults

If you want to completely eliminate a default from your database, use the same `DROP` syntax that you’re already familiar with for tables and rules:

```
DROP DEFAULT <default name>
```

Determining Which Tables and Data Types Use a Given Rule or Default

If you ever go to delete or alter your rules or defaults, you may first want to take a look at which tables and data types are using them. Again, SQL Server comes to the rescue with a system stored procedure. This one is called `sp_depends`. Its syntax looks like this:

```
EXEC sp_depends <object name>
```

`sp_depends` provides a listing of all the objects that depend on the object you've requested information about.

Unfortunately, `sp_depends` is not a sure bet to tell you about every object that depends on a parent object. SQL Server supports a deferred name resolution. Basically, *deferred name resolution* means that you can create objects (primary stored procedures) that depend on another object even before the second (target of the dependency) object is created. For example, SQL Server now enables you to create a stored procedure that refers to a table even before the said table is created. In this instance, SQL Server isn't able to list the table as having a dependency on it. Even after you add the table, it won't have any dependency listing if you use `sp_depends`.

Triggers for Data Integrity

Although there's a whole chapter coming up on triggers (Chapter 13), any discussion of constraints, rules, and defaults would not be complete without at least a mention of triggers.

One of the most common uses of triggers is to implement data integrity rules. Because you have that chapter coming up, I'm not going to get into it very deep here other than to say that triggers can do many things data-integrity-wise that a constraint or rule could never hope to do. The downside (and you knew there had to be one) is that they incur substantial additional overhead and are, therefore, much (very much) slower in almost any circumstance. They are procedural in nature, which is where they get their power, but they also happen after everything else is done, and should be used only as a last resort.

Choosing What to Use

Wow. Here you are with all these choices. How do you figure out which is the right one to use? Some of the constraints are fairly independent (`PRIMARY` and `FOREIGN KEYS`, `UNIQUE` constraints). You are using either them or nothing. The rest have some level of overlap with each other and it can be confusing to decide which to use. You got some hints from me while going through this chapter about what some of the strengths and weaknesses are of each of the options, but it will probably make a lot more sense if you look at them all together in the following table.

Chapter 5

Restriction	Pros	Cons
Constraints	Fast.	Must be redefined for each table.
	Can reference other columns.	Can't reference other tables.
	Happens before the command occurs.	Can't be bound to data types.
	ANSI compliant.	
Rules, Defaults	Independent objects.	Slightly slower.
	Reusable.	Can't reference across columns.
	Can be bound to data types.	Can't reference other tables.
	Happens before the command occurs.	Really meant only for backward compatibility.
Triggers	Ultimate flexibility.	Happens after the command occurs.
	Can reference other columns and other tables.	High overhead.
	Can even use .NET to reference information that is external to your SQL Server.	

The main time to use rules and defaults is when you are implementing a rather robust logical model and are making extensive use of user-defined data types. In this instance, rules and defaults can provide a lot of functionality and ease of management without much programmatic overhead. You just need to be aware that they may go away in a future release someday. Probably not soon, but someday.

Triggers should be used only when a constraint is not an option. Like constraints, they are attached to the table and must be redefined with every table you create. On the bright side, they can do most things that you are likely to want to do data-integrity-wise. Indeed, they used to be the common method of enforcing foreign keys (before FOREIGN KEY constraints were added).

That leaves constraints, which should become your data integrity solution of choice. They are fast and not that difficult to create. Their downfall is that they can be limiting (not being able to reference other tables except for a FOREIGN KEY), and they can be tedious to redefine repeatedly if you have a common constraint logic.

Regardless of what kind of integrity mechanism you're putting in place (keys, triggers, constraints, or rules, defaults), the thing to remember can best be summed up in just one word: balance.

Every new thing that you add to your database adds additional overhead, so you need to make sure that whatever you're adding honestly has value. Avoid things like redundant integrity implementations. (For example, I can't tell you how often I've come across a database that has both foreign keys defined for referential integrity and triggers to do the same thing.) Make sure you know which constraints you have before you add the next one, and make sure you know exactly what you hope to accomplish with it.

Summary

The different types of data integrity mechanisms described in this chapter are part of the backbone of a sound database. Perhaps the biggest power of RDBMSs is that the database can now take responsibility for data integrity rather than depending on the application. This means that even ad hoc queries are subject to the data rules, and that multiple applications are all treated equally with regard to data integrity issues.

In the chapters to come, you look at the tie between some forms of constraints and indexes, along with the advanced data integrity rules than can be implemented using triggers. You also begin looking at how the choices between these different mechanisms affect your design decisions.

6

Asking a Better Question: Advanced Queries

It was a tough decision. Advanced query design before cursors, or cursors before advanced query design? You see, it's something of a chicken and egg thing (which came first?). Not that you need to know anything about cursors to make use of the topics covered in this chapter, but rather because we'll be discussing some benefits of different query methods that avoid cursors—and it really helps to understand the benefits if you know what you're trying to avoid.

That said, I went for the advanced queries first notion. In the end, I figured I wanted to try to get you thinking about non-cursor-based queries as much as possible before we start talking cursors. Since I figure that a large percentage of the readers of this book will already have experience in some programming language, I know that you're going to have a natural tendency to think of things in a procedural fashion rather than a "set" fashion. Since cursors are a procedural approach, the odds are you're going to think of the more complex problems in terms of cursors first rather than how you could do it in a single query.

Suffice to say that I want to challenge you in this chapter. Even if you don't have that much procedural programming experience, the fact is that your brain has a natural tendency to break complex problems down into their smaller subparts (subprocedures—logical steps) as opposed to as a whole (the "set," or SQL way). My challenge to you is to try and see the question as its whole first. Be certain that you can't get it in a single query. Even if you can't think of a way, quite often you can break it up into several small queries and then combine them one at a time back into a larger query that does it all in one task. Try to see it as a whole, and, if you can't, then go ahead and break it down, but then recompile it into the whole again to the largest extent possible.

Chapter 6

Several of the topics in this chapter represent, to me, significant marks of the difference between a “beginner” and “professional” when it comes to SQL Server programming. While they are certainly not the only thing that marks when you are a true “pro,” developers that can move from “Yeah, I know those exist and use one or two of them” to using them to make unsolvable queries solvable are true gold. I write on these subjects for beginners to let them know they are there and give them a taste of what they can do. I write on them for professionals because full understanding of these concepts is critical to high-level success with SQL Server (or almost any major DBMS for that matter).

In this chapter, we’re going to be looking at ways to ask what amounts to multiple questions in just one query. Essentially, we’re going to look at ways of taking what seems like multiple queries and place them into something that will execute as a complete unit. Writing top-notch queries isn’t just about trickiness or being able to make them complex—it’s making them perform. With that in mind, we’ll also be taking a look at query performance, and what we can do to get the most out of our queries.

Among the topics we’ll be covering in this chapter are:

- Nested subqueries
- Correlated subqueries
- Derived tables
- Making use of the `EXISTS` operator
- Using external calls to perform complex actions
- Optimizing query performance

We’ll see how by using subqueries we can make the seemingly impossible completely possible, and how an odd tweak here and there can make a big difference in our query performance.

What Is a Subquery?

A *subquery* is a normal T-SQL query that is nested inside another query—using parentheses—created when you have a `SELECT` statement that serves as the basis for either part of the data or the condition in another query.

Subqueries are generally used to fill one of several needs:

- Break a query up into a series of logical steps
- Provide a listing to be the target of a `WHERE` clause together with [`IN | EXISTS | ANY | ALL`]
- To provide a lookup driven by each individual record in a parent query

Some subqueries are very easy to think of and build, but some are extremely complex—it usually depends on the complexity of the relationship between the inner (the sub) and outer (the top) query.

It’s also worth noting that most subqueries (but definitely not all) can also be written using a join. In places where you can use a join instead, the join is usually the preferable choice.

I once got into a rather lengthy (perhaps 20 or 30 e-mails flying back and forth with examples, reasons, etc. over a few days) debate with a coworker over the joins versus subqueries issue.

Traditional logic says to always use the join, and that was what I was pushing (due to experience rather than traditional logic—you've already seen several places in this book where I've pointed out how traditional thinking can be bogus). My coworker was pushing the notion that a subquery would actually cause less overhead—I decided to try it out.

What I found was essentially (as you might expect) that you were both right in certain circumstances. We will explore these circumstances fully toward the end of the chapter after you have a bit more background.

Now that you know what a subquery theoretically is, take a look at some specific types and examples of subqueries.

Building a Nested Subquery

A *nested subquery* is one that goes in only *one* direction—returning either a single value for use in the outer query, or perhaps a list of values to be used with the `IN` operator. In the event you want to use an explicit `=` operator, then you're going to be using a query that returns a single value—that means one column from one row. If you are expecting a list back, then you'll need to use the `IN` operator with your outer query.

In the loosest sense, your query syntax is going to look something like one of these two syntax templates:

```
SELECT <SELECT list>
FROM <SomeTable>
WHERE <SomeColumn> = (
    SELECT <single column>
    FROM <SomeTable>
    WHERE <condition that results in only one row returned>)
```

Or:

```
SELECT <SELECT list>
FROM <SomeTable>
WHERE <SomeColumn> IN (
    SELECT <single column>
    FROM <SomeTable>
    [WHERE <condition>])
```

Obviously, the exact syntax will vary. Not for just substituting the select list and exact table names, but also because you may have a multi-table join in either the inner or outer queries—or both.

Nested Queries Using Single Value **SELECT** Statements

Let's get down to the nitty-gritty with an explicit example. Let's say, for example, that we wanted to know the ProductIDs of every item sold on the first day any product was purchased from the system.

If you already know the first day that an order was placed in the system, then it's no problem; the query would look something like this:

```
USE AdventureWorks

SELECT DISTINCT soh.OrderDate, sod.ProductID
FROM Sales.SalesOrderHeader soh
JOIN Sales.SalesOrderDetail sod
    ON soh.SalesOrderID = sod.SalesOrderID
WHERE OrderDate = '07/01/2001' --This is first OrderDate in the system
```

This yields us the correct results:

OrderDate	ProductID
2001-07-01 00:00:00.000	707
2001-07-01 00:00:00.000	708
2001-07-01 00:00:00.000	709
...	
...	
...	
2001-07-01 00:00:00.000	776
2001-07-01 00:00:00.000	777
2001-07-01 00:00:00.000	778

(47 row(s) affected)

But let's say, just for instance, that we are regularly purging data from the system, and we still want to ask this same question as part of an automated report. Since it's going to be automated, we can't run a query to find out what the first date in the system is and manually plug that into our query.

One way to do this would be to add a variable (we will discuss variable use in Chapter 11) and make it part of a batch:

```
DECLARE @FirstDate smalldatetime

SELECT @FirstDate = MIN(OrderDate) FROM Sales.SalesOrderHeader

SELECT DISTINCT soh.OrderDate, sod.ProductID
FROM Sales.SalesOrderHeader soh
JOIN Sales.SalesOrderDetail sod
    ON soh.SalesOrderID = sod.SalesOrderID
WHERE soh.OrderDate = @FirstDate
```

While this works (you should get back the exact same results), we can actually clean things up a bit by putting it all into one statement:

Asking a Better Question: Advanced Queries

```
SELECT DISTINCT o.OrderDate, od.ProductID
FROM Sales.SalesOrderHeader soh
JOIN Sales.SalesOrderDetail sod
    ON soh.SalesOrderID = sod.SalesOrderID
WHERE soh.OrderDate = (SELECT MIN(OrderDate) FROM Sales.SalesOrderHeader)
```

It's just that quick and easy. The inner query (`SELECT MIN...`) retrieves a single value for use in the outer query. Since we're using an equals sign, the inner query absolutely must return only one column from one single row, or we will get a runtime error.

Nested Queries Using Subqueries That Return Multiple Values

Perhaps the most common of all subqueries that are implemented in the world are those that retrieve some form of domain list and use it as a criterion for a query. What we want is a list of all the employees that have applied for another job within the company. We keep our applicants listed in a table called `HumanResources.JobCandidate`, so what we need is a list of `EmployeeIDs` that have a record in the job candidate table. The actual list of all employees is, of course, in our `HumanResources.Employee` table. We will also have to use our `Person.Contact` table to get things like the employee's name.

We might write something like this:

```
USE AdventureWorks

SELECT e.EmployeeID, FirstName, LastName
FROM HumanResources.Employee e
JOIN Person.Contact c
    ON e.ContactID = c.ContactID
WHERE e.EmployeeID IN
    (SELECT DISTINCT EmployeeID FROM HumanResources.JobCandidate)
```

This gets us back just two rows:

EmployeeID	FirstName	LastName
41	Peng	Wu
268	Stephen	Jiang

(2 row(s) affected)

Queries of this type almost always fall into the category of one that can be done using an inner join rather than a nested `SELECT`. For example, we could get the same results as the preceding subquery by running this simple join:

```
USE AdventureWorks

SELECT e.EmployeeID, FirstName, LastName
FROM HumanResources.Employee e
JOIN Person.Contact c
    ON e.ContactID = c.ContactID
JOIN HumanResources.JobCandidate jc
    ON e.EmployeeID = jc.EmployeeID
```

Chapter 6

For performance reasons, you want to use the join method as your default solution if you don't have a specific reason for using the nested SELECT—we'll discuss this more before the chapter is done.

Using a Nested **SELECT** to Find Orphaned Records

This type of nested SELECT is nearly identical to the previous example, except that we add the NOT operator. The difference this makes when you are converting to join syntax is that you are equating to an outer join rather than an inner join.

This is for the scenario where you want to see what's left out. In many cases, this might be something like order details that don't have a parent record in the header table (this can't happen in the AdventureWorks database thanks to your foreign key constraint, but there are those databases out there where this kind of thing happens). For our example, we'll change the scenario around to ask what employee's have *not* applied for a different job in the company. See that "not" in there and you know just what to do—add the NOT to our query (but beware a special case issue here that we have to deal with):

```
USE AdventureWorks

SELECT e.EmployeeID, FirstName, LastName
FROM HumanResources.Employee e
JOIN Person.Contact c
    ON e.ContactID = c.ContactID
WHERE e.EmployeeID NOT IN
    (SELECT DISTINCT EmployeeID
     FROM HumanResources.JobCandidate
     WHERE EmployeeID IS NOT NULL)
```

Run this, and, of course, you get a large result set (every employee but the two we saw in the previous example).

As always, beware tests against sets that might contain a `NULL` value. Comparisons against `NULL` always result in `NULL`. In the preceding case, the `JobCandidate` table has rows where the `EmployeeID` is null. If I had allowed `NULL` to come back in my subquery, then every row in the outer query would have evaluated false when compared to `NOT IN`—I would get an empty list back (I recommend experimenting with it to make sure you understand the distinction).

The **ANY**, **SOME**, and **ALL** Operators

Up to this point, we've been looking only for an item that was `IN` a list—that is, where at least one value in the list was an exact match to what we were looking for. Ahh, but life isn't that simple. What if we want to do something other than an exact match—something that isn't quite equivalent to an equals (`=`) sign? What if we want a more complete list of operators? No problem!

But "Wait, hold on!" you say. "What if I want to see if *every* value in a list matches something—what then?" Again—no problem!

ANY and SOME

ANY and SOME are functional equivalents; if you choose to use one of them (you'll see later why I don't recommend them), then I would suggest using SOME, as it is the ANSI-compliant one of the two. In all honesty, for the rest of this section, I'm only going to use SOME, but feel free to substitute ANY in any place—you should get the same results.

The ANY and SOME operators allow you to use a more broad range of other operators against the lists created by your subqueries. They can also be used with any of the other operators you would typically use for comparisons (such as \geq , \leq , $<$, $>$, etc.).

Taking $>$ as an example, $>\text{SOME}$ means greater than any one value, that is, greater than the minimum. So, $>\text{SOME} (1, 2, 3)$ means greater than 1. If you use them with an equals sign ($=$), then they are a functional equivalent of the IN operator.

Actually, I'm not a big fan of this keyword. The reason is that I find it functionally useless. The things you can do with SOME (or ANY) fall into two categories. The first of these is that there isn't anything you can do with SOME that you can't do with some other syntax (the other syntax has always been more succinct in my experience). The only exception to this is the case of $\text{<>} \text{ SOME}$. Where the NOT IN (A, B, C) clause gets you a logical expansion of $\text{<>} A \text{ AND } \text{<>} B \text{ AND } \text{<>} C$, $\text{<>} \text{ SOME}$ gets you $\text{<>} A \text{ OR } \text{<>} B \text{ OR } \text{<>} C$. This last option is a positively useless construct. Think about it for a minute—any comparison you run against $\text{<>} \text{ ANY}$ is going to yield you a non-filtered list. By definition, anything that is = A is $\text{<>} B$.

In short, if you find a reason to use this, great—I'd love to hear about it—but, to date, I haven't seen anything done with SOME that can't be done more clearly and with better performance using a different construct (which one depends on the nature of the SOME you're trying to match).

ALL

The ALL operator is similar to the SOME and ANY operators in allowing you to work with a broader range of comparison operators. However, applying our previous $>$ example to the ALL statement, we see the difference. $>\text{ALL}$ means greater than every value, that is, greater than the maximum. So, $>\text{ALL} (1, 2, 3)$ means greater than 3.

Correlated Subqueries

Two words for you on this section: Pay attention! This is another one of those little areas that, if you truly "get it," can really set you apart from the crowd. By "get it" I don't just mean that you understand how it works but also that you understand how important it can be.

Correlated subqueries are one of those things that make the impossible possible. What's more, they often turn several lines of code into one, and often create a corresponding increase in performance. The problem with them is that they require a substantially different style of thought than you're probably used to. Correlated subqueries are probably the single easiest concept in SQL to learn, understand, and then promptly forget because it simply goes against the grain of how you think. If you're one of the few who choose to remember it as an option, then you will be one of the few who figure out that hard to figure out problem. You'll also be someone with a far more complete toolset when it comes to squeezing every ounce of performance out of your queries.

How Correlated Subqueries Work

What makes correlated subqueries different from the nested subqueries we've been looking at is that the information travels in *two* directions rather than one. In a nested subquery, the inner query is only processed once, and that information is passed out for the outer query, which will also execute just once—essentially providing the same value or list that you would have provided if you had typed it in yourself.

With correlated subqueries, however, the inner query runs on information provided by the outer query, and vice versa. That may seem a bit confusing (that chicken or the egg thing again), but it works in a three-step process:

1. The outer query obtains a record and passes it into the inner query.
2. The inner query executes based on the passed in value(s).
3. The inner query then passes the values from its results back out to the outer query, which uses them to finish its processing.

Correlated Subqueries in the WHERE Clause

I realize that this is probably a bit confusing, so take a look at it in an example.

Let's look again at the query where we wanted to know the orders that happened on the first date that an order was placed in the system. However, this time we want to add a new twist: We want to know the OrderID(s) and OrderDate of the first order in the system for each customer. That is, we want to know the first day that a customer placed an order and the IDs of those orders. Let's look at it piece by piece.

First, we want the OrderDate, OrderID, and CustomerID for each of our results. All of that information can be found in the SalesOrderHeader table, so we know that your query is going to be based, at least in part, on that table.

Next, we need to know what the first date in the system was for each customer. That's where the tricky part comes in. When we did this with a nested subquery, we were looking only for the first date in the entire file—now we need a value that's by individual customer.

This wouldn't be that big a deal if we were to do it in two separate queries—we could just create a temporary table, and then join back to it—like this:

```
USE AdventureWorks

SELECT CustomerID, MIN((OrderDate)) AS OrderDate
INTO #MinOrderDates
FROM Sales.SalesOrderHeader
GROUP BY CustomerID
ORDER BY CustomerID

SELECT o.CustomerID, o.SalesOrderID, o.OrderDate
FROM Sales.SalesOrderHeader o
JOIN #MinOrderDates t
```

Asking a Better Question: Advanced Queries

```
ON o.CustomerID = t.CustomerID
AND o.OrderDate = t.OrderDate
ORDER BY o.CustomerID

DROP TABLE #MinOrderDates
```

We get back approximately 19,134 rows (plus or minus a few, depending on what experimentation you've done with your data):

```
(19119 row(s) affected)
CustomerID  SalesOrderID OrderDate
-----
1           43860        2001-08-01 00:00:00.000
2           46976        2002-08-01 00:00:00.000
3           44124        2001-09-01 00:00:00.000
4           46642        2002-07-01 00:00:00.000

...
...
...
29481       45427        2002-02-13 00:00:00.000
29482       49746        2003-03-22 00:00:00.000
29483       49665        2003-03-13 00:00:00.000

(19134 row(s) affected)
```

As previously stated, don't worry if your results are slightly different from those shown here—it just means you've been playing around with the AdventureWorks data a little more or a little less than I have.

The fact that we are building two completely separate result sets here is emphasized by the fact that you see two different row(s) affected in the results. That, more often than not, has a negative impact on performance. We'll explore this further after you explore your options some more.

Sometimes using this two-query approach is simply the only way to get things done without using a cursor—this is not one of those times.

Okay, so if we want this to run in a single query, we need to find a way to look up each individual. We can do this by making use of an inner query that performs a lookup based on the current `CustomerID` in the outer query. We will then need to return a value back out to the outer query, so it can match things up based on the earliest order date.

It looks like this:

```
SELECT o1.CustomerID, o1.SalesOrderID, o1.OrderDate
FROM Sales.SalesOrderHeader o1
WHERE o1.OrderDate = (SELECT MIN(o2.OrderDate)
                      FROM Sales.SalesOrderHeader o2
                      WHERE o2.CustomerID = o1.CustomerID)
ORDER BY CustomerID
```

With this, we get back the same 19,134 rows. There are a few key things to notice in this query:

- ❑ We see only one row(s) affected line — giving us a good clue that only one query plan had to be executed.
- ❑ The outer query (in this example) looks pretty much just like a nested subquery. The inner query, however, has an explicit reference to the outer query (notice the use of the “o1” alias).
- ❑ Aliases are used in both queries — even though it looks like the outer query shouldn’t need one — that’s because they are required whenever you explicitly refer to a column from the other query (inside refers to a column on the outside or vice versa).

The latter point of needing aliases is a big area of confusion. The fact is that sometimes you need them, and sometimes you don’t. While I don’t tend to use them at all in the types of nested subqueries that you looked at in the early part of this chapter, I alias everything when dealing with correlated subqueries.

The hard-and-fast “rule” is that you must alias any table (and its related columns) that’s going to be referred to by the other query. The problem is that this can quickly become very confusing. The way to be on the safe side is to alias everything — that way you’re positive of which table in which query you’re getting your information from.

We see that 19134 row(s) affected only once. That’s because it affected 19,134 rows only one time. Just by observation, we can guess that this version probably runs faster than the two-query version, and, in reality, it does. Again, we’ll look into this a bit more shortly.

In this particular query, the outer query references the inner query only in the WHERE clause — it could also have requested data from the inner query to include in the select list.

Normally, it’s up to us whether we want to make use of an alias or not, but, with correlated subqueries, they are required. This particular query is a really great one for showing why because the inner and outer queries are based on the same table. Since both queries are getting information from each other, without aliasing, how would they know which instance of the table data that you were interested in?

Correlated Subqueries in the SELECT List

Subqueries can also be used to provide a different kind of answer in your selection results. This kind of situation is often found where the information you’re after is fundamentally different from the rest of the data in your query (for example, you want an aggregation on one field, but you don’t want all the baggage from that to affect the other fields returned).

To test this out, let’s just run a somewhat modified version of the query we used in the last section. What we’re going to say we’re after here is just the name of the customer and the first date on which they ordered something.

This one creates a somewhat more significant change than is probably apparent at first. We’re now asking for the customer’s name, which means that we have to bring the Customers table into play. In addition, we no longer need to build any kind of condition in — we’re asking for all customers (no restrictions), we just want to know when their first order date was.

Asking a Better Question: Advanced Queries

The query actually winds up being a bit simpler than the last one, and it looks like this:

```
SELECT c.LastName,
       (SELECT MIN(OrderDate)
        FROM Sales.SalesOrderHeader o
        WHERE o.ContactID = c.ContactID)
        AS "Order Date"
  FROM Person.Contact c
```

This gets us data that looks something like this:

LastName	Order Date
Achong	2001-09-01 00:00:00.000
Abel	2003-09-01 00:00:00.000
Abercrombie	2001-09-01 00:00:00.000
Acevedo	2001-09-01 00:00:00.000
...	
...	
...	
He	2004-04-12 00:00:00.000
Zheng	2004-02-15 00:00:00.000
Hu	2003-11-17 00:00:00.000
(19972 row(s) affected)	

Note that, if you look down through all the data, there are a couple of rows that have a `NULL` in the `OrderDate` column. Why do you suppose that is? The cause is, of course, that there is no record in the `SalesOrderHeader` table that matches the then current record in the `Customers` table (the outer query).

This brings us to a small digression to take a look at a particularly useful function for this situation—`ISNULL()`.

Dealing with `NULL` Data—the `ISNULL` Function

There are actually a few functions that are specifically meant to deal with `NULL` data, but the one of particular use to you at this point is `ISNULL()`. `ISNULL()` accepts a variable or expression and tests it for a null value. If the value is indeed `NULL`, then the function returns some other pre-specified value. If the original value is not `NULL`, then the original value is returned. This syntax is pretty straightforward:

```
ISNULL(<expression to test>, <replacement value if null>)
```

So, for example:

ISNULL Expression	Value Returned
<code>ISNULL(NULL, 5)</code>	5
<code>ISNULL(5, 15)</code>	5
<code>ISNULL(@MyVar, 0) where @MyVar IS NULL</code>	0
<code>ISNULL(@MyVar, 0) where @MyVar = 3</code>	3
<code>ISNULL(@MyVar, 0) where @MyVar = 'Fred Farmer'</code>	Fred Farmer

Chapter 6

Now let's see this at work in our query:

```
SELECT c.LastName,
    ISNULL(CAST((SELECT MIN(OrderDate)
        FROM Sales.SalesOrderHeader o
        WHERE o.ContactID = c.ContactID) AS varchar), ' NEVER ORDERED')
    AS 'Order Date'
FROM Person.Contact c
```

Now, in the lines that we had problems with, we went from simple nulls to something more descriptive:

```
...
Akers                               Sep  1 2001 12:00AM
Alameda                            NEVER ORDERED
Alberts                             Sep  1 2002 12:00AM
...
```

Notice that I also had to put the CAST() function into play to get this to work. The reason has to do with casting and implicit conversion. Since the first row starts off returning a valid date, the column OrderDate is assumed to be of type datetime. However, when we get to our first ISNULL, there is an error generated since NEVER ORDERED can't be converted to the datetime datatype. Keep CAST() in mind—it can help you out of little troubles like this one. This is covered a bit more later in the chapter.

So, at this point, you've seen correlated subqueries that provide information for both the WHERE clause, and for the select list. You can mix and match these two in the same query if you wish.

Derived Tables

Sometimes you get in situations where you need to work with the results of a query, but you need to work with the results of that query in a way that doesn't really lend itself to the kinds of subqueries that I've discussed up to this point. An example is where, for each row in a given table, you may have multiple results in the subquery, but you're looking for a more complex action that your IN operator provides. Essentially, what I'm talking about here are situations where you wish you could use a JOIN operator on your subquery.

It's at times like these that you turn to a somewhat lesser known construct in SQL—a *derived table*. A derived table (sometimes called an “inline view”) is made up of the columns and rows of a result set from a query (heck, they have columns, rows, data types, etc., just like normal tables, so why not use them as such?).

Imagine for a moment that you want to get a list of customers that ordered a particular product—say, a minipump. No problem! Your query might look something like this:

```
SELECT c.FirstName, c.LastName
    FROM Person.Contact AS c
    JOIN Sales.SalesOrderHeader AS soh
        ON c.ContactID = soh.ContactID
    JOIN Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
```

Asking a Better Question: Advanced Queries

```
JOIN Production.Product AS p
    ON sod.ProductID = p.ProductID
    WHERE p.Name = 'Minipump'
```

Okay, so that was easy. Now I'm going to throw you a twist though—now say I want to know all the customers that ordered not only a minipump, but also the AWC Logo Cap. Notice that I said they have to have ordered both—now you have a problem. You're first inclination might be to write something like:

```
WHERE p.Name = 'Minipump' AND p.Name = 'AWC Logo Cap'
```

But that's not going to work at all—each row is for a single product, so how can it have both Minipump and AWC Logo Cap as the name at the same time? Nope—that's not going to get it at all (indeed, while it will run, you'll never get any rows back at all).

What we really need here is to join the results of a query to find buyers of minipumps with the results of a query to find buyers of AWC Logo Caps. How do we join results though? Well, as you might expect given the title of this section, through the use of derived tables.

To create our derived table, you need two things:

- To enclose our query that generates the result set in parenthesis
- To alias the results of the query

So, the syntax looks something like this:

```
SELECT <select list>
FROM (<query that returns a regular result set>) AS <alias name>
JOIN <some other base or derived table>
```

So let's take this now and apply it to our requirements. Again, what we want is the names of all the companies that have ordered both a minipump and AWC Logo Cap. So, our query should look something like this:

```
SELECT DISTINCT c.FirstName, c.LastName
    FROM Person.Contact AS c
    JOIN (SELECT ContactID
        FROM Sales.SalesOrderHeader AS soh
        JOIN Sales.SalesOrderDetail AS sod
            ON soh.SalesOrderID = sod.SalesOrderID
        JOIN Production.Product AS p
            ON sod.ProductID = p.ProductID
            WHERE p.Name = 'Minipump') pumps
    ON c.ContactID = pumps.ContactID
    JOIN (SELECT ContactID
        FROM Sales.SalesOrderHeader AS soh
        JOIN Sales.SalesOrderDetail AS sod
            ON soh.SalesOrderID = sod.SalesOrderID
        JOIN Production.Product AS p
            ON sod.ProductID = p.ProductID
            WHERE p.Name = 'AWC Logo Cap') caps
    ON c.ContactID = caps.ContactID
```

Chapter 6

As it happens, it seems that the combination of minipumps and caps is very popular—we get 83 rows:

FirstName	LastName
Aidan	Delaney
Alexander	Deborde
Amy	Alberts
...	
...	
...	
Valerie	Hendricks
Yale	Li
Yuping	Tian

(83 row(s) affected)

If you want to check things out on this, just run the queries for the two derived tables separately and compare the results.

For this particular query, I needed to use the DISTINCT keyword. If I didn't, then I would have potentially received multiple rows for each customer.

As you can see, we were able to take a seemingly impossible query and make it both possible and even reasonably well performing.

Keep in mind that derived tables aren't the solutions for everything. For example, if the result set is going to be fairly large and you're going to have lots of joined records, then you may want to look at using a temporary table and building an index on it (derived tables have no indexes). Every situation is different, but now you have one more tool in your arsenal.

The EXISTS Operator

I call EXISTS an operator, but all you'll hear the Books Online call it is a keyword. That's probably because it defies description in some senses. It's both an operator much like the IN keyword is, but it also looks at things just a bit differently.

When you use EXISTS, you don't really return data—instead, you return a simple TRUE/FALSE regarding the existence of data that meets the criteria established in the query that the EXISTS statement is operating against.

Let's go right to an example, so you can see how this gets applied. For this example, we're going to reuse one of our nested selects examples from earlier—we want a list of employees that have applied for another position within the company at some point.

```
SELECT e.EmployeeID, FirstName, LastName
FROM HumanResources.Employee e
JOIN Person.Contact c
ON e.ContactID = c.ContactID
```

Asking a Better Question: Advanced Queries

```
WHERE EXISTS
    (SELECT EmployeeID
     FROM HumanResources.JobCandidate jc
     WHERE jc.EmployeeID = e.EmployeeID)
```

This gets us what amounts to the same two records that we had under the more standard nested query:

EmployeeID	FirstName	LastName
41	Peng	Wu
268	Stephen	Jiang

(2 row(s) affected)

We could have easily done this same thing with a join:

```
SELECT DISTINCT e.EmployeeID, FirstName, LastName
FROM HumanResources.Employee e
JOIN Person.Contact c
    ON e.ContactID = c.ContactID
JOIN HumanResources.JobCandidate jc
    ON jc.EmployeeID = e.EmployeeID
```

This join-based syntax, for example, would have yielded us exactly the same results (subject to possible sort differences). So why, then, would we need this new syntax? Performance—plain and simple.

When you use the `EXISTS` keyword, SQL Server doesn't have to perform a full row by row join. Instead, it can look through the records until it finds the first match and stop right there. As soon as there is a single match, the `EXISTS` is true, so there is no need to go further. The performance difference here is even more marked than with the inner join. SQL Server just applies a little reverse logic versus the straight `EXISTS` statement. In the case of the `NOT` we're now using, SQL can still stop looking as soon as it finds one matching record—the only difference is that it knows to return `FALSE` for that lookup rather than `TRUE`. Performance wise, everything else about the query is the same.

Using `EXISTS` in Other Ways

If you work around SQL creation scripts much, you will see an oddity preceding many `CREATE` statements. It will look something like this:

```
IF EXISTS (SELECT * FROM sysobjects WHERE id =
object_id(N'[Sales].[SalesOrderHeader]') AND OBJECTPROPERTY(id,
N'IsUserTable') = 1)
DROP TABLE [Sales].[ SalesOrderHeader]
GO

CREATE TABLE [Sales].[ SalesOrderHeader] (
...
...
```

You may see variants on the theme—that is, they may use `sys.objects`, `sys.databases`, or the `INFORMATION_SCHEMA` views—but the concept is still the same: They are testing to see whether an object exists before performing a `CREATE`. Sometimes they may just skip the `CREATE` if the table already exists,

Chapter 6

and sometimes they may drop it (as I did in the example above). The idea is pretty simple though—they want to skip a potential error condition (the `CREATE` would error out and blow up your script if the table already existed).

Just as a simple example, we'll build a little script to create a database object. We'll also keep the statement to a minimum since we're interested in the `EXISTS` rather than the `CREATE` command:

```
USE master

GO

IF NOT EXISTS (SELECT 'True' FROM sys.databases WHERE name = 'DBCreateTest')
BEGIN
    CREATE DATABASE DBCreateTest
END
ELSE
BEGIN
    PRINT 'Database already exists. Skipping CREATE DATABASE Statement'
END
GO
```

The first time you run this, there won't be any database called `DBCreateTest` (unless by sheer coincidence that you created something called that before you got to this point), so the database will be created.

Now run the script a second time, and you'll see a change:

```
Database already exists. Skipping CREATE DATABASE Statement
```

So, without much fanfare or fuss, we've added a rather small script in that will make things much more usable for the installers of your product. That may be an end user who bought your off-the-shelf product, or it may be you—in which case it's even better that it's fully scripted.

The long and the short of it is that `EXISTS` is a very handy keyword indeed. It can make some queries run much faster, and it can also simplify some queries and scripts.

A word of caution here—this is another one of those places where it's easy to get trapped in "traditional thinking." While `EXISTS` blows other options away in a large percentage of queries where `EXISTS` is a valid construct, that's not always the case—just remember that rules are sometimes made to be broken.

Mixing Data Types: `CAST` and `CONVERT`

I'm not going to spend a ton of time on these since these are usually a somewhat beginning concept, but I still see them overlooked on a regular basis by people that have used SQL for a long time, so they deserve some mention even in a more advanced book.

Both `CAST` and `CONVERT` perform data type conversions for you. In most respects, they both do the same thing, with the exception that `CONVERT` also does some date formatting conversions that `CAST` doesn't offer.

Asking a Better Question: Advanced Queries

So, the question probably quickly rises to your mind — hey, if CONVERT does everything that CAST does, and CONVERT also does date conversions, why would I ever use CAST? I have a simple answer for that — ANSI compliance. CAST is ANSI compliant, and CONVERT isn't — it's that simple.

Let's take a look at the syntax for each.

```
CAST (expression AS data_type)  
CONVERT(data_type, expression[, style])
```

With a little flip-flop on which goes first, and the addition of the formatting option on CONVERT (with the style argument), they have basically the same syntax.

CAST and CONVERT can deal with a wide variety of data type conversions that you'll need to do when SQL Server won't do it implicitly for you. For example, converting a number to a string is a very common need. To illustrate:

```
USE AdventureWorks  
  
SELECT 'The Customer has an Order numbered ' + SalesOrderID  
FROM Sales.SalesOrderHeader  
WHERE CustomerID = 5
```

Will yield an error:

```
Msg 245, Level 16, State 1, Line 1  
Conversion failed when converting the varchar value 'The Customer has an Order  
numbered ' to data type int.
```

But change the code to convert the number first:

```
SELECT 'The Customer has an Order numbered ' + CAST(SalesOrderID AS varchar)  
FROM Sales.SalesOrderHeader  
WHERE CustomerID = 5
```

And you get a much different result:

```
-----  
The Customer has an Order numbered 47436  
The Customer has an Order numbered 48374  
The Customer has an Order numbered 49534  
The Customer has an Order numbered 50746  
The Customer has an Order numbered 53607  
The Customer has an Order numbered 59014  
The Customer has an Order numbered 65307  
The Customer has an Order numbered 71890  
  
(8 row(s) affected)
```

Chapter 6

We can also convert dates. Take this query for example:

```
SELECT OrderDate, CAST(OrderDate AS varchar) AS "Converted"
FROM Sales.SalesOrderHeader
WHERE SalesOrderID = 43660
```

It yields a result that's useful enough:

OrderDate	Converted
2001-07-01 00:00:00.000	Jul 1 2001 12:00AM

(1 row(s) affected)

CAST can still do date conversion, you just don't have any control over the formatting as you do with CONVERT. For example:

```
SELECT OrderDate, CONVERT(varchar(12), OrderDate, 111) AS "Converted"
FROM Sales.SalesOrderHeader
WHERE SalesOrderID = 43660
```

Yields us:

OrderDate	Converted
2001-07-01 00:00:00.000	2001/07/01

(1 row(s) affected)

Which is quite a bit different from what CAST did. Indeed, we could have converted to any one of several two-digit or four-digit year formats.

All you need is to supply a code at the end of the CONVERT function (111 in the preceding example gave us the JAPAN standard, with a four-digit year) that tells what format you want. Anything in the 100s is a four-digit year, anything less than 100, with a few exceptions, is a two-digit year. The available formats can be found in Books Online under the topic of CONVERT or CASE.

Keep in mind that you can set a split point that SQL Server will use to determine whether a two-digit year should be have a 20 added on the front or a 19. The default breaking point is 49/50—a two-digit year of 49 or less will be converted using a 20 on the front. Anything higher will use a 19. These can be changed using sp_configure, or by setting them in the server properties of the Management Studio.

Using External Calls to Perform Complex Actions

We have always had the need, on occasion, to get information that is sourced outside of SQL Server. For the vast, vast majority of installations, actually getting that information from within SQL Server was out

Asking a Better Question: Advanced Queries

of reach. Instead, there was typically a client or middle tier component that sorted out what was needed from SQL Server and what was needed from the external source.

In many ways, this was just fine—after all, having your database server hung up waiting on an external call seems risky at best, and deadly at worst. Who knows how long before that call is going to return (if ever?). The risk of hung processes within your database server winds up being fairly high.

Now, I said for the *majority* of installations, and that implies that a few got around it—and they did. There were a few different methods available.

First, there was the idea of an extended stored procedure. These are DLLs that have that you can create in C using special SQL Server libraries. They run in process with SQL Server and can be (assuming you have a smart DLL writer) very fast, save for one problem—an example is an external call. That means that we are beholden to the external process we are calling to return to us in a timely fashion. The additional issue was one of general safety. Since you’re running in process to SQL Server, if your DLL crashes, then SQL Server is going to crash (if you’re distributing software, I’m sure you can guess at how your customer would react if your product was taking down their SQL Server installation). Last, but not least, very few had the knack for figuring out how to get these written.

Another solution was added to SQL Server in the OLE/COM era. The `sp_CreateOAMethod` family of stored procedures allowed you to instantiate a COM object and make calls to it. These passed data back and forth using variants, and were always run out of process. They were safer, but they were clumsy at best and painfully slow.

With the advent of .NET and SQL Server becoming CLR language aware, we live in a new world. You can write your scripts using any .NET language, and can instantiate the objects you need to get the job done. You can create user-defined functions to call external processes—such as cross-communicating with some other online system that you cannot directly link to. Imagine, for a moment, allowing SQL Server to apply information gleaned from a Web service and merge that data in the end query? Heady stuff.

The possibilities are endless; however, you need to keep your head about this. External calls are still external calls! Any time you rely on something external to your system, you are now at the mercy of that external system. Be very, very careful with such calls.

External calls should be considered to be an extreme measure. You are taking risks in terms of security (what is the risk of someone spoofing your external source?) and also taking an extreme performance risk. Tread lightly in this area.

Performance Considerations

We’ve already touched on some of the macro-level “what’s the best thing to do” stuff as we’ve gone through the chapter, but, like most things in life, it’s not as easy as all that. What I want to do here is provide something of a quick reference for performance issues for your queries. I’ll try to steer you toward the right kind of query for the right kind of situation.

Yes, it's time again folks for one of my now famous soapbox diatribes. At issue this time is the concept of blanket use of blanket rules.

What I'm going to be talking about in this section is about the way that things usually work. The word usually is extremely operative here. There are very few rules in SQL that will be true 100 percent of the time. In a world full of exceptions, SQL has to be at the pinnacle of that—exceptions are a dime a dozen when you try and describe the performance world in SQL Server.

In short, you need to gage just how important the performance of a given query is. If performance is critical, then don't take these rules too seriously—instead, use them as a starting point, and then TEST, TEST, TEST!!!

JOINs vs. Subqueries vs. ?

Deciding between joins and subqueries (and for that matter, other options) is that area I mentioned earlier in the chapter that I had a heated debate with a coworker over. And, as you might expect when two people have such conviction in their point of view, both of us were correct up to a point (and it follows, wrong up to a point).

The long-standing, traditional viewpoint about subqueries has always been that you are much better off to use joins instead if you can. This is absolutely correct—sometimes. In reality, it depends on a large number of factors. The following is a table that discusses some of the issues that the performance balance will depend on, and which side of the equation they favor.

Situation	Favors
The value returned from a subquery is going to be the same for every row in the outer query.	Prequery. Declaring a variable and then selecting the needed value into that variable will allow the would-be subquery to be executed just once rather than once for every record in the outer table. The optimizer in SQL Server is actually pretty smart about this and will do the prequery for you if it detects the scenario, but do not rely on it. If you know this is the scenario, perform your own prequery just to be sure.
Both tables are relatively small (say 10,000 records or less).	Subqueries. I don't know the exact reasons, but I've run several tests on this, and it held up pretty much every time. I suspect that the issue is the lower overhead of a lookup vs. a join when all the lookup data fits on just a data page or two.
The match, after considering all criteria, is going to return only one value.	Subqueries. Again, there is much less overhead in going and finding just one record and substituting it than in having to join the entire table.
The match, after considering all criteria, is going to return only a relatively few values, and there is no index on the lookup column.	Subqueries. A single lookup or even a few lookups will usually take less overhead than a hash join.

Situation	Favors
The lookup table is relatively small, but the base table is large.	Nested subqueries if applicable; joins if vs. a correlated subquery. With subqueries the lookup will happen only once and has relatively low overhead. With correlated subqueries, however, you will be cycling the lookup many times—in this case, the join would be a better choice in most cases.
Correlated subquery vs. join	Join. Internally, a correlated subquery is going to create a nested loop situation. This can create quite a bit of overhead. It is substantially faster than cursors in most instances, but slower than other options that might be available.
Derived tables vs. whatever	Derived tables typically carry a fair amount of overhead, so proceed with caution. The thing to remember is that they are run (derived if you will) once, and then they are in memory, so, most of the overhead is in the initial creation and the lack of indexes in larger result sets. They can be fast or slow—it just depends. Think before coding on these.
EXISTS vs. whatever	EXISTS. It does not have to deal with multiple lookups for the same match—once it finds one match for that particular row, it is free to move onto the next lookup—this can seriously cut down on overhead.

These are just the highlights. The possibilities of different mixes and additional situations are positively endless.

I can't stress enough how important it is, when in doubt—heck, even when you're not in doubt but performance is everything—to make reasonable tests of competing solutions to the problem. Most of the time the blanket rules will be fine, but not always. By performing reason tests, you can be certain you've made the right choice.

Summary

The query options you learned back in Chapter 3 cover perhaps 80 percent or more of the query situations that you run into, but it's that other 20 percent that can kill you. Sometimes the issue is whether you can even find a query that will give you the answers you need. Sometimes it's that you have a particular query or sproc that has unacceptable performance. Whatever the case, you'll run across plenty of situations where simple queries and joins just won't fit the bill. You need something more, and, hopefully, the options covered in this chapter have given you a little more of an arsenal to deal with those tough situations.

7

Daring to Design

And so I come to another one of those things where I have to ponder how much to assume you already know. “To normalize, or not to normalize—THAT is the question!” Okay, the real question is one of whether you already understand the most basic tenants of relational database design yet or not. Since you come to this book with a degree of experience already, I’m going to take an approach that assumes you’ve heard of it, know it’s important, and even grasp the basics of it. I’m going to assume you need the information filled in for you rather than that you are starting from scratch.

With the exception of a chapter or two, this book has an *Online Transaction Processing*, or *OLTP*, flare to the examples. Don’t get me wrong; I will point out, from time to time, some of the differences between *OLTP* and its more analysis-oriented cousin *Online Analytical Processing* (*OLAP*). My point is that you will, in most of the examples, be seeing a table design that is optimized for the most common kind of database—*OLTP*. Thus, the table examples will typically have a database layout that is, for the most part, *normalized* to what is called the third normal form.

What is “normal form”? We’ll start off by taking a very short look at that in this chapter and then will move quickly onto more advanced concepts. For the moment though, just say that it means your data has been broken out into a logical, nonrepetitive format that can easily be reassembled into the whole. In addition to normalization (which is the process of putting your database into normal form), we’ll also be examining the characteristics of *OLTP* and *OLAP* databases. And, as if we didn’t have enough to do between those two topics, we’ll also be looking at many examples of how the constraints we’ve already seen are implemented in the overall solution.

Normalization 201

I want to start off by saying that there are six normal forms (plus or minus one or two depending on which academician you listen to). You’ll leave several of those to the academicians though. Those in the real world usually deal with only three normal forms. Indeed, a fully normalized database is one that is generally considered to be one that is normalized to the third normal form.

The concept of normalization has to be one of most over-referenced yet misunderstood concepts in programming. Everyone thinks they understand it, and many do in at least its academic form.

Unfortunately, it also tends to be one of those things that many database designers wear like a cross—it is somehow their symbol that they are “real” database architects. What it really is, however, is a symbol that they know what the normal forms are—and that’s all. Normalization is really just one piece of a larger database design picture. Sometimes you need to normalize your data—then again, sometimes you need to deliberately de-normalize your data. Even within the normalization process, there are often many ways to achieve what is technically a normalized database.

My point in this latest soapbox diatribe is that normalization is a theory, and that’s all it is. Once you choose to either implement a normalized strategy or not, what you have is a database—hopefully the best one you could possibly design. Don’t get stuck on what the books (including this one) say you’re supposed to do—do what’s right for the situation that you’re in. As the author of this book, all I can do is relate concepts to you—I can’t implement them for you, and neither can any other author (at least not with the written word). You need to pick and choose between these concepts in order to achieve the best fit and the best solution.

We’ve already looked at how to create a primary key and some of the reasons for using one in our tables—if we want to be able to act on just one row, then we need to be able to uniquely identify that row. The concepts of normalization are highly dependent on issues surrounding the definition of the primary key and what columns are dependent on it. One phrase you might hear frequently in normalization is:

The key, the whole key, and nothing but the key.

The somewhat fun addition to this is:

The key, the whole key, and nothing but the key, so help me Codd!

This is a super-brief summarization of what normalization is about out to the third normal form (for those who don’t know, Codd is considered the father of relational design). When you can say that all your columns are dependent only on the whole key and nothing more or less, then you are at third normal form.

Now I want to review the various normal forms and what each does for you.

Where to Begin

The concepts of relational database design are founded on the notion of *entities* and *relations*. If you’re familiar with object-oriented programming, then you can liken most top-level entities to objects in an object model. Much as a parent object might contain other objects that further describe it, tables may have a child or other table that further describe the rows in the original table.

An entity will generally tie to one “parent” table. That table will usually have one and only one row per instance of entity you’re describing (for example, a table that is the top table for tracking orders in a system will have only one row per individual order). The one entity may, however, require multiple other tables to provide additional descriptive information (for example, a details or line item table to carry a list of all the things that were purchased on that particular order).

A relation is a representation of how two entities relate to each other logically. For example, a customer is a different entity from an order, but they are related. For example, you cannot have so much as one order without at least one customer. Furthermore, your order relates to only one customer.

As you start the process of “normalizing” these entities and relations into tables, some things about your data are assumed even before you get to the first of the normal forms.

- The table should describe one and only one entity. (No trying to shortcut and combine things!)
- All rows must be unique, and there must be a primary key.
- The column and row order must not matter.

As you gain experience, this will become less of a “process” and more of the natural starting point for your tables. You will find that creating a normalized set of tables will be the way things flow from your mind to start with rather than anything special that you have to do.

Getting to Third Normal Form

As I indicated earlier, there are, from a practical point of view, three normal forms.

- The first normal form (1NF)** is all about eliminating repeating groups of data and guaranteeing *atomicity* (the data is self-contained and independent). At a high level, it works by creating a primary key (which you already have), then moving any repeating data groups into new tables, creating new keys for those tables, and so on. In addition, you break out any columns that combine data into separate rows for each piece of data.
- Second normal form (2NF)** further reduces the incidence of repeated data (not necessarily groups). Second normal form has two rules to it:
 - The table must meet the rules for first normal form. (Normalization is a building block kind of process — you can’t stack the third block on if you don’t have the first two there already.)
 - Each column must depend on the *whole* key.
- Third normal form (3NF)** deals with the issue of having all the columns in your table not just be dependent on something — but the right thing. Third normal form has just three rules to it:
 - The table must be in 2NF (I told you this was a building block thing).
 - No column can have any dependency on any other non-key column.
 - You cannot have derived data.

Other Normal Forms

There are a few other forms out there that are considered, at least by academics, to be part of the normalization model. These include:

- ❑ **Boyce-Codd** (considered to really just be a variation on third normal form) — This one tries to address situations where you have multiple overlapping candidate keys. This can only happen if:
 - a. All the candidate keys are composite keys (that is, it takes more than one column to make up the key).
 - b. There is more than one candidate key.
 - c. The candidate keys each have at least one column that is in common with another candidate key.

This is typically a situation where any number of solutions works, and almost never gets logically thought of outside the academic community.

- ❑ **Fourth normal form** — This one tries to deal with issues surrounding multivalued dependence. This is the situation where, for an individual row, no column depends on a column other than the primary key and depends on the whole primary key (meeting third normal form). However, there can be rather odd situations where one column in the primary key can depend separately on other columns in the primary key. These are rare and don't usually cause any real problem. Thus, they are largely ignored in the database world, and you will not address them here.
- ❑ **Fifth normal form** — Deals with non-loss and loss decompositions. Essentially, there are certain situations where you can decompose a relationship such that you cannot logically recompose it into its original form. Again, these are rare, largely academic, and you won't deal with them any further here.

This is, of course, just a really quick look at these — and that's deliberate on my part. The main reason you need to know these in the real world is either to impress your friends (or prove to them you're a "know it all") and to not sound like an idiot when some database guru comes to town and starts talking about them. However you choose to use it, I do recommend against attempting to use it to get dates.

Relationships

Well, I've always heard from women that men immediately leave the room if you even mention the word "relationship." With that in mind, I hope that I didn't just lose about half my readers.

I am, of course, kidding — but not by as much as you might think. Experts say the key to successful relationships is that you know the role of both parties and that everyone understands the boundaries and rules of the relationship that they are in. I can be talking about database relationships with that statement every bit as much as people relationships.

There are three different kinds of major relationships:

- ❑ **One-to-one** — This is exactly what it says it is. A one-to-one relationship is one where the fact that you have a record in one table means that you have exactly one matching record in another table.
- ❑ **One-to-many** — This is one form of your run-of-the-mill, average, everyday foreign key kind of relationship. Usually, this is found in some form of header/detail relationship, and generally implements some idea of a parent to child hierarchy. For example, for every one customer, you might have several orders.

- **Many-to-many** — In this type of relationship, both sides of the relationship may have several records that match. An example of this would be the relationship of products to orders—an order may contain several products, and, likewise, a product will appear on many orders. SQL Server has no way of physically establishing a direct many-to-many relationship, so you cheat by having an intermediate table to organize the relationship.

Each of these has some variations depending on whether one side of the relationship is nullable or not. For example, instead of a one-to-one relationship, you might have a zero or one-to-one relationship.

Diagramming

Entity-relationship diagrams (ERDs) are an important tool in good database design. Small databases can usually be easily created from a few scripts and implemented directly without drawing things out at all. The larger your database gets, however, the faster it becomes very problematic to just do things “in your head.” ERDs solve a ton of problems because they allow you to quickly visualize and understand both the entities and their relationships.

For this book, I've decided to do things somewhat in reverse of how I've done things before. SQL Server includes a very basic diagramming tool that you can use as a starting point for building rudimentary ERDs. Unfortunately, it employs a proprietary diagramming methodology that does not look remotely like any standard I'm aware of out there. In addition, it does not allow for the use of logical modeling—something I consider a rather important concept. Therefore, I'm going to start off talking about the more standard diagramming methodologies first—later in the chapter you will look at SQL Server's built in tools and how to use them.

There are two reasonably common diagramming paradigms—IE and IDEF1X. You'll find both of these in widespread use, but I'm going to limit things here to a once over of the basics of IE (also called Information Engineering). For the record, IDEF1X is a perfectly good diagramming paradigm, and was first put forth by the U.S. Air Force. IE (again, Information Engineering—not Internet Explorer) is, however, the method I use personally, and I do so for just one reason—it is far more intuitive for the inexperienced reviewer of your diagrams. I also find it to be the far more common of the two.

I can't say enough about the importance of having the right tools. While the built-in tools at least give you “something,” they are a long way away from “what you need.”

ERD tools are anything but cheap—running from somewhere over \$1,000 to just under \$3,500 (that's per seat!). They are also something of a language unto themselves. Don't plan on just sitting down and going to work with any of the major ER tools—you had better figure on some spin-up time to get it to do what you expect.

Don't let the high price of these tools keep you from building a logical model. While Visio (the low-cost editions anyway) is not the world's answer to database design problems, it does do okay in a pinch for light logical modeling. That said, if you're serious about database design, and going to be doing a lot of it, you really need to find the budget for a real ERD tool.

Expense aside, there is no comparison between the productivity possible in the third-party tools out there and the built-in tools. Depending on the ER tool you select, they give you the capability to do things like:

- Create logical models, and then switch back and forth between the logical and physical model.
- Work on the diagram offline — then propagate all your changes to the physical database at one time (when you're ready, and opposed to when you need to log off).
- Reverse engineer your database from any one of a number of mainstream RDBMS systems (even some ISAM databases), and then forward engineer them to a completely different RDBMS.
- Create your physical model on numerous different systems.

This really just scratches the surface.

A Couple of Relationship Types

Before you get going too far in more diagramming concepts, I want to explore two types of relationships: identifying and non-identifying.

Identifying Relationships

For some of you, I'm sure the term *identifying relationship* brings back memories of some boyfriend or girlfriend you've had in the past who got just a little over possessive — this is not that kind of relationship. Instead, you're dealing with the relationships that are defined by foreign keys.

An identifying relationship is one where the column or columns (remember, there can be more than one) being referenced (in the parent table) are used as all or part of the referencing (child) table's primary key. Since a primary key serves as the identity for the rows in a table, and all or part of the primary key for the child table is dependent on the parent table — the child table can be said to, at least in part, be "identified" by the parent table.

Non-Identifying Relationships

Non-identifying relationships are those that are created when you establish a foreign key that does not serve as part of the referencing (child) table's primary key. This is extremely common in situations where you are referencing a domain table — where essentially the sole purpose of the referenced table is to limit the referencing field to a set list of possible choices.

The Entity Box

One of the many big differences you'll see in both IE and IDEF1X versus SQL Server's own brand of diagramming comes in the *entity box*. The entity box, depending on whether you're dealing with logical or physical models, equates roughly to a table. By looking over the entity box, you should be able to easily identify the entity's name, primary key, and any attributes (effectively columns) that entity has. In addition, the diagram may expose other information such as the attribute's data type or whether it has a foreign key defined for it. As an example, consider the entity box in Figure 7-1.

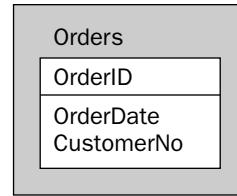


Figure 7-1

The name of our entity is kept on the top outside the box. Then, in the top area of the overall box, but in a separate box of its own, you have the primary key (you'll look at an example with more than one column in the primary key shortly), and last, but not least, come the attributes of the entity.

Take a look at a slightly different entity (Figure 7-2):

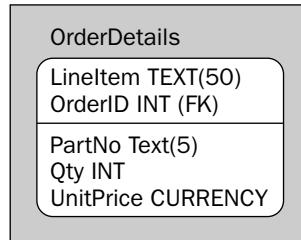


Figure 7-2

Several new things appear:

- The data types (I've turned on the appropriate option)
- Foreign keys (If any—again I've turned on the option to make this show).
- You have multiple columns in the primary key (everything above the line is part of the primary key).
- This time, the entity is rounded on the corners. This tells you that this table is identified (remember identifying relationships?) by at least one other table.

Depending on the ER tool, the data types can be defined right within the ER diagram. Also, as you draw the lines that form your relationships (you'll look at those shortly), you are able to define foreign keys, which can also be shown. For most available ER tools, you can even tell the tool to automatically define the referenced field(s) in the foreign key relationship as being part (or possibly all) of the primary key in the referencing table.

The Relationship Line

There are two kinds, and they match 100 percent with our relationship types:

Chapter 7

A solid line indicates an identifying relationship:



A broken or dashed line indicates a non-identifying relationship:



Again, an identifying relationship is one where the column that is referencing another table serves as all or part of the primary key of the referencing table. In a non-identifying relationship, the foreign key column has nothing to do with the primary key in the referencing table.

Terminators

Ahh, this is where things become slightly more interesting. The terminators you're talking about here are, of course, not the kind you'd see Arnold Schwarzenegger play in a movie—they are the end caps that you put on our relationship lines.

The terminators on our lines will communicate as much or more about the nature of our database as the entities themselves will. They are the thing that will tell you the most information about the true nature of the relationship, including the cardinality of the relationship.

Cardinality is, in its most basic form, the number of records on both sides of the relationship. When you say it is a one-to-many relationship, then you are indicating cardinality. Cardinality can, however, be much more specific than the zero, one, or many naming convention that you use more generically. Cardinality can address specifics, and is often augmented in a diagram with two numbers and a colon, such as:

- 1:M
- 1:6 (which, while meeting a one-to-many criteria, is more specific and says there is a maximum of 6 records on that side of the relationship.)

Walk through a couple of the parts of a terminator and examine what they mean:

Just as a reminder, the terminators that follow are the ones from the IE diagramming methodology. As I have indicated, there is another diagramming standard that is in widespread use (though I see it much less than IE) called IDEF1X. While its entity boxes are much like IE's, its terminators on the relationship lines are entirely different.

In the top half of the terminator shown in Figure 7-3, it is indicating the first half of our relationship. In this case, you have a zero. For the bottom half, we are indicating the second half of our relationship—in this case, a many. In this example, then, we have a zero, one, or many side of a relationship.

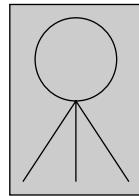


Figure 7-3

In Figure 7-4, you're not allowing nulls at this end of the relationship—this is a one or many end to a relationship

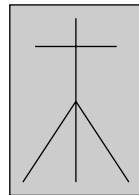


Figure 7-4

In Figure 7-5, you're back to allowing a zero at this end of the relationship, but you are now allowing a maximum of one. This is a zero or one side of a relationship.

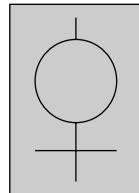


Figure 7-5

And last, but not least, you have Figure 7-6. This one is pretty restrictive—it's simply a “one” side of a relationship.

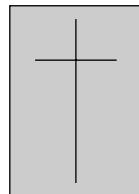


Figure 7-6

Chapter 7

Since it's probably pretty confusing to look at these just by themselves, take a look at a couple of example tables and relationships (Figure 7-7).

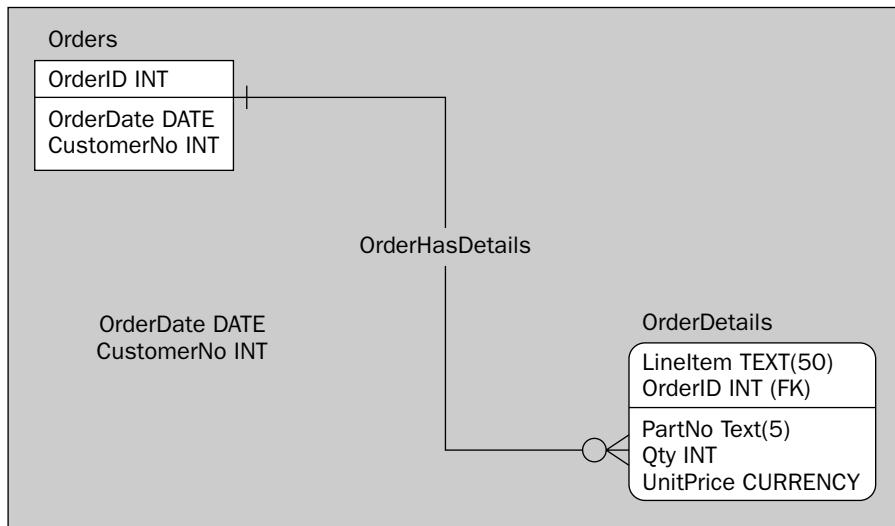


Figure 7-7

Figure 7-7 is a diagram that shows two tables that support the notion of just one logical entity—an order. You have an **Orders** table to keep track of information that is global to the order (this has just a **CustomerNo**, but it may have contained things like a shipping address, a date of the order, a due date, etc.). You also have an **OrderDetails** table to track the individual line items placed on this order. The diagram depicts not only your **Orders** and **OrderDetails** tables but also the one (the **Orders** side) to zero, one, or many (the **OrderDetails** side) relationship between the two tables. The relationship is an identifying relationship (solid, rather than dashed line), and the relationship is called **OrderHasDetails**.

In Figure 7-8, you add in a **Products** table:

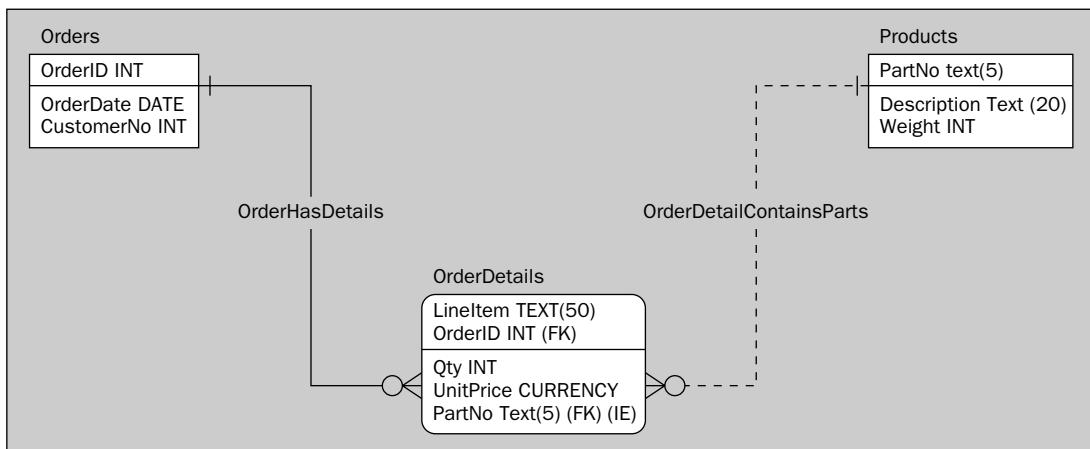


Figure 7-8

This new relationship is very similar to the relationship that you already looked at. It is again a one (Products this time) to zero, one, or many (OrderDetails again) relationship, but this is it is non-identifying (as represented by the broken line). The IE indicates that, for this table, PartNo is an *Inversion Entry*, or an index that is not associated with anything other than a foreign key. The Inversion Entry has been added as it usually makes sense to have an index on a field that is a foreign key (since it is a frequent target of lookups).

By looking at all three together, you can see that there is a many to many relationship between Orders and Products by virtue of their relationship through the OrderDetails table.

As I've indicated before, you are still really only scratching the surface of the different information that your ER diagrams can convey. Still, as you look later in the chapter at the SQL Server diagramming tools, you will be able to see that the more accepted methodologies out there have an awful lot more information to convey than what is offered by the included tools. In addition, just the nature of how tables are displayed makes information such as keys more visible and easier to read.

Logical versus Physical Design

In your database work, you may have already heard about the concepts of logical versus physical models. In this section, you'll be exploring the differences between the two.

The physical model is one that's probably pretty easy to grasp. It is essentially what you have been working with up to this point in the book. You can think of anything that you can perform a CREATE statement on as being part of the physical model. Indeed—if you run any statements in SQL Server on it at all then it must be part of the physical model.

That being said, a logical model is a means to a number of different things—the physical model in particular. This means that, as you work on the logical model, you are working your way toward being able to generate DDL (Data Definition Language—or things like CREATE, ALTER, and DROP statements).

Purpose of a Logical Model

The first thing to understand about logical models is that they have somewhat different goals than physical models do. A logical model does several things for you:

- Allows you to begin to build abstracts of complex, data-related business issues as well as provide a high-level effort at identifying your entities
- Allows you to use these abstracts to effectively communicate business rules and content as relates to data
- Represents the purest form of the data (before you start introducing the realities of what will really work)
- Serves as a major piece of documentation in the data requirements portion of your project

Because logical models aren't strictly rooted in the exact syntax to create the database, they give you a flexibility that you can't obtain from a physical model. You can attach dialog and rules to the logical model regardless of whether your particular RDBMS will support those rules or not. In short, it allows you squeeze in all the facts before you start paring down your design to a more specific implementation.

What's nice about this is that logical models allow you to capture all of your data rules in one place regardless of where that rule will be actually implemented. You will frequently run into situations where you cannot sensibly implement your rules in the databases. The rules in question may be data related, but due to some constraint or requirement, you need to implement them using more procedural code in your client or in some form of middle tier. With logical models — you go ahead and include the data related rules anyway.

Regardless of its source, you include all data-related information in a logical design to create one or more abstracts of the data in your system. These abstracts can then be used as a representation to your customer about what you really are intending to store and what rules you believe you have captured. Using such a representation early (and often) can save your projects valuable time and money by opening extra doors of communication. Even a customer who is not very data savvy can often look at the highest level diagrams and say things like "Where are the purchase requisitions?" Usually, you have some handy dandy explanation of why you called them something else and you can point to them on the diagram — other times, however, you find yourself uttering that most fearsome of words — "Oops!" I don't know about you, but I'd rather utter those words in the first weeks of a project rather than the first weeks of deployment. Logical modeling, when properly shared with the customer, can help avoid those deployment time "Oops" statements.

I can't do enough to stress the importance of sharing your logical design (there had better be one!) with your customer both early and often. With a little education of the customer in how to read your logical model (this should also include good documentation on cause and purpose of the entities and relationships of the model), you can save a fortune in both time and money.

I haven't met a developer with any real experience who hasn't, at least once (and probably far more often than that), learned the hard way about the cost of late changes to your system. Changing code is very expensive, but that typically doesn't even begin to touch what happens when you need to change your database late in a project. If you haven't done a good job of abstracting your database (3-tier or n-tier design), then every change you make to your database is going to cascade through tons of code. In other words, one little change in your database can potentially cost several — possibly hundreds or even thousands (depending on the size of the system) of changes in the code that accesses the database.

In short, communication is everything, and logical modeling should be a huge part of your tool set for communicating with your customer.

Parts of a Logical Model

A logical model contains three major parts:

- ❑ Structure
- ❑ Constraints
- ❑ Rules

The combination of these three should completely describe the requirements of the data in your system, but they may not translate entirely to the physical model. Some of the issues identified in the logical model may need to be implemented in some procedural form (such as in a middle-tier component). Other times, the entire logical model can be implemented through the various features of your RDBMS.

This is a really important point, and I want to stress it again—just because it's in your logical model doesn't mean that it will be in your physical database. A logical model should take into account all of your data requirements—even those that are not possible to implement in your RDBMS (for example, data that you might be retrieving from a third-party source—perhaps in an XML document or some other storage medium). Having everything in your logical model allows you to plan the physical design in such a way that you can be sure that you have addressed all data issues—not just those that will physically reside in the database.

Structure

Structure is that part of the logical design that deals with the concept of actually storing the data. When you deal with the structure of the database, you're talking about entities—most of which will translate to tables that will store our data—and the particular columns you are going to need to maintain the atomicity of your data.

Constraints

Constraints, from a logical model standpoint, are a bit broader than the way that you've used the word *constraint* up until now. Prior to now, when you used the word *constraint*, you were talking about a specific set of features to limit data to certain values. From a logical standpoint, a constraint is anything that defines the “what” question for our data—that is, what data is valid. A logical model includes constraints, which is to say that it includes things like:

- ❑ Data types (notice that this is really a separate thought from the notion that a column needs to exist or what the name of that column should be).
- ❑ Constraints in the form you're used to up until now—that is, `CHECK` constraints, foreign keys, or even primary keys and `UNIQUE` constraints (alternate keys). Each of these provides a logical definition of what data can exist in our database. This area would also include things like domain tables (which you would reference using foreign keys)—which restrict the values in a column to a particular “domain” list.

Rules

If constraints were the “what” in our data, then *rules* are the “when and how much” in our data.

When we define logical rules, we're defining things like “Do we require a value on this one?” (which equates to “Do we allow nulls?”) and “How many of these do we allow” (which defines the cardinality of our data—do we accept one or many?).

It's worth noting yet again that any of these parts may not be implemented in the physical part of your database—we may decide that the restrictions that you want to place on things will be handled entirely at the client—regardless of where the requirement is implemented, it should still be part of our comprehensive logical data model. It is only when you achieve this complete modeling of your data that you can really know that you have addressed all the issues (regardless of where you addressed them).

Dealing with File-Based Information

BLOBs. You haven't really seen enough of them to hate them yet. Whether that's a "yet" or not largely depends on whether you need to support backward compatibility or not.

Beginning with SQL Server 2005, we have some new data types (varchar(max), nvarchar(max), and varbinary(max)) available that can greatly simplify dealing with BLOBs. When used with a compatible data access model (ADO.NET 2.0 or higher), you can access BLOB data as though it were the same as its smaller base data type (varchar, nvarchar, or varbinary). Unfortunately, for many of you reading this, you'll need to deal with backward compatibility issues, so you'll have to use the older (and even slower) "chunking" method to access your data. Regardless of which access method you're using, BLOBs, or Binary Large Objects, are slow—very slow and big. Hey, did I mention they were slow?

BLOBs are nice in the sense that they let you break the 8K barrier on row size (BLOBs can be up to about 2GB in size). The first problem is that they can be clumsy to use (particularly under the old data types and access methods). Perhaps the larger problem, however, is that they are painfully slow (I know, I'm repeating myself, but I suspect I'm also making a point here). In the race between the BLOB and the tortoise (the sequel to the tortoise and the hare), the BLOB won only after the tortoise stopped for a nap.

Okay, okay, so I've beaten the slow thing into the ground. Indeed, there have been substantial performance improvements in BLOB handling over the years, and the difference is not what it used to be, but at the risk of mentioning it one too many times, BLOBs are still relatively slow.

All right, so now you've heard me say BLOBs are slow and you still need to store large blocks of text or binary information. Normally, you'd do that using a BLOB—and, with the recent performance improvements in BLOB handling, that's probably best—but you do have the option of doing it another way. You can go around the problem by storing things as files instead.

Okay, so by now some of you have to be asking the question of "isn't a database going to be a faster way of accessing data than the file system?" My answer is quite simply—"Usually not."

There are two ways to do this. We'll start with the method that has traditionally been implemented, and then we'll talk about another potential way to do it in the .NET era. I'm going to warn you right up front that, in order to pull typical way of doing this off, you need to be planning for it in your client—this isn't a database server only kind of thing to do. Indeed, you'll be removing most of the work from the database server and putting it into your middle tier and file system.

You can start by looking at what you need to do on the server's file system side. The only thing that you need is to make sure that you have at least one directory to store the information in. Depending on the nature of our application, you may also need to have logic in a middle-tier object that will allow it to create additional directories as needed.

All of the Windows operating systems have limits on the number of files they can store in one directory. With the 64bit operating systems out, the maximum number of files per directory has increased such that the maximum isn't so much the issue, as raw performance (Windows still tends to get very slow in file access as the number of files in a particular directory rises). As such, you still need to think about

how many files you're going to be storing. If it will be many (say, over 500), then you'll want to create a mechanism in the object that stores your BLOB so that it can create new directories either on an as-needed basis, or based on some other logical criteria.

Your business component will be in charge of copying the BLOB information to the file you're going to store it in. If it is already in some defined file format, you're on easy street—just run your language's equivalent to a copy command (with a twist we'll go over shortly), and you're in business. If it is streamed data, then you'll need to put the logic in your component to store the information in a logical format for later retrieval.

One big issue with this implementation is that of security. Since you're storing the information in a file that's outside of SQL Server's realm, it is also outside SQL Server's protection security-wise. Instead, you have to rely on your network security.

There are several "Wow, that's scary!" things that should come to mind for you here. First, if someone's going to read data out of the directory that you're storing all this in, doesn't that mean they can see other files that are stored in there? Yes, it does (if you wanted to get really tricky, you could get around this by changing the Windows security for each file, but it would be very tedious indeed—in the case of a Web application, you would need to do something like implementing a DLL on your Web server). Second, since you'd have to give people rights to copy the file into the directory, wouldn't there be a risk of someone altering the file directly rather than using the database (potentially causing your database to be out of sync with the file)? Absolutely.

The answer to these and the many other questions that you could probably come up with lies in your data access layer (I'm assuming an n-tier approach here). You can, for example, have the access component run under a different security context than the end user. This means that you can create a situation where the user can access their data—but only when they are using the data access component to do it (they don't have any rights to the directory themselves—indeed, they probably don't even know where the files are stored).

So then, where does SQL Server come into play in all this? It keeps track of where you stored the information in question. Theoretically, the reason why you were trying to store this information in the databases in the first place is because it relates to some other information in the row you were going to store it as part of. But instead of saving the actual data in the row in the form of a blob, you will now store a path to the file that you saved. The process for storage will look something like this:

1. Determine the name you're going to store it as.
2. Copy the file to the location that you're going to store it at.
3. Save the full name and path in a varchar (255) (which also happens to be the maximum size for a name in a Windows directory) along with the rest of the data for that row.
4. To retrieve the data, run your query much as you would have if you were going to retrieve the data direction from the table, only this time, retrieve the path to where the actual BLOB data is stored.
5. Retrieve the data from the file system.

In general, this approach will run perhaps twice as fast than if you were using BLOBS. There are, however, some exceptions to the rule of wanting to use this approach:

- The BLOBS you are saving are consistently small (less than 8K) in size.
- The data is text or some format that MS Search has a filter for, and you want to be able to perform full-text searches against it.

If the size of your BLOBS is consistently less than 8K, then the data is able to all fit on one data page. This significantly minimizes the overhead in dealing with your BLOB. While the file system approach is still probably going to be faster, the benefits will be sharply reduced such that it doesn't make as much sense. If you're in this scenario, and speed is everything, then all I can suggest is to experiment.

If you want to perform full-text searches, you're probably going to be better off going ahead and storing the large blocks of text as a TEXT data type (which is a BLOB) in SQL Server. If the text is stored in a binary format that has a MS Search filter available (or you could write your own if you're desperate enough), then you can store the file in an image data type and MS Search will automatically use the filter to build the full-text index. Don't get me wrong; it's still very possible to do full-text searches against the text in the file, but you're going to have to do substantially more coding to keep your relationships intact if you want non-BLOB data from the same functional row. In addition, you're most likely going to wind up having to program your middle tier to make use of index server (which is what supports SQL Server's full-text search also).

If push comes to shove, and you need to make a full-text search against file system-based information, you could take a look at accessing the index server via a query directly. SQL Server can issue remote queries such that you can potentially access any OLEDB datasource. The MS Search service has an OLEDB provider and can be used at the target as a linked server or in an OPENQUERY. The bad news, however, is that performing an index server query against an index server that is not on the same physical box as your SQL Server really doesn't work. The only workaround is to have index server on the system local to SQL Server, but have its catalog files stored on another system. The problem with this is the network chatter during the cataloging process and the fact that it doesn't let you offload the cataloging work (which hurts scalability).

Okay, so that was way #1 (you may recall I said there were two). The second leverages the new .NET assembly architecture that is part of SQL Server 2005. We haven't really gotten to a discussion of .NET integration yet, so we'll keep this fairly high level.

This approach actually leverages many of the same concepts that were used in the middle-tier file access approach. The only real change is in what server or component takes charge of the file access.

With the advent of Common Language Runtime (CLR) integration, we have the ability to create user-defined functions far more complex than those previously possible. As part of that, we have the ability to define table-valued functions that can retrieve data from nearly any base source. Indeed, in Chapter 14 we will take a look at how we can enumerate files in a directory and return them as a table-valued function, but we could just as easily return a varbinary(max) column that contains the file. Under this model, all file access would be performed under whatever network security context we establish for that assembly to run under, but it would only be performed as part of the table-valued function.

Subcategories

Subcategories are a logical construct that provides you another type of relationship (sometimes called a “Supertype” or “SubType” relationship) to work with. On the physical side of the model, a subcategory is implemented using a mix the types of relationships that I’ve already talked about (you’ll see the specifics of that before you’re done).

A subcategory deals with the situation where you have a number of what may first seem like different entities but which share some, although not all, things in common.

I think the best way to get across the concept of a subcategory is to show you one. To do this, we’ll take the example of a document in a company.

A document has a number of attributes that are common to any kind of document. For example:

- Title
- Author
- Date created
- Date last modified
- Storage location

I’m sure there are more. Note that I’m not saying that every document has the same title, rather that every document has a title. Every document has an author (possibly more than one actually, but, for this example, you’ll assume a limit of one). Every document was created on some date. You get the picture—you’re dealing with the attributes of the concept of a document, not any particular instance of a document.

But there are lots of different kinds of documents. From things like legal forms (say your mortgage documents) to office memos, to report cards—there are lots of document types. Still, each of these can still be considered to be a document—or a subcategory of a document. Consider a few examples:

For our first example, we’ll look at a lease. A lease has all the characteristics that we expect to find our in documents category, but it also has information that is particular to a lease. A lease has things like:

- Lessor
- Lessee
- Term (how long the lease is for)
- Rate (how much per month or week)
- Security deposit
- Start date
- Expiration date
- Option (with usually offers an extension at a set price for a set additional term)

The fact that a lease has all of these attributes does not preclude the fact that it is still a document.

Chapter 7

You can come up with a few more examples, and I'll stay with my legal document trend — start with a divorce document. It has attributes such as:

- Plaintiff (the person suing for a divorce)
- Defendant (the plaintiff's spouse)
- Separation date
- Date the plaintiff files for the divorce
- Date the divorce was considered "final"
- Alimony (if any)
- Child support (if any)

You could also have a bill of sale — our bill of sale might include attributes such as:

- Date of sale
- Amount of the sale
- Seller
- Purchaser
- Warranty period (if any)

Again, the fact that divorces and bills of sale both have their own attributes does not change the fact that they are documents.

In each case — leases, divorces, and bills of sale — we have what is really a subcategory of the category of "documents." A document really has little or no meaning without also belonging to a subcategory. Likewise, any instance of a subcategory has little meaning without the parent information that is found only in the supercategory — documents.

Types of Subcategories

Subcategories fall into two separate classifications of their own — exclusive and non-exclusive.

When you refer to a subcategory as simply a "subcategory," then you are usually referring to a subcategory arrangement where you have a record in a table that represents the supercategory (a document in our previous example), and a matching record in at least one of the subcategories.

This kind of subcategory is represented with a symbol that appears rather odd as compared to those you've seen thus far (Figure 7-9):

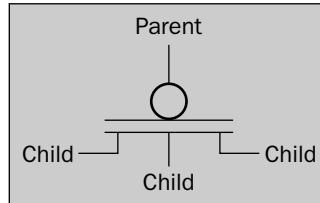


Figure 7-9

Even though there are three subcategories depicted both here and in the document example, don't misconstrue this as being any kind of official limit to the number of subcategories — there isn't one. You could have a single subcategory or 10 of them — it doesn't really make any difference.

Far more common is the situation where you have an exclusive subcategory. An exclusive subcategory works exactly as a category did with only one exception — for every record in the supercategory, there is only one matching record in any of the subcategories. Each subcategory is deemed to be mutually exclusive, so a record to match the supercategory exists as exactly one row in exactly one of the subcategory tables.

The diagramming for an exclusive sub-type looks even a little odder yet (Figure 7-10):

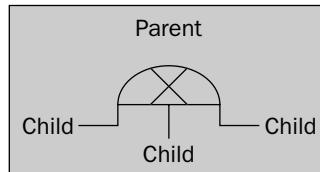


Figure 7-10

Keeping Track of What's What—Implementing Subcategories

The thing that's really cool about subcategories is that they allow you to store all of a similar construct in one place. Before learning this concept, you would have taken one of two approaches to implement our document model:

- Add all of the attributes into one column and just leave the columns null for the information that doesn't fit the specific type of document you're interested in for a given record.
- Have separate tables for each type of document. The columns that are essentially the same between document types would be repeated for each table (each table stores its own copy of the document information as it applies to the records in that particular table).

Using the notion of a subcategory, you can now store all documents, regardless of type, such that they all begin in one place. Any query that you have that is looking for information about all the documents in your system can now run against just one table instead of having to do something like using the UNION operator on three (maybe more, maybe less) different tables. It probably goes without saying, then, that implementing this kind of situation using a subcategory can provide a serious performance enhancement over the other options.

There is a catch though (you knew there would be, right?)—you need to provide some mechanism to point to the rest of the information for that document. Your query of all documents may provide the base information on the specific document that you’re looking for, but when you want the rest of the information for that document (the things that are unique to that document type), then how does your application know which of the subcategory tables to search for the matching record in? To do this, just add a field to your supercategory that indicates what the subcategory is for that record. In our example, you would probably implement another column in our documents table called “`DocumentType`.¹” From that type, you would know which of our other tables to look through for the matching record with more information. Furthermore, you might implement this using a domain table—a table to limit the values in our `DocumentType` column to just those types that you have subcategories for—and a foreign key to that table.

Keep in mind that while what I’m talking about here is the physical storage and retrieval of the data, there is no reason why you couldn’t abstract this using either a sproc or a series of views (or both). For example, you could have a stored procedure call that would pull together the information from the Documents table and then join to the appropriate subcategory.

Oh—for those of you who are thinking, “Wait, didn’t that other text that I read about n-tier architecture say to never use sprocs?,” well, that’s a garbage recommendation in my not so humble opinion (you’ll look more at sprocs in Chapter 11). It’s foolish not to use the performance tools available—just remember to access them only through your data access layer—don’t allow middle-tier or client components to even know your sprocs exist. Follow this advice, and you’ll get better performance, improved overall encapsulation, shorter dev times, and, even with all that, still live within the real theory of a separate data access layer that is so fundamental to n-tier design.

In addition to establishing a pointer to the type of document, you also need to determine whether you’re dealing with a plain subcategory or an exclusive subcategory. In our document example, you have what should be designed as an exclusive subcategory. You may have lots of documents, but you do not have documents that are both a lease and a divorce (a non-exclusive subcategory would allow any mix of our subcategories). Even if you had a lease with a purchase option, the bill of sale would be a separate document created at the time the lease option was exercised.

Figure 7-11 shows an implementation of our logical model.

Okay, so you have an entity called documents. These documents are of a specific type, and that type is limited to a domain—the boundaries of that domain are set by `DocumentType`. In addition, each of the types is represented by its own entity—or subcategory. The symbol in the middle of it all (the half-circle with an “X” through it), tells you that the three subcategories are exclusive in nature (you have one, and only one, for each instance of a document).

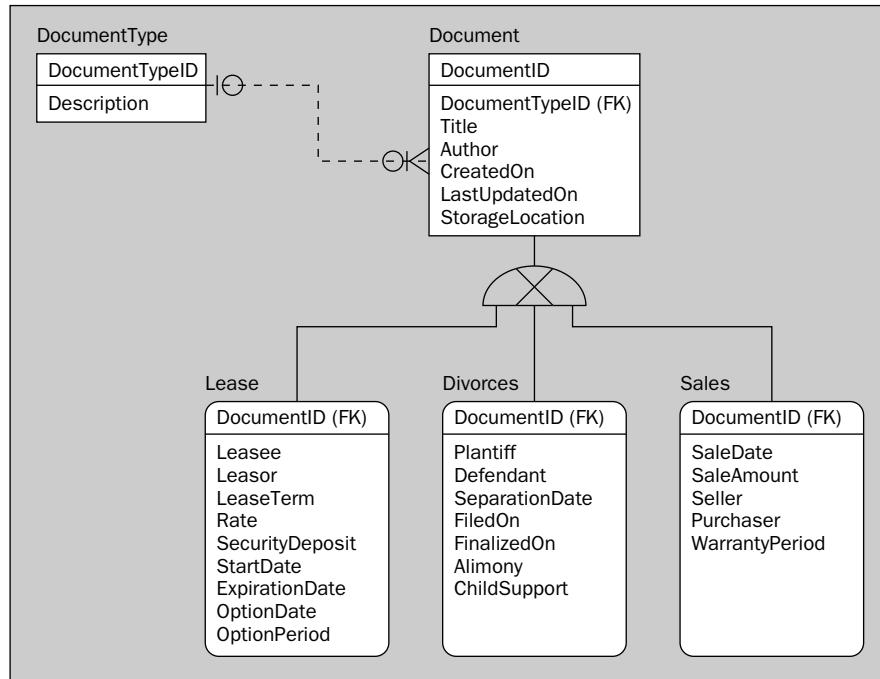


Figure 7-11

This is an excellent place to step back and reflect on what our logical model can do for you. As I discussed earlier in the chapter, our logical model, among other things, provides you with a way to communicate the business rules and requirements of our data. In this case, with a little explanation, someone (a customer perhaps?) can look at this and recognize the concept that you are saying that `Leases`, `Divorces`, and `Sales` are all variations on a theme—that they are really the same thing. This gives the viewer the chance to say, “Wait—no, those aren’t really the same thing.” Or perhaps something like, “Oh, I see—you know, you also have will and power-of-attorney documents—they are pretty much the same, aren’t they?” These are little pieces of information that can save you a bundle of time and money later.

Getting Physical—The Physical Implementation of Subcategories

On the physical side of things, there’s nothing quite as neat and clean as it looks in the logical model. Indeed, all you do for the physical side is implement a series of one to zero or one relationships. You do, however, draw them out as being part of a single, multi-table relationship (Figure 7-12):

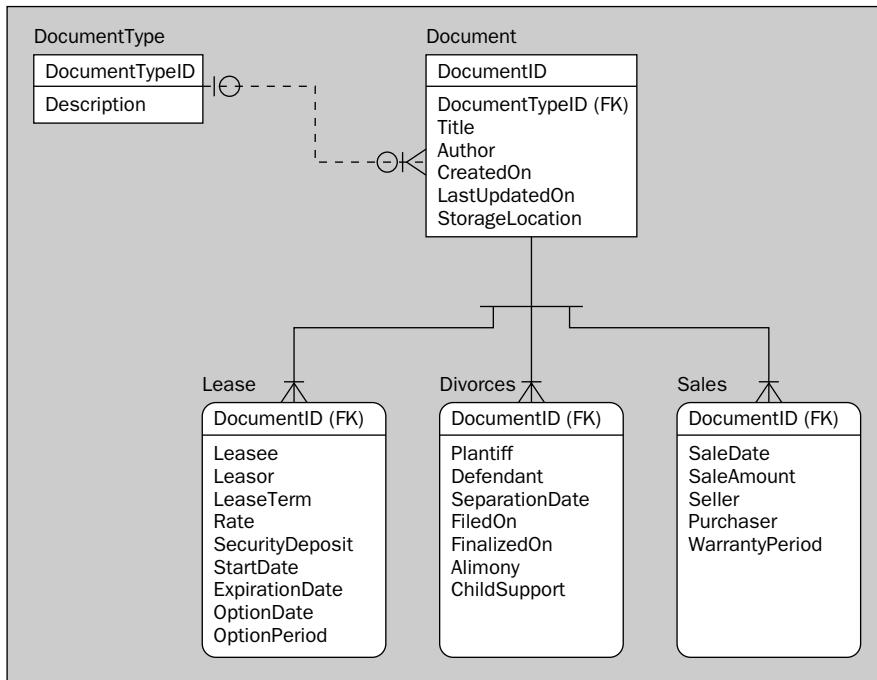


Figure 7-12

The only real trick in the game occurs if you have an exclusive subcategory (which is actually the case much more often than not). In this case, you also need to put some logic into the subcategory tables (in the form of triggers) to ensure that, if any row is to be inserted, there is not already another matching row in one of the other subcategories. For example, you would need to place an insert trigger in `Leases` that queried the `Divorces` and `Sales` tables for records with the same `DocumentID`. If one was found, then the trigger should reject inserted record with an appropriate error message and a `ROLLBACK`.

Adding to Extensibility with Subcategories

Subcategories are one of those concepts that can make a huge difference in the success of your database design. If used when appropriate, you can cut significant time off your queries and significantly simplify pulling together aggregate information for related but different pieces of information. Yet these aren't the only benefits to subcategories.

Subcategories can provide a pathway to making your database more extensible. If you need to add another subcategory, the only queries you need to deal with are those that are specific to your new subcategory. Any of your queries that worked only with the parent table will still work fine—what's more, they'll pick up the information on your new subcategory without any changes!

In short, you're picking up two major scalability benefits:

- ❑ The information for your supercategory (documents in the example) can be scanned from just one table rather than using a UNION operator. This means fewer joins and faster relative query performance—especially as your tables grow larger or you have more and more subcategories.
- ❑ The addition of new subcategories often does not take as much development time as it would have if you were developing the framework for those categories from scratch.

Now, just as with most things, you do need to keep in mind one downside—subcategories can create a bottleneck at the parent table. Every query that you run against all the tables and data involved in the overall set of categories is probably going to need to access the parent table—think about the locking implications there (if you’re new to locking considerations, they are discussed in full in Chapter 12). If you are not careful about your index and query strategies, this can lead to some very bad blocking and/or deadlocking problems. That said, with intelligent planning and query writing, this is usually not a problem. Also, if the sheer size of the parent table becomes a problem, SQL Server now gives us the option of using partitioned tables to scale to larger sizes.

Database Reuse

This is almost never thought of, but you can create databases that facilitate reusability. Why do I say that it’s almost never thought of? Well, just trust me on this—developers think of things like reusable components. Things such as objects to validate credit cards, distribute mail, stream binary information in and out—these are all things that you would immediately think about placing in a repository and using over and over again. For whatever reason, however, databases just don’t seem to get thought of in that way.

Perhaps one reason for this is that databases, by definition, store data. Data is normally thought of as being unique to one company or industry and, most of all, as being private. I’m guessing that you then automatically think of the storage container for that data as also being personal—who knows?

Contrary to popular belief, however, databases can be built to be reusable. Surprisingly, to do this you apply a lot of the same concepts that make code components reusable—most of all compartmentalization and the use of common interfaces.

Just remember to make sure you have a really a good fit before you try to reuse an existing database structure. Much like most things in programming that I’ve seen reuse of, it’s very possible to have your reuse become a situation where you are trying to use the wrong tool for the job, and things can actually become even more expensive than they would have been if you had written things from scratch to begin with.

Candidates for Reusable Databases

The databases that have the best chance at being reusable are those that can be broken up into separate subject areas (much as components are usually broken up into functional groupings). Each subject area is kept as generic as is feasible. An example would be something like an accounting database. You could have separate subject areas that match up with the functional areas in accounting:

- ❑ Purchasing
- ❑ Accounts receivable (which in turn may be broken up into invoicing and cash receipts)
- ❑ Inventory
- ❑ Accounts payable
- ❑ General ledger
- ❑ Cash management

The list could go on. You can also take the approach down to a more granular level and create many, many databases, down to the level of things like persons, commercial entities (ever noticed how similar customers are to vendors), orders—there are lots of things that have base constructs that are used repeatedly. You can roll these up into their own “mini-database,” and then plug them into a larger logical model (tied together using sprocs, views, or other components of your data access layer).

How to Break Things Up

This is where the logical versus physical modeling really starts to show its stuff. When you’re dealing with databases that you’re trying to make reusable, you often have one logical database (that contains all the different subject areas) that contains many physical databases. Sometimes you’ll choose to implement your logical design by referencing each of the physical implementations directly. Other times you may choose an approach that does a better job of hiding the way that you’ve implemented the database—you can create what amounts to a “virtual” database in that it holds nothing but views that reference the data from the appropriate physical database.

Let me digress long enough to point out that this process is essentially just like encapsulation in object-oriented programming. By using the views, you are hiding the actual implementation of your database from the users of the view. This means that you can remove one subject area in your database and replace it with an entirely different design—the only trick in doing this is to map the new design to your views—from that point on, the client application and users are oblivious to the change in implementation.

Breaking things up into separate physical databases and/or virtualizing the database places certain restrictions on you, and many of these restrictions contribute to the idea of being able to separate one subject area from the whole, and reuse it in another environment:

Some of the things to do include:

- ❑ Minimize or eliminate direct references to other functional areas. If you’ve implemented the view approach, connect each physically separate piece of the database to the logical whole only through the views.
- ❑ Don’t use foreign key constraints—where necessary, use triggers instead. Triggers can span databases; foreign key constraints can’t.

The High Price of Reusability

All this reuse comes at a price. Many of the adjustments that you make to your design in order to facilitate reuse have negative performance impacts. Some of these include:

- ❑ Foreign key constraints are faster than triggers overall, but triggers are the only way to enforce referential integrity that crosses database boundaries.
- ❑ Using views means two levels of optimization run on all your queries (one to get at the underlying query and mesh that into your original query, another to sort out the best way to provide the end result)—that's more overhead, and it slows things down.
- ❑ If not using the virtual database approach (one database that has views that map to all the other databases), maintaining user rights across many databases can be problematic.

In short, don't look for things to run as fast unless you're dealing with splitting the data across more servers than you can with the single database model.

Reusing your database can make lots of sense in terms of reduced development time and cost, but you need to balance those benefits against the fact that you may suffer to some degree in the performance category.

De-Normalization

I'm going to keep this relatively short, since this tends to get into fairly advanced concepts, but remember not to get carried away with the normalization of your data.

As I stated early in this chapter, normalization is one of those things that database designers sometimes wear like a cross. It's somehow turned into a religion for them, and they begin normalizing data for the sake of normalization rather than for good things it does to their database. Here are a couple of things to think about in this regard:

- ❑ If declaring a computed column or storing some derived data is going to allow you to run a report more effectively, then by all means put it in. Just remember to take into account the benefit versus the risk. (For example, what if your "summary" data gets out of sync with the data it can be derived from? How will you determine that it happened, and how will you fix it if it does happen?)
- ❑ Sometimes, by including just one (or more) de-normalized column in a table, you can eliminate or significantly cut down the number of joins necessary to retrieve information. Watch for these scenarios—they actually come up reasonably frequently. I've dealt with situations where adding one column to one commonly used base table cut a nine-table join down to just three, and cut the query time by about 90 percent in the process.
- ❑ If you are keeping historical data—data that will largely go unchanged and is just used for reporting—then the integrity issue becomes a much smaller consideration. Once the data is written to a read-only area and verified, you can be reasonably certain that you won't have the kind of "out of sync" problems that is one of the major things that data normalization addresses. At that point, it may be much nicer (and faster) to just "flatten" (de-normalize) the data out into a few tables, and speed things up.

- ❑ The fewer tables that have to be joined, the happier your users who do their own reports are going to be. The user base out there continues to get more and more savvy with the tools they are using. Increasingly, users are coming to their DBA and asking for direct access to the database to be able to do their own custom reporting. For these users, a highly normalized database can look like a maze and become virtually useless. De-normalizing your data can make life much easier for these users.

All that said, if in doubt, normalize things. There is a reason why that is the way relational systems are typically designed. When you err on the side of normalizing, you are erring on the side of better data integrity, and on the side of better performance in a transactional environment.

Partitioning for Scalability

Beginning with SQL Server 2000, you have picked up the marvelous ability to create one logical table from multiple physical tables—partitioned views. That is, the data from one logical table is partitioned such that is stored in a separate well-defined set of physical tables. But the notion of partitioning your data has been around a lot longer than partitioned views have been. Indeed, keeping your main accounting system on one server and your order entry and inventory systems on another is a form of partitioning—you are making sure that the load of handling the two activities is spread across multiple servers. SQL Server 2005 has taken an additional step by adding what are called partitioned tables.

Partitioned tables are a bit different from partitioned views in a way that is implied in their name—they truly remain a table throughout. Whereas a partitioned view could not support some of the functionality found in tables (constraints, defaults, identity columns, etc.), a partitioned table supports all these.

There is, of course, a catch—partitioned tables are limited to just one server (it is a means of separating a table across multiple file groups and, therefore, drive volumes). Partitioned views should still be used when the load is such that you need to span multiple servers. For purposes of this chapter, you’re going to stick with the basic notions of partitioning that apply to both the view and table models.

Regardless of which partitioning method you’re using, the concepts are pretty much the same. You utilize one or more columns in the logical table as a divider to physically separate your data. This allows you to use multiple I/O pathways and even multiple servers to process your query for you. The question of just *how* to partition your data should be a very big one. The tendency is going to be to take the hyper-simplistic approach and just divide things up equally based on the possible values in a partitioning column. This approach may work fine, but it is also a little shortsighted for two big reasons:

- ❑ Data rarely falls into nice, evenly distributed piles. Often, predicting the distribution requires a lot of research and sampling up front.
- ❑ It fails to take into account the way the data will actually be used once stored.

The way that you partition your data does a lot more than determine the volume of data that each partition will receive—much more importantly, it makes a positively huge difference in how well your overall system is going to perform. Keep in mind:

- ❑ Tables rarely live in a bubble. Most of the time you are going to be joining data from any given table with other data in the system—is how the “other” data is partitioned compatible (from a performance perspective)?
- ❑ Network bandwidth tends to be a huge bottleneck in overall system performance—how are you taking that into account when designing your partitions? This is not that big of a deal if dealing with a partitioned table scheme (which will be local to just one server) but can be huge for a portioned view model.

So, with all this in mind, here are a couple of rules for you:

- ❑ If using partitioned view to spread data across servers, keep data that will be used together stored together. That is, if certain tables are going to be used together frequently in queries, then try to partition those tables such that data that is likely to be returned as part of a query will most likely reside on the same server. Obviously, you won’t be able to make that happen 100 percent of the time, but, with careful thought and recognition of how your data gets used, you should find that you can arrange things so that most queries will happen local to just one server. For example, for a given order, all the related order detail rows will be on the same server.
- ❑ When you design your application, you should ideally make it partition aware—that is, you should code the routines that execute the queries such that they know which server most likely has their data. The data may be broken out across multiple machines—wouldn’t it be nice if the database server your application made the request to was the right one from the start, and there was no need for the request to be forwarded to another server?

If you’ve gotten as far as deciding that you need to go with a partitioned system, then you must really have one heck of a load you’re planning on dealing with. How you partition your data is going to have a huge impact on how well your system is going to deal with that load. Remember to take the time to fully plan out your partitioning scheme. After you think you’ve decided what you’re going to do—test! Test! Test!

The SQL Server Diagramming Tools

You can open up SQL Server’s built-in tools by navigating to the Diagrams node of the database you want to build a diagram for (expand your server first, then the database). Some of what you are going to see you’ll find familiar—some of the dialogs are the same as you saw in Chapter 4 when you were creating tables. The SQL Server diagramming tools don’t give you all that many options, so you’ll find that you’ll get to know them fairly quickly.

You can start by creating your first diagram. You can create your new diagram by right-clicking the Diagrams node underneath the AdventureWorks database and choosing the New Database Diagram option.

As you saw back in Chapter 4, you may (if it’s the first time you’ve tried to create a diagram) see a dialog come up warning you that some of the objects needed to support diagramming aren’t in the database and asking if you want to create them—choose yes.

Chapter 7

SQL Server starts you out with the same Add Table dialog (see Figure 7-13) you saw back in Chapter 4—the only thing different are the tables listed.

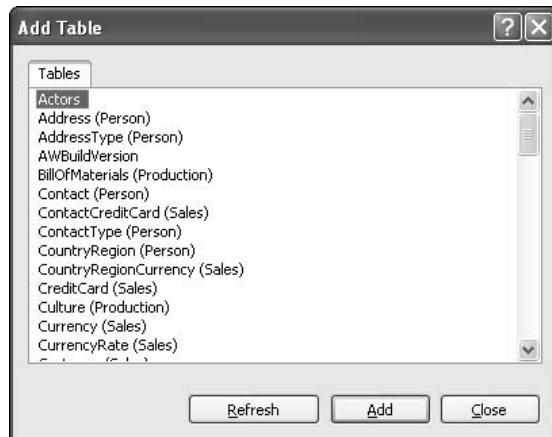


Figure 7-13

Select the following tables (remember to hold down the control key to select more than one table):

- Address
- Customer
- CustomerAddress
- SalesOrderHeader
- SalesOrderDetail

Then click Add. After a brief pause while SQL Server draws all the tables you selected, click the Close button. SQL Server has added our tables to the diagram, as shown in Figure 7-14:

I've rearranged my layout slightly from what SQL Server came up with by default (to make it easier to fit into this book). Depending on your screen resolution, it may be difficult to see the entire diagram at once due to the zoom. To pull more of the tables into view, change the zoom setting in the toolbar.

SQL Server enumerates through each table you have said you want added and analyzed what other objects are associated with those tables. The various other items you see beyond the table itself are some of the many other objects that tie into tables—primary keys, foreign keys.

So, having gotten a start, use this diagram as a launching point for explaining how the diagramming tool works and building a few tables here and there.

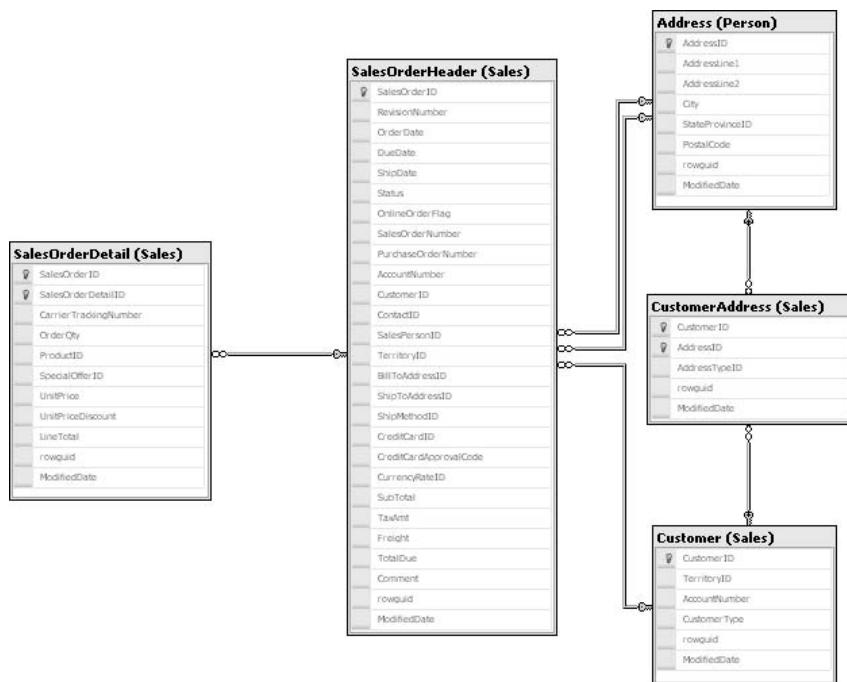


Figure 7-14

Tables

Each table has its own window you can move around. The primary key is shown with the little symbol of a key in the column to the left of the name like the one next to the CustomerID. This is just the default view for the table—you can select from several others that allow you to edit the very make-up of the table. To check out your options for views of a table, right-click on the table that you’re interested in. The default is column names only, but you should also take an interest in the choice of Custom; this or “standard” is what you would use when you want to edit the table from right within the diagram (very nice!).

Adding Tables

You can add a new table to the diagram in one of two ways:

- ❑ If you have a table that already exists in the database (but not in the diagram), but now you want to add it to your diagram, you simply click the “add table” button on the diagramming window’s toolbar, or right-click anywhere in the diagram and choose “Add Table . . .”. You’ll be presented with a list of all the tables in the database—just choose the one that you want to add, and it will appear along with any relationships it has to other tables in the diagram.
- ❑ If you want to add a completely new table, click on the “new table” on the diagramming window’s toolbar or right-click in the diagram and choose “New Table . . .”—you’ll be asked for a name for the new table, and the table will be added to the diagram in “Column Properties” view. Simply edit the properties to have the column names, data types, etc. that you want, and you have a new table in the database.

Let me take a moment to point out a couple of gotchas in this process.

First, don't forget to add a primary key to your table. SQL Server does not automatically do this, nor does it even prompt you (as Access does). This is a somewhat less than intuitive process. To add a primary key, you must select the columns that you want to have in the key. Then right-click and choose "Set Primary Key."

Next, be aware that your new table is not actually added to the database until you choose to save — this is also true of any edits that you make along the way.

Go ahead and quickly add a table to see how this works and set you up for some later examples.

First, right-click anywhere in the diagramming pane, and choose "New Table . . ." You'll be prompted for a table name — call this one CustomerNotes. Now add just three columns as shown in Figure 7-15.

CustomerNotes *		
Column Name	Data Type	Allow Nulls
CustomerID	int	<input type="checkbox"/>
SequenceNo	smalldatetime	<input type="checkbox"/>
NoteText	varchar(MAX)	<input type="checkbox"/>
		<input type="checkbox"/>

Figure 7-15

Notice the asterisk in the title bar for the table — that means there are unsaved changes to this table (specifically, the table has yet to be saved at all). Go ahead and save the diagram, and that will also create the table in the physical database. You now have a table with three NOT NULL columns. There is not, as yet, any primary key for this table (you'll deal with that in our section on adding constraints).

Dropping Tables from Either the Database or Diagram

Dropping tables is a bit confusing since there is a more vague distinction between deleting them from the diagram versus deleting them from the database. You can drop a table from the diagram either of two ways:

- ❑ Select the table and press your delete key.
- ❑ Select the table and choose the "Remove From Diagram" button on the toolbar.

To entirely drop the table from the database, you have three choices:

- ❑ Select the table, and choose Edit>Delete Tables from Database . . . menu choice
- ❑ Select the table, and click the Delete Tables from Database icon on the toolbar
- ❑ Right-click the table header, and choose Delete Tables from Database . . . menu option.

Note that, while deleting a table from the diagram does become permanent until you save the diagram, deleting it from the database happens immediately after you confirm the deletion.

Dealing with Constraints

If you're using the diagram tools at all, you'll want to do more than create just the basic table — you'll want to be able to establish constraints as well. The diagramming tools make these relatively easy.

Primary Keys

This really couldn't be much simpler. To create a primary key, just select the column(s) you want to participate in the key (again, hold down the control key if you need to select multiple columns), right click and select "Set Primary Key," as shown in Figure 7-16.

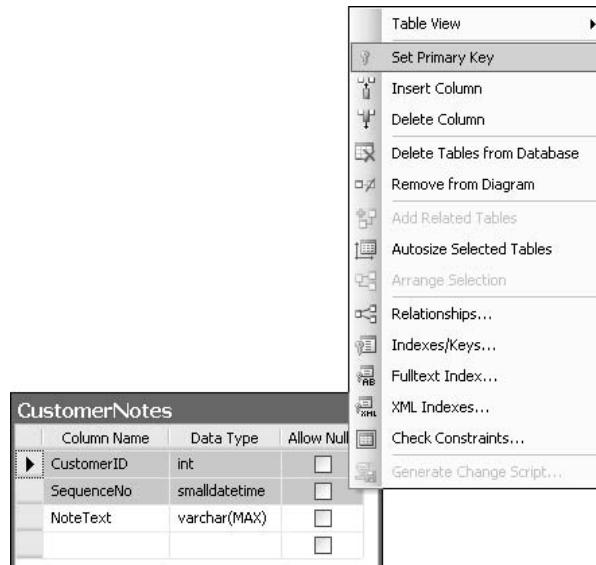


Figure 7-16

I'm adding a primary key to the `CustomerNotes` table you created in the previous section. As you choose the "Set Primary Key" option, you'll see it add a key icon to each of the fields that participate in your column. To change the primary key, just select a new set of columns and again choose "Set Primary Key." To remove it, just choose "Remove Primary Key" from the same menu (it does not show in my figure, because no primary key had been set yet).

Foreign Keys

Foreign keys are nearly as easy as primary keys were — they use a simple drag and drop model.

In our `CustomerNotes` example, you'll notice I used `CustomerID` — this is intended to be the same `CustomerID` that is used elsewhere in the `AdventureWorks` database, so it makes sense that you would want a foreign key to the base table for `CustomerID`'s (`Customer`). To do this, simply click on the `CustomerID` column in the `Customer` table, and drag it onto the `CustomerNotes` table. Management Studio will then give you the dialog in Figure 7-17 to confirm the foreign key you're after.

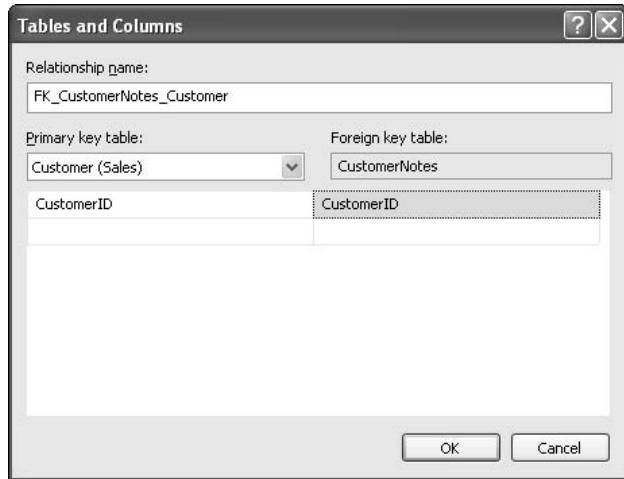


Figure 7-17

From here, you can change what the columns are in both the referenced and referencing tables, and even add additional columns if you need to. Click OK, and you move on to the dialog in Figure 7-18, which allows you to set the other properties that go with a foreign key definition, including such things as cascading actions and whether this foreign key should be propagated to any replicated databases you have out there.

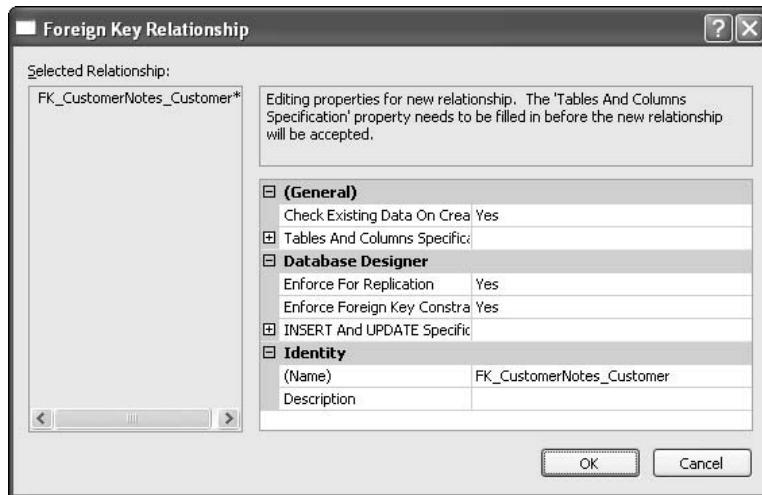


Figure 7-18

To edit the foreign key after you've created it, select it (by clicking on it), and you will see properties in the pane on the right-hand side of the screen.

Note that the properties pane is a dockable window, so it's possible you have moved it away from the default right-hand side.

To delete a foreign key, simply right-click the relationship like and choose “Delete Relationships From Database.”

CHECK Constraints

To work on the CHECK constraints for your table, simply right-click the table and choose Check Constraints. This brings up a dialog that allows you to either create a new constraint or to select from those already defined for the table. After you create a new one or select one of the existing ones, Management Studio brings you up a dialog that is not all that different from that used for foreign keys.

As with when you created tables, you can see the asterisk next to the CK_CustomerNotes name — this lets you know that there are unsaved changes. The primary thing you want to focus on in this dialog is the Expression field — this is where you would enter in the conditions of your constraint.

Do not confuse the Identity box in this dialog with an IDENTITY column — this section of the dialog is only there for providing the name and, optionally, a description of the constraint.

To edit an existing constraint, just change the properties as desired. To remove it, just select the constraint you’re interested in and click the Delete button.

Summary

Database design is a huge concept, and one that has many excellent books dedicated to it as their sole subject. It is essentially impossible to get across every database design notion in just a chapter or two.

In this chapter, you have, however, gotten you off to a solid start. You’ve seen that data is considered normalized when you take it out to the third normal form. At that level, repetitive information has been eliminated and our data is entirely dependent on our key — in short, the data is dependent on: “the key, the whole key, and nothing but the key.” You’ve seen that normalization is, however, not always the right answer — strategic de-normalization of our data can simplify the database for users and speed reporting performance. Finally, you’ve looked at some non-normalization-related concepts in our database design, plus how to make use of the diagramming tools to design our database.

In the next chapter, you will be taking a very close look at how SQL Server stores information and how to make the best use of indexes.

8

SQL Server — Storage and Index Structures

Indexes are a critical part of your database planning and system maintenance. They provide SQL Server (and any other database system for that matter) with additional ways to look up data and take shortcuts to that data's physical location. Adding the right index can cut huge percentages of time off your query executions. Unfortunately, too many poorly planned indexes can actually increase the time it takes for your query to run. Indeed, indexes tend to be one of the most misunderstood objects that SQL Server offers and, therefore, also tend to be one of the most mismanaged.

We will be studying indexes rather closely in this chapter from both a developer's and an administrator's point of view, but in order to understand indexes, we also need to understand how data is stored in SQL Server. For that reason, we will also take a look at SQL Server's data storage mechanism including the index allocation strategies SQL Server employs and internal structures.

SQL Server Storage

Data in SQL Server can be thought of as existing in something of a hierarchy of structures. The hierarchy is pretty simple. Some of the objects within the hierarchy are things that you will deal with directly and will therefore know easily. A few others exist under the cover, and while they can be directly addressed in some cases, they usually are not. Take a look at them one by one.

The Database

Okay—this one is easy. I can just hear people out there saying, "Duh! I knew that." Yes, you probably did, but I point it out as a unique entity here because it is the highest level of the definition of storage (for a given server). This is the highest level that a *lock* can be established at, although you cannot explicitly create a database level lock.

Chapter 8

A lock is something of both a hold and a place marker that is used by the system. We will be looking into locking extensively in Chapter 12, but we will see the lockability of objects within SQL Server discussed in passing as we look at storage.

The File

By default, your database has two files associated with it:

- ❑ The first is the primary physical database file — that's where your data is ultimately stored. This file should be named with a *.mdf extension (this is a recommendation, not a requirement — but I think you'll find doing it in other ways will become confusing over time).
- ❑ The second is something of an offshoot to the database file — the log. We'll dive into the log quite a bit when we deal with transactions and locks in Chapter 12, but you should be aware that it resides in its own file (which should end with a *.ldf extension), and that your database will not operate without it. The log is the serial recording of what's happened to your database since the last time that data was "committed" to the database. The database isn't really your complete set of data. The log isn't your complete set of data. Instead, if you start with the database and "apply" (add in all the activities from) the log, you have your complete set of data.

There is no restriction about where these files are located relative to each other. It is possible (actually, it's even quite desirable) to place each file on a separate physical device. This not only allows for the activity in one file not to interfere with that in the other file, but it also creates a situation where losing the file with the database does not cause you to lose your work — you can restore a backup and then reapply the log (that was safe on the other drive). Likewise, if you lose the drive with the log, you'll still have a valid database up through the time of the last *checkpoint* (checkpoints are fully covered in the chapter on locks and transactions).

The Extent

An *extent* is the basic unit of storage used to allocate space for tables and indexes. It is made up of eight contiguous 64K data pages.

The concept of allocating space based on extents, rather than actual space used, can be somewhat difficult to understand for people used to operating system storage principles. The important points about an extent include:

- ❑ Once an extent is full, the next record will take up not just the size of the record but the size of a whole new extent. Many people who are new to SQL Server get tripped up in their space estimations in part due to the allocation of an extent at a time rather than a record at a time.
- ❑ By pre-allocating this space, SQL Server saves the time of allocating new space with each record.

It may seem like a waste that a whole extent is taken up just because one too many rows were added to fit on the currently allocated extent(s), but the amount of space wasted this way is typically not that much. Still, it can add up — particularly in a highly fragmented environment — so it's definitely something you should keep in mind.

The good news in taking up all this space is that SQL Server skips some of the allocation time overhead. Instead of worrying about allocation issues every time it writes a row, SQL Server deals with additional space allocation only when a new extent is needed.

Don't confuse the space that an extent is taking up with the space that a database takes up. Whatever space is allocated to the database is what you'll see disappear from your disk drive's available space number. An extent is merely how things are, in turn, allocated within the total space reserved by the database.

The Page

Much like an extent is a unit of allocation within the database, a page is the unit of allocation within a specific extent. There are eight pages to every extent.

A page is the last level you reach before you are at the actual data row. Whereas the number of pages per extent is fixed, the number of rows per page is not—that depends entirely on the size of the row, which can vary. You can think of a page as being something of a container for both table and index row data. A row is not allowed to be split between pages.

Figure 8-1 illustrates how data gets put into a page. Notice how, for every row you insert, you have to place the row offset down at the end of the page to indicate where in the page that particular row's data begins.

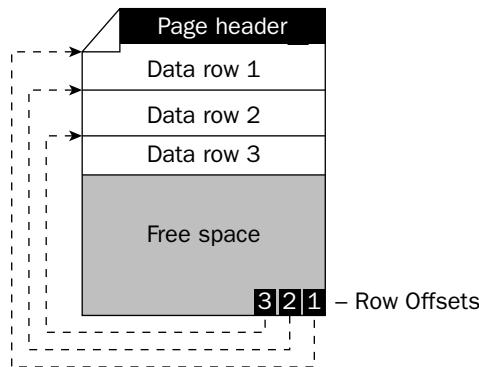


Figure 8-1

There are a number of different *page types*. For purposes of this book, the types we care about are:

- Data
- Index
- Binary Large Object (BLOB) (for Image as well as most Text and Ntext data)
- Global and Shared Global Allocation Map (GAM, or SGAM)
- Page Free Space (PFS)
- Index Allocation Map (IAM)
- Bulk Changed Map
- Differential Changed Map

Data Pages

Data pages are pretty self-explanatory—they are the actual data in your table, with the exception of any BLOB data that is not defined with the *text in row* option. In the case of a row that has a column that contains BLOB data, the regular data is stored in a data page—usually with a 16-byte pointer to where to find the BLOB page that contains the binary information that makes up the BLOB. In a situation where you are using the *text in row* option, the text will be stored with the data if the text is short enough to fit in the available page space (otherwise, you get the 16-byte pointer business again).

Index Pages

Index pages are also pretty straightforward: They hold both the non-leaf and leaf level pages (we'll examine what these are later in the chapter) of a non-clustered index, as well as the non-leaf level pages of a clustered index. These index types will become much clearer as we continue through this chapter.

BLOB Pages

BLOB pages are for storing Binary Large Objects. For SQL Server, these amount to any data stored in an Image field as well as the vast majority of Text and Ntext data. BLOB pages are special as far as data storage pages go, in that they don't have any rows as such. Since a BLOB can be as large as 2GB, they have to be able to go on more than one page—for this portion of things it doesn't matter what the version is. SQL Server will allocate as many pages as it needs in order to store the entire BLOB, but there is no guarantee that these pages will be contiguous—the pages could be located anywhere within the database file(s).

As mentioned before, the connection between the non-blob data for a row and any BLOB related to that row comes in the form of a pointer. The nature of that pointer and how SQL Server navigates to the BLOB data was changed for version 7.0 of SQL Server. In version 6.5 and before, the BLOB pages were put together in a chain—similar to a linked list. In order to find any page that was part of the BLOB, you needed to start at the beginning and navigate through the BLOB page by page. If you were trying to perform some form of text or binary search, this kind of arrangement would be deadly, given that you were forced into a serial scan of the data. Beginning with version 7.0, however, the pages were changed to be organized into a B-Tree structure (which I will discuss fully a little later in the chapter). B-Trees provide more of a branching structure, and, therefore, a more direct path for larger BLOBS. This has made quite a difference in how quickly text operations can be performed.

Even with the significant improvements made in version 7.0, BLOBS are very slow performance-wise, so we will talk about alternative storage methods when we look at advanced design issues later on.

Global Allocation Map, Shared Global Allocation Map, and Page Free Space Pages

Global Allocation Map (GAM), Shared Global Allocation Map (SGAM), and Page Free Space (PFS) page types are involved with figuring out which extents and pages are in use, and which are not. Essentially, these pages store records that indicate where there is space available. Understanding these page types is not really necessary in order to do high-quality development or systems administration, and is beyond the scope of this book. If, however, you're just dying to know about them (or you're having problems with insomnia), then you can find more information on them in the Books Online—just look up GAM in the index.

Bulk Changed Map

Hmmm. How to address this one, since we haven't addressed bulk operations yet

SQL Server has the concept of "bulk operations." Bulk operations are very high-speed changes to the database (usually a mass import of data or a truncation of a table). Part of this speed comes from the idea that they don't "log" every single thing they do. The log is a critical part of the backup and recovery system, and bulk operations mean that unlogged activity (well, it logs that it did an operation, but not the specifics, and so the log can not reconstruct what you did) has occurred in your database.

The Bulk Changed Map—or BCM—is a set of pages that track what extents have been altered via bulk operations. It cares nothing about the specifics of the changes—merely that you messed with that particular extent. Since it knows you altered that extent, it provides more options when you back up your database. More specifically, when backing up the log, you can supplement the log backup with backing up of the physical data in the extents that were affected by bulk operations.

Differential Changed Map

This is nearly the same thing as the Bulk Changed Map, but, instead of tracking only those extents changed by bulk operations, it instead tracks any extents that were changed since the last full backup of your database.

When you request a differential backup, the Differential Changed Map—or DCM—supplies information about what extents need to be backed up. You wind up with a much smaller and faster running (albeit only partial) backup as only those extents that have changed since the prior backup are included.

Page Splits

When a page becomes full, it splits. This means more than just a new page being allocated—it also means that approximately half the data from the existing page is moved to the new page.

The exception to this process is when a clustered index is in use. If there is a clustered index, and the next inserted row would be physically located as the last record in the table, then a new page is created, and the new row is added to the new page without relocating any of the existing data. We will see much more on page splits as we investigate indexes.

ROWS

You will hear much about "Row Level Locking," so it shouldn't be a surprise to hear this term. Rows typically can be up to 8K.

In addition to the limit of 8,060 characters, there is also a maximum of 1,024 columns. In practice, you'll find it very unusual to run into a situation where you run out of columns before you run into the 8,060-character limit. 1,024 gives you an average column width of 8 bytes. For most uses, you'll easily exceed that. The exception to this tends to be in measurement and statistical information—where you have a large number of different things that you are storing numeric samples of. Still, even those applications will find it a rare day when they bump into the 1,024 column count limit.

I did, as you may have noted, use the term *typically* when I mentioned the 8K limit. This limit is based on a row being limited to a single page, and the page having a 8K size, but it can be exceeded in a few

circumstances—specifically, with `varchar(max)` or `varbinary(max)` as well as traditional BLOB data types like image and text. If a row contains too much data in one of these types to fit within the single page, then these special data types know how to make your data span multiple pages (up to 2GB in a single row). In this case, the original row is used to keep track of where the actual data for that column is stored (all other columns are still stored in the original row).

Full-Text Catalogs

These are a separate storage mechanism outside of your normal database. While you can associate a full-text catalog as being the default for a given database, and even, beginning in SQL Server 2005, back up your full-text catalogs together with your database, they are stored completely separately. The internal structures are much different, and the storage is managed by a completely different service (we discuss full text in Chapter 21).

Understanding Indexes

Webster's dictionary defines an index as:

A list (as of bibliographical information or citations to a body of literature) arranged usually in alphabetical order of some specified datum (as author, subject, or keyword).

I'll take a simpler approach in the context of databases and say it's a way of potentially getting to data a heck of a lot quicker. Still, the Webster's definition isn't too bad—even for your specific purposes.

Perhaps the key thing to point out in the Webster's definition is the word *usually* that's in there. The definition of "alphabetical order" changes depending on a number of rules. For example, in SQL Server, we have a number of different collation options available to us. Among these options are:

- ❑ **Binary**—Sorts by the numeric representation of the character (for example, in ASCII, a space is represented by the number 32, the letter *D* is 68, but the letter *d* is 100). Because everything is numeric, this is the fastest option—but unfortunately, it's also not at all the way in which people think, and can also really wreak havoc with comparisons in your `WHERE` clause.
- ❑ **Dictionary order**—This sorts things just as you would expect to see in a dictionary, with a twist—you can set a number of different additional options to determine sensitivity to case, accent, and character set.

It's fairly easy to understand that, if we tell SQL Server to pay attention to case, then *A* is not going to be equal to *a*. Likewise, if we tell it to be case insensitive, then *A* will be equal to *a*. Things get a bit more confusing when we add accent sensitivity—that is, SQL Server pays attention to diacritical marks, and therefore *a* is different from *á*, which is different from *à*. Where many people get even more confused is in how collation order affects not only the equality of data but also the sort order (and, therefore, the way it is stored in indexes).

By way of example, let's look at the equality of a couple of collation options in the following table, and what they do to our sort order and equality information:

Collation Order	Comparison Values	Index Storage Order
Dictionary order, case insensitive, accent insensitive (the default)	A = a = à = á = â = Ä = ä = Å = å	a, A, à, â, á, Ä, ä, Å, å
Dictionary order, case insensitive, accent insensitive, uppercase preference	A = a = à = á = â = Ä = ä = Å = å	A, a, à, â, á, Ä, ä, Å, å
Dictionary order, case sensitive	A ≠ a, Ä ≠ ä, Å ≠ å, a ≠ à ≠ á ≠ â ≠ ä ≠ å A ≠ Ä ≠ Å	A, a, à, á, â, Ä, ä, Å, å

The point here is that what happens in your indexes depends on the collation information you have established for your data. Collation can be set at the database and column level, so you have a fairly fine granularity in your level of control. If you're going to assume that your server is case insensitive, then you need to be sure that the documentation for your system deals with this or you had better plan on a lot of tech support calls—particularly if you're selling outside of the United States. Imagine you're an independent software vendor (ISV) and you sell your product to a customer who installs it on an existing server (which is going to seem like an entirely reasonable thing to the customer), but that existing server happens to be an older server that's set up as case sensitive. You're going to get a support call from one very unhappy customer.

Once the collation order has been set, changing it is very non-trivial (but possible), so be certain of the collation order you want before you set it.

To “B,” or Not to “B”: B-Trees

The concept of a *Balanced Tree*, or *B-Tree*, is certainly not one that was created with SQL Server. Indeed, B-Trees are used in a very large number of indexing systems both in and out of the database world.

A B-Tree simply attempts to provide a consistent and relatively low-cost method of finding your way to a particular piece of information. The *Balanced* in the name is pretty much self-descriptive—a B-Tree is, with the odd exception, self-balancing, meaning that every time the tree branches, approximately half the data is on one side, and half on the other side. The *Tree* in the name is also probably pretty obvious at this point (hint: tree, branch—see a trend here?)—it's there because, when you draw the structure, then turn it upside down, it has the general form of a tree.

A B-Tree starts at the *root node* (another stab at the tree analogy there, but not the last). This root node can, if there is a small amount of data, point directly to the actual location of the data. In such a case, you would end up with a structure that looked something like Figure 8-2.

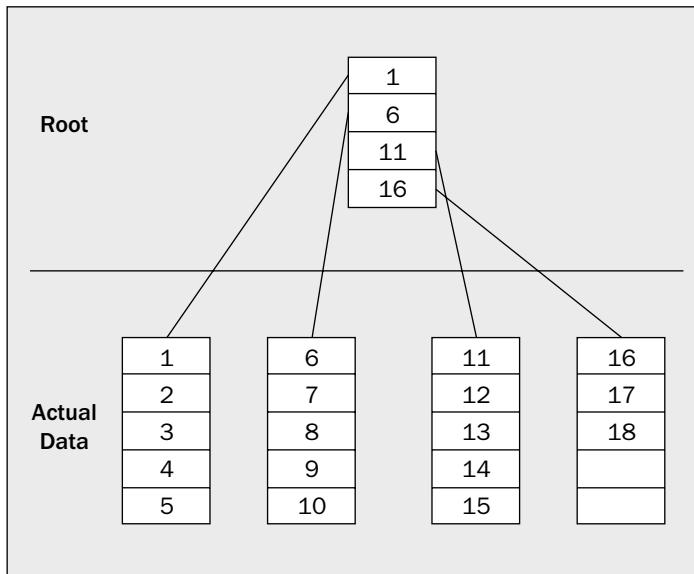


Figure 8-2

So, we start at the root and look through the records until we find the last page that starts with a value less than what we're looking for. We then obtain a pointer to that node, and look through it until we find the row that we want.

In most situations though, there is too much data to reference from the root node, so the root node points at intermediate nodes—or what are called *non-leaf level nodes*. Non-leaf level nodes are nodes that are somewhere in between the root and the node that tells you where the data is physically stored. Non-leaf level nodes can then point to other non-leaf level nodes, or to *leaf level nodes* (last tree analogy reference—I promise). Leaf-level nodes are the nodes where you obtain the real reference to the actual physical data. Much like the leaf is the end of the line for navigating the tree, the node we get to at the leaf level is the end of the line for our index—from here, we can go straight to the actual data node that has our data on it.

As you can see in Figure 8-3, we start with the root node just as before, then move to the node that starts with the highest value that is equal to or less than what we're looking for and is also in the next level down. We then repeat the process—look for the node that has the highest starting value at or below the value for which we're looking. We keep doing this, level by level down the tree, until we get to the leaf level—from there we know the physical location of the data and can quickly navigate to it.

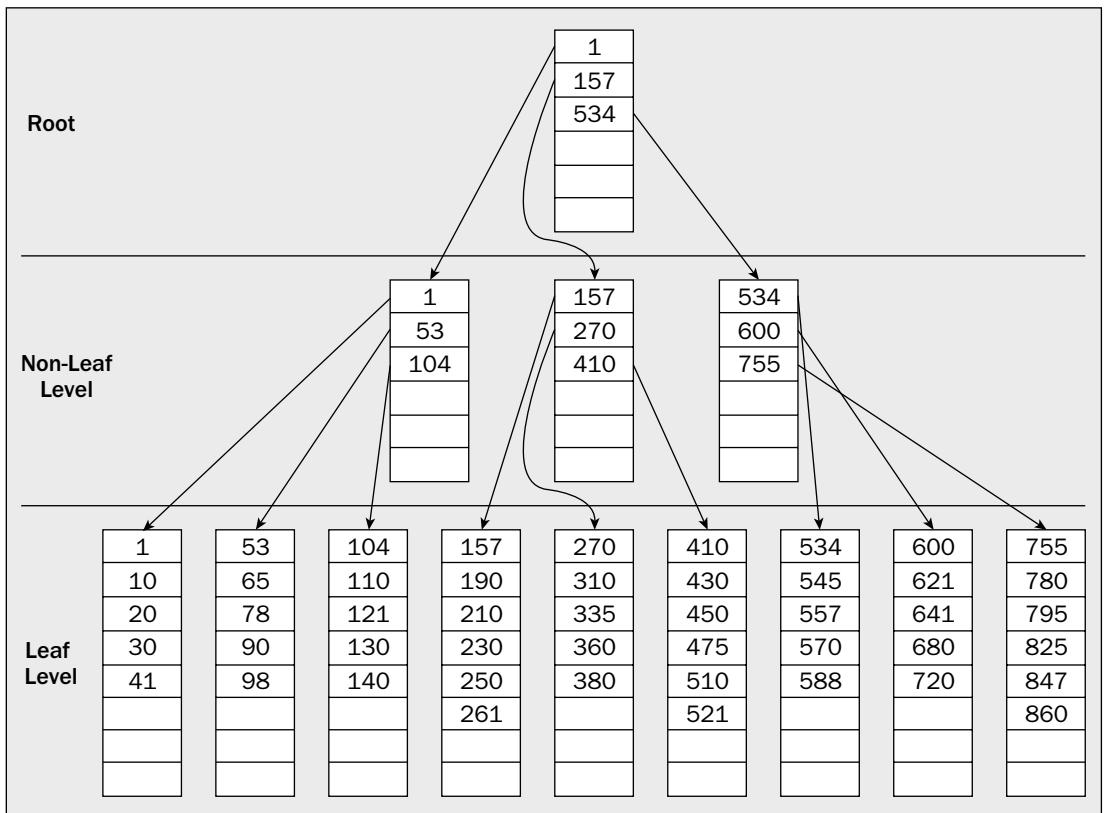


Figure 8-3

Page Splits—A Deeper Look

All of this works quite nicely on the read side of the equation—it's the insert that gets a little tricky. Recall that the *B* in B-Tree stands for *balanced*. You may also recall that I mentioned that a B-Tree is balanced because about half the data is on either side every time you run into a branch in the tree. B-Trees are sometimes referred to as *self-balancing* because the way new data is added to the tree generally prevents them from becoming lopsided.

When data is added to the tree, a node will eventually become full, and will need to split. Because, in SQL Server, a node equates to a page—this is called a *page split*, illustrated in Figure 8-4.

When a page split occurs, data is automatically moved around to keep things balanced. The first half of the data is left on the old page, and the rest of the data is added to a new page—thus you have about a 50-50 split, and your tree remains balanced.

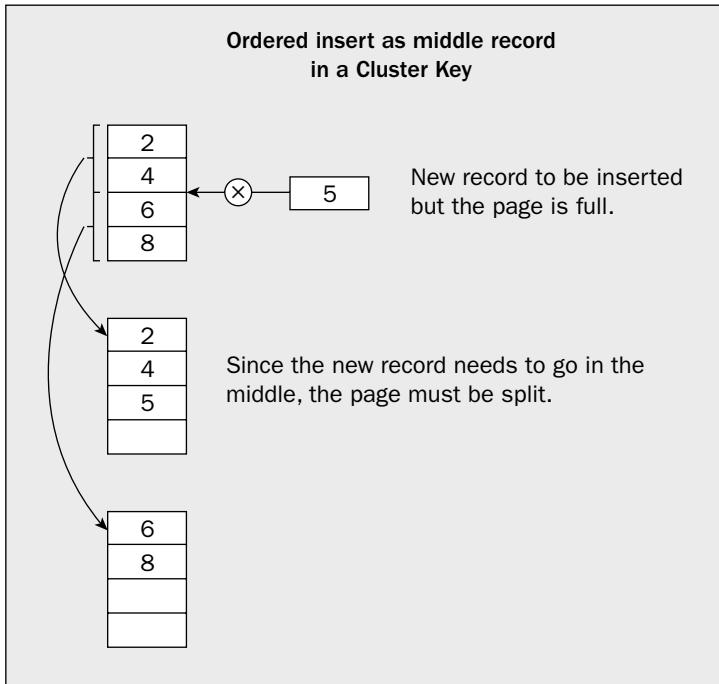


Figure 8-4

If you think about this splitting process a bit, you'll realize that it adds a substantial amount of overhead at the time of the split. Instead of inserting just one page, you are:

- Creating a new page
- Migrating rows from the existing page to the new page
- Adding your new row to one of the pages
- Adding another entry in the parent node

But the overhead doesn't stop there. Since you're in a tree arrangement, you have the possibility for something of a cascading action. When you create the new page (because of the split), you need to make another entry in the parent node. This entry in the parent node also has the potential to cause a page split at that level, and the process starts all over again. Indeed, this possibility extends all the way up to and can even affect the root node.

If the root node splits, then you actually end up creating two additional pages. Because there can be only one root node, the page that was formerly the root node is split into two pages, and becomes a new intermediate level of the tree. An entirely new root node is then created, and will have two entries (one to the old root node, one to the split page).

Needless to say, page splits can have a very negative impact on system performance and are characterized by behavior where your process on the server seems to just pause for a few seconds (while the pages are being split and rewritten).

We will talk about page-split prevention before we're done with this chapter.

While page splits at the leaf level are a common fact of life, page splits at intermediate nodes happen far less frequently. As your table grows, every layer of the index will experience page splits, but, because the intermediate nodes have only one entry for several entries on the next lower node, the number of page splits gets less and less frequent as you move further up the tree. Still, for a split to occur above the leaf level, there must have already been a split at the next lowest level—this means that page splits up the tree are cumulative (and expensive performance-wise) in nature.

SQL Server has a number of different types of indexes (which we will discuss shortly), but they all make use of this B-Tree approach in some way or another. Indeed, they are all very similar in structure thanks to the flexible nature of a B-Tree. Still, we shall see that there are indeed some significant differences, and these can have an impact on the performance of our system.

For a SQL Server index, the nodes of the tree come in the form of pages, but you can actually apply this concept of a root node, the non-leaf level, the leaf level, and the tree structure to more than just SQL Server or even just databases.

How Data Is Accessed in SQL Server

In the broadest sense, there are only two ways in which SQL Server retrieves the data you request:

- Using a table scan
- Using an index

Which method SQL Server will use to run your particular query will depend on what indexes are available, what columns you are asking about, what kind of joins you are doing, and the size of your tables.

Use of Table Scans

A table scan is a pretty straightforward process. When a table scan is performed, SQL Server starts at the physical beginning of the table looking through every row in the table. As it finds rows that match the criteria of your query, it includes them in the result set.

You may hear lots of bad things about table scans, and in general, they will be true. However, table scans can actually be the fastest method of access in some instances. Typically, this is the case when retrieving data from rather small tables. The exact size where this becomes the case will vary widely according to the width of your table and what the specific nature of the query is.

See if you can spot why the use of EXISTS in the WHERE clause of your queries has so much to offer performance-wise where it fits the problem. When you use the EXISTS operator, SQL Server stops as soon as it finds one record that matches the criteria. If you had a million record table, and it found a matching record on the third record, then use of the EXISTS option would have saved you the reading of 999,997 records! NOT EXISTS works in much the same way.

Use of Indexes

When SQL Server decides to use an index, the process actually works somewhat similarly to a table scan, but with a few shortcuts.

During the query optimization process, the optimizer takes a look at all the available indexes and chooses the best one (this is primarily based on the information you specify in your joins and WHERE clause, combined with statistical information SQL Server keeps on index make-up). Once that index is chosen, SQL Server navigates the tree structure to the point of data that matches your criteria and again extracts only the records it needs. The difference is that, since the data is sorted, the query engine knows when it has reached the end of the current range it is looking for. It can then end the query, or move on to the next range of data as necessary.

If you ponder the query topics you've studied thus far (Chapter 6 specifically), you may notice some striking resemblances to how the EXISTS option worked. The EXISTS keyword allowed a query to quit running the instant that it found a match. The performance gains using an index are similar or even better since the process of searching for data can work in a similar fashion—that is, the server is able to know when there is nothing left that's relevant, and can stop things right there. Even better, however, is that by using an index, you don't have to limit yourself to Boolean situations (does the piece of data I was after exist—yes or no?). You can apply this same notion to both the beginning and end of a range—you are able to gather ranges of data with essentially the same benefits that using an index gives to finding data. What's more, you can do a very fast lookup (called a SEEK) of your data rather than hunting through the entire table.

Don't get the impression from my comparing what indexes do for you to the EXISTS operator that indexes replace the EXISTS operator altogether (or vice versa). The two are not mutually exclusive; they can be used together, and often are. I mention them here together only because they have the similarity of being able to tell when their work is done, and quit before getting to the physical end of the table.

Index Types and Index Navigation

Although there are nominally two types of indexes in SQL Server (*clustered* and *non-clustered*), there are actually, internally speaking, three different types:

- ❑ Clustered indexes
- ❑ Non-clustered indexes — which comprise:
 - ❑ Non-clustered indexes on a heap
 - ❑ Non-clustered indexes on a clustered index

The way the physical data is stored varies between clustered and non-clustered indexes. The way SQL Server traverses the B-Tree to get to the end data varies between all three index types.

All SQL Server indexes have leaf level and non-leaf level pages. As I mentioned when I discussed B-Trees, the leaf level is the level that holds the “key” to identifying the record, and the non-leaf level pages are guides to the leaf level.

The indexes are built over either a clustered table (if the table has a clustered index) or what is called a heap (what's used for a table without a clustered index).

- ❑ A *clustered table* is any table that has a clustered index on it. Clustered indexes are discussed in detail shortly, but what they mean to the table is that the data is physically stored in a designated order. Individual rows are uniquely identified through the use of the *cluster key*—the columns that define the clustered index.

This should bring to mind the question of, “What if the clustered index is not unique?” That is, how can a clustered index be used to uniquely identify a row if the index is not a unique index? The answer lies under the covers—SQL Server forces any clustered indexes to be unique—even if you don’t define it that way. Fortunately, it does this in a way that doesn’t change how you use the index. You can still insert duplicate rows if you wish, but SQL Server will add a suffix to the key internally to ensure that the row has a unique identifier.

- ❑ A *heap* is any table that does not have a clustered index on it. In this case, a unique identifier, or row ID (RID) is created based on a combination of the extent, pages, and row offset (places from the top of the page) for that row. A RID is necessary only if there is no cluster key available (no clustered index).

Clustered Indexes

A *clustered index* is unique for any given table—you can have only one per table. You don’t have to have a clustered index, but you’ll find it to be one of the most commonly chosen types as the first index, for a variety of reasons that will become apparent as you look at your index types.

What makes a clustered index special is that the leaf level of a clustered index is the actual data—that is, the data is resorted to be stored in the same physical order that the index sort criteria state. This means that, once you get to the leaf level of the index, you’re done—you’re at the data. Any new record is inserted according to its correct physical order in the clustered index. How new pages are created changes depending on where the record needs to be inserted.

In the case of a new record that needs to be inserted into the middle of the index structure, a normal page split occurs. The last half of the records from the old page are moved to the new page and the new record is inserted into the new or old page as appropriate.

In the case of a new record that is logically at the end of the index structure, a new page is created, but only the new record is added to the new page, as shown in Figure 8-5.

Navigating the Tree

As I’ve indicated previously, even the indexes in SQL Server are stored in a B-Tree. Theoretically, a B-Tree always has half of the remaining information in each possible direction as the tree branches. Take a look at a visualization of what a B-Tree looks like for a clustered index (Figure 8-6).

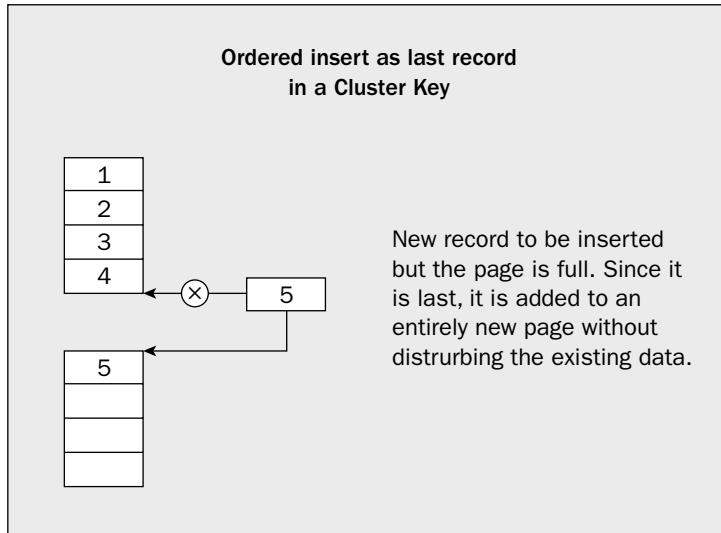


Figure 8-5

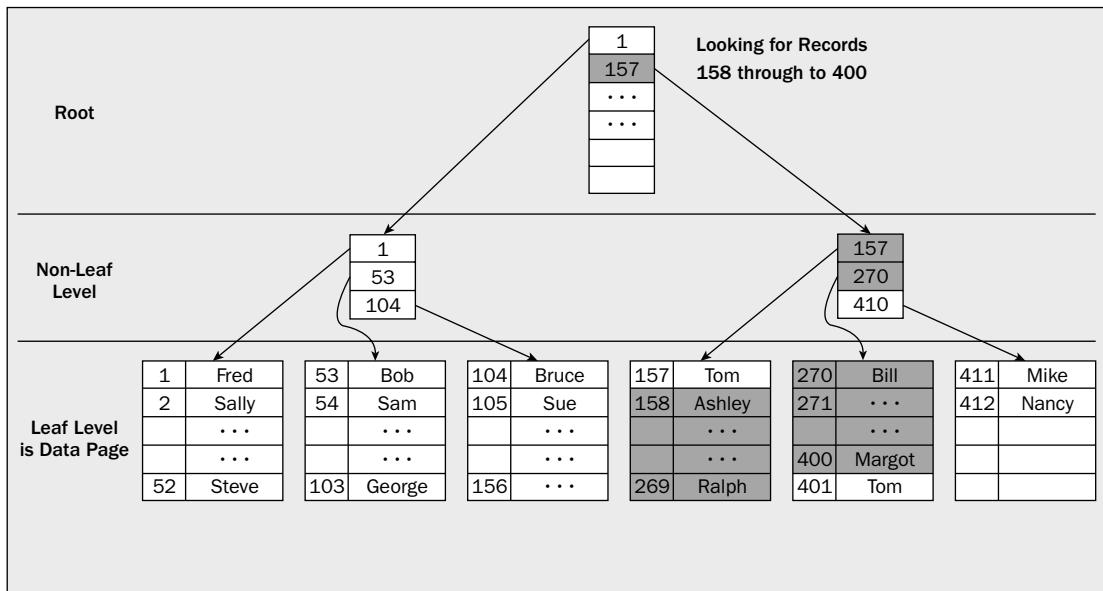


Figure 8-6

As you can see, it looks essentially identical to the more generic B-Trees we discussed earlier in the chapter. In this case, we're doing a range search (something clustered indexes are particularly good at) for numbers 158–400. All we have to do is the following:

Navigate to the first record, and include all remaining records on that page—we know we need the rest of that page because the information from one node up lets us know that we'll also need data from a few other pages. Because this is an ordered list, we can be sure it's continuous—that means if the next page has records that should be included, then the rest of this page must be included. We can just start spewing out data from those pages without having to do the verification side of things.

We start off by navigating to the root node. SQL Server is able to locate the root node based on an entry that is kept in the system table called `sysindexes`.

Every index in your database has an entry in `sysindexes`. This system table is part of your database (as opposed to being in the `master` database) and stores the location information for all the indexes in your database and on which columns they are based. While you can query the table directly, I highly recommend, if looking for information from that table that you instead use the new sys functions—in this case, `sys.indexes`. Since it is a table valued function, you can access it just like a table.

By looking through the page that serves as the root node, we can figure out what the next page we need to examine is (the second page on the second level as we have it drawn here). We then continue the process. With each step we take down the tree, we are getting to smaller and smaller subsets of data.

Eventually, we will get to the leaf level of the index. In the case of our clustered index, getting to the leaf level of the index means that we are also at our desired row(s) and our desired data.

I can't stress enough the importance of the distinction that, with a clustered index, when you've fully navigated the index, you've fully navigated to your data. How much of a performance difference this can make will really show its head as you look at non-clustered indexes—particularly when the non-clustered index is built over a clustered index.

Non-Clustered Indexes on a Heap

Non-clustered indexes on a heap work very similarly to clustered indexes in most ways. They do, however, have a few notable differences:

The leaf level is not the data—instead, it is the level at which you are able to obtain a pointer to that data. This pointer comes in the form of the RID, which, as described earlier in the chapter, is made up of the extent, page, and row offset for the particular row being pointed to by the index. Even though the leaf level is not the actual data (instead, it has the RID), you have only one more step than with a clustered index—because the RID has the full information on the location of the row, you can go directly to the data.

Chapter 8

Don't, however, misunderstand this "one more step" to mean that there's only a small amount of overhead difference, and that non-clustered indexes on a heap will run close to as fast as a clustered index. With a clustered index, the data is physically in the order of the index. That means, for a range of data, when you find the row that has the beginning of your data on it, there's a good chance that the other rows are on that page with it (that is, you're already physically almost to the next record since they are stored together). With a heap, the data is not linked together in any way other than through the index. From a physical standpoint, there is absolutely no sorting of any kind. This means that, from a physical read standpoint, your system may have to retrieve records from all over the file. Indeed, it's quite possible (possibly even probable) that you will wind up fetching data from the same page several separate times—SQL Server has no way of knowing it will have to come back to that physical location because there was no link between the data. With the clustered index, it knows that's the physical sort, and can therefore grab it all in just one visit to the page.

Just to be fair to the non-clustered index on a heap here versus the clustered index, the odds are extremely high that any page that was already read once will still be in the memory cache, and, thus, will be retrieved extremely quickly. Still, it does add some additional logical operations to retrieve the data.

Figure 8-7 shows the same search you did with the clustered index, only with a non-clustered index on a heap this time.

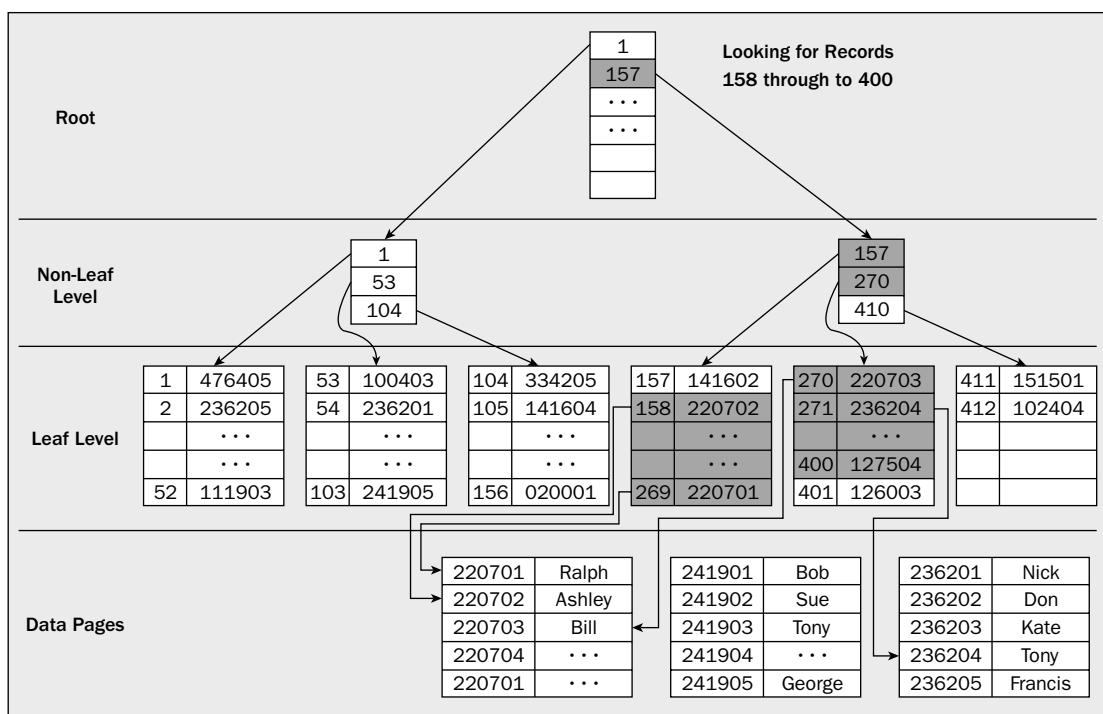


Figure 8-7

Through most of the index navigation, things work exactly as they did before. You start out at the same root node, and you traverse the tree dealing with more and more focused pages until you get to the leaf level of your index. This is where you run into the difference. With a clustered index, you could have stopped right here, but, with a non-clustered index, you have more work to do. If the non-clustered index is on a heap, then you have just one more level to go. You take the Row ID from the leaf-level page, and navigate to it—it is not until that point that you are at your actual data.

Non-Clustered Indexes on a Clustered Table

With *non-clustered indexes on a clustered table*, the similarities continue—but so do the differences. Just as with non-clustered indexes on a heap, the non-leaf level of the index looks pretty much as it did for a clustered index. The difference does not come until you get to the leaf level.

At the leaf level, you have a rather sharp difference from what you've seen with the other two index structures—you have yet another index to look over. With clustered indexes, when you got to the leaf level, you found the actual data. With non-clustered indexes on a heap, you didn't find the actual data, but did find an identifier that let you go right to the data (you were just one step away). With non-clustered indexes on a clustered table, you find the cluster key. That is, you find enough information to go and make use of the clustered index.

You end up with something that looks like Figure 8-8.

What you end up with is two entirely different kinds of lookups.

In the example from your diagram, you start off with a ranged search—you do one single lookup in your index and are able to look through the non-clustered index to find a continuous range of data that meets your criterion (`LIKE 'T%`'). This kind of lookup, where you can go right to a particular spot in the index, is called a *seek*.

The second kind of lookup then starts—the lookup using the clustered index. This second lookup is very fast; the problem lies in the fact that it must happen multiple times. You see, SQL Server retrieved a list from the first index lookup (a list of all the names that start with "T"), but that list doesn't logically match up with the cluster key in any continuous fashion—each record needs to be looked up individually as shown in Figure 8-9.

Chapter 8

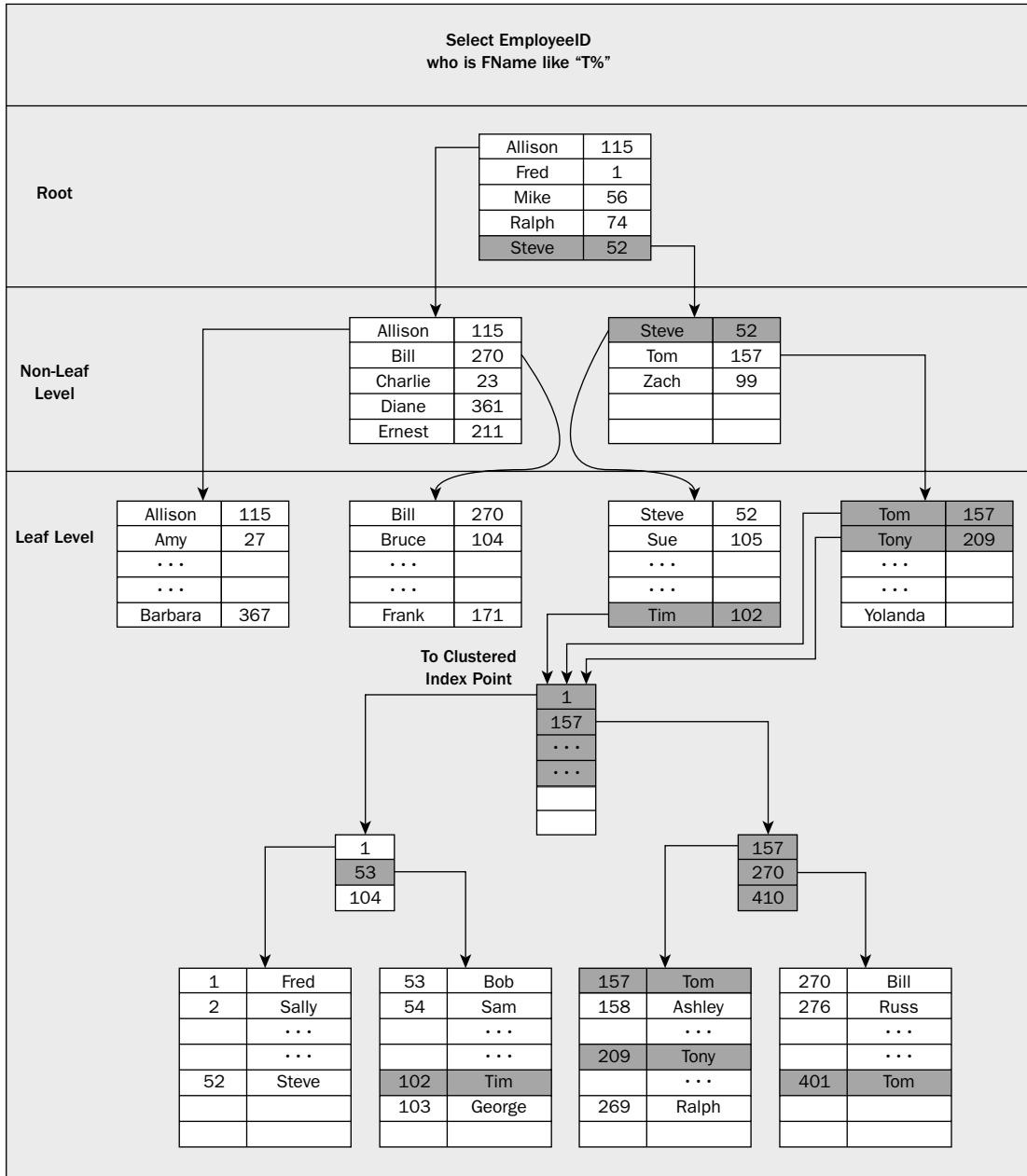


Figure 8-8

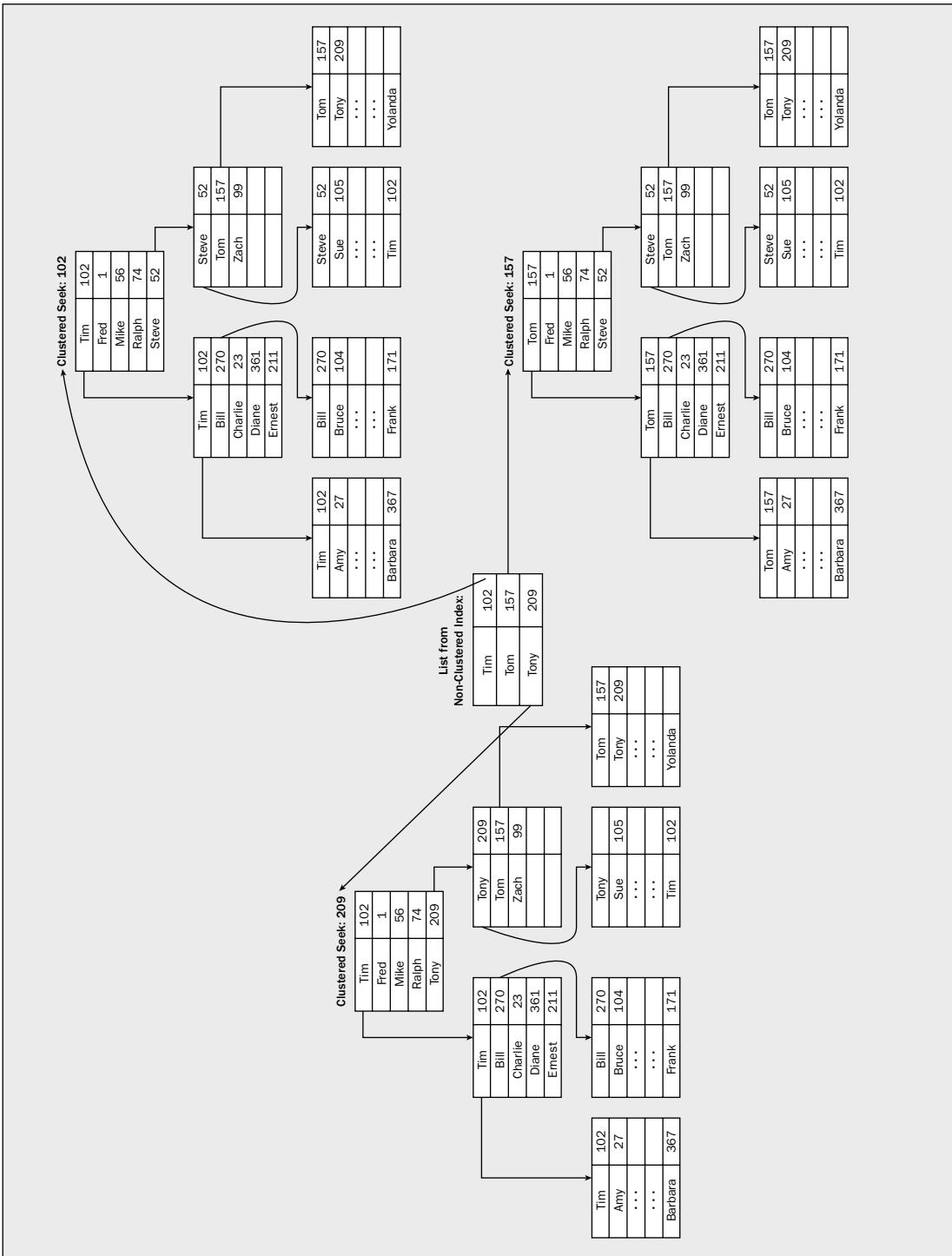


Figure 8-9

Needless to say, this multiple lookup situation introduces more overhead than if you had just been able to use the clustered index from the beginning. The first index search—the one through your non-clustered index—is going to require very few logical reads.

For example, if I have a table with 1,000 bytes per row, and I did a lookup similar to the one in your drawing (say, something that would return 5 or 6 rows); it would take only something to the order of 8–10 logical reads to get the information from the non-clustered index. However, that gets me only as far as being ready to look up the rows in the clustered index. Those lookups would cost approximately 3–4 logical reads *each*, or 15–24 additional reads. That probably doesn't seem like that big a deal at first, but look at it this way:

Logical reads went from 3 minimum to 24 maximum—that's an 800 percent increase in the amount of work that had to be done.

Now expand this thought out to something where the range of values from the non-clustered index wasn't just five or six rows, but five or six thousand, or five or six *hundred* thousand rows—that's going to be a huge impact.

Don't let the extra overhead versus a clustered index scare you—the point isn't meant to scare you away from using indexes, but rather to recognize that a non-clustered index is not going to be as efficient as a clustered index from a read perspective (it can, in some instances, actually be a better choice at insertion time). An index of any kind is usually (there are exceptions) the fastest way to do a lookup. I'll explain what index to use and why later in the chapter.

Creating, Altering, and Dropping Indexes

These work much as they do on other objects such as tables. Take a look at each, starting with the CREATE.

Indexes can be created in two ways:

- Through an explicit CREATE INDEX command
- As an implied object when a constraint is created

Each of these has its own quirks about what it can and can't do, so take a look at each of them individually.

The **CREATE INDEX Statement**

The CREATE INDEX statement does exactly what it sounds like—it creates an index on the specified table or view based on the stated columns.

The syntax to create an index is somewhat drawn out, and introduces several items that I haven't really talked about up to this point:

```
CREATE [UNIQUE] [CLUSTERED|NONCLUSTERED]
INDEX <index name> ON <table or view name>(<column name> [ASC|DESC] [, ...n])
INCLUDE (<column name> [, ...n])
[WITH
[PAD_INDEX = { ON | OFF }]
```

```
[[,] FILLFACTOR = <fillfactor>]
[[,] IGNORE_DUP_KEY = { ON | OFF }]
[[,] DROP_EXISTING = { ON | OFF }]
[[,] STATISTICS_NORECOMPUTE = { ON | OFF }]
[[,] SORT_IN_TEMPDB = { ON | OFF }]
[[,] ONLINE = { ON | OFF }
[[,] ALLOW_ROW_LOCKS = { ON | OFF }
[[,] ALLOW_PAGE_LOCKS = { ON | OFF }
[[,] MAXDOP = <maximum degree of parallelism>
]
[ON {<filegroup> | <partition scheme name> | DEFAULT }]
```

There is legacy syntax available for many of these options, and so you may see that syntax put into use to support prior versions of SQL Server. That syntax is, however, considered deprecated and will be removed at some point—I highly recommend that you stay with the newer syntax where possible.

There is a similar but sufficiently different syntax for creating XML indexes. That will be handled separately at the end of this section.

Loosely speaking, this statement follows the same `CREATE <object type> <object name>` syntax that you've seen plenty of already (and will see even more of). The primary hitch in things is that you have a few intervening parameters that you haven't seen elsewhere.

Just as you'll see with views in the next chapter, you do have to add an extra clause onto your `CREATE` statement to deal with the fact that an index isn't really a standalone kind of object. It has to go together with a table or view, and you need to state the table that your column(s) are "ON."

After the `ON <table or view name>(<column name>)` clause, everything is optional. You can mix and match these options. Many of them are seldom used, but some (such as `FILLFACTOR`) can have a significant impact on system performance and behavior, so take a look at them one by one.

ASC/DESC

These two allow you to choose between an ascending and a descending sort order for your index. The default is `ASC`, which is, as you might guess, ascending order.

A question that might come to mind is why ascending versus descending matters—you see, SQL Server can just look at an index backwards if it needs the reverse sort order. Life is not, however, always quite so simple. Looking at the index in reverse order works just fine if you're dealing with only one column, or if your sort is always the same for all columns—but what if you needed to mix sort orders within an index? That is, what if you need one column to be sorted ascending, but the other descending? Since the indexed columns are stored together, reversing the way you look at the index for one column would also reverse the order for the additional columns. If you explicitly state that one column is ascending, and the other is descending, then you invert the second column right within the physical data—there is suddenly no reason to change the way that you access your data.

As a quick example, imagine a reporting scenario where you want to order your employee list by the hire date, beginning with the most recent (a descending order), but you also want to order by their last name (an ascending order). In previous versions, SQL Server would have to do two operations—one for

Chapter 8

the first column and one for the second. By allowing you to control the physical sort order of your data, you gain flexibility in the way you combine columns.

Generally speaking, you'll want to leave this one alone (again, remember backward compatibility). Some likely exceptions are:

- ❑ You need to mix ascending and descending order across multiple columns.
- ❑ Backward compatibility is not an issue.

INCLUDE

This one is a very sweet new addition with SQL Server 2005. Its purpose is to provide better support for what are called *covered queries*. A query is considered to be “covered” when all of the data the query needs is covered in the index that is being used. If all the data needed is already in the index, then there is no need to go to the actual data page — as soon as it has gotten to the leaf level of the index, it has all it needs and can stop there (saving a bunch of I/O operations).

When you `INCLUDE` columns as opposed to placing them in the `ON` list, SQL Server adds them only at the leaf level of the index. Because each row at the leaf level of an index corresponds to a data row, what you're doing is essentially including more of just the raw *data* in the leaf level of your index. If you think about this, you can probably make the guess that `INCLUDE` really applies only to non-clustered indexes (clustered indexes already *are* the data at the leaf level, so there would be no point).

Why does this matter? Well, as you'll discuss further as the book goes on, SQL Server stops working as soon as it has what it actually needs. So, if while traversing the index, it can find all the data that it needs without continuing on to the actual data row, then it won't bother going to the data row (what would be the point?). By including a particular column in the index, you may “cover” a query that utilizes that particular index at the leaf level and save the I/O associated with using that index pointer to go to the data page.

Careful not to abuse this one! When you INCLUDE columns, you are enlarging the size of the leaf level of your index pages. That means fewer rows will fit per page, and, therefore, more I/O may be required to see the same number of rows. The result may be that you effort to speed up one query may slow down others. To quote an old film from the eighties, “Balance Danielson — balance!” Think about the effects on all parts of your system, not just the particular query you're working on that moment.

WITH

`WITH` is an easy one — it just tells SQL Server that you will indeed be supplying one or more of the options that follow.

PAD_INDEX

In the syntax list, this one comes first — but that will seem odd when you understand what `PAD_INDEX` does. In short, it determines just how full the non-leaf level pages of your index are going to be (as a percentage), when the index is first created. You don't state a percentage on `PAD_INDEX` because it will use whatever percentage is specified in the `FILLCFACTOR` option that follows. Setting `PAD_INDEX = ON` would be meaningless without a `FILLCFACTOR` (which is why it seems odd that it comes first).

FILLFACTOR

When SQL Server first creates an index, the pages are, by default, filled as full as they can be, minus two records. You can set the `FILLFACTOR` to be any value between 1 and 100. This number will be how full your pages are as a percentage, once index construction is completed. Keep in mind, however, that as your pages split, your data will still be distributed 50-50 between the two pages—you cannot control the fill percentage on an ongoing basis other than regularly rebuilding the indexes (something you should do—setting up a maintenance schedule for this is covered in Chapter 24).

You use a `FILLFACTOR` when you need to adjust the page densities. Think about things this way:

- If it's an OLTP system, you want the `FILLFACTOR` to be low.
- If it's an OLAP or other very stable (in terms of changes—very few additions and deletions) system, you want the `FILLFACTOR` to be as high as possible.
- If you have something that has a medium transaction rate and a lot of report type queries against it, then you probably want something in the middle (not too low, not too high).

If you don't provide a value, then SQL Server will fill your pages to two rows short of full, with a minimum of one row per page (for example, if your row is 8,000 characters wide, you can fit only one row per page—so leaving things two rows short wouldn't work).

IGNORE_DUP_KEY

The `IGNORE_DUP_KEY` option is a way of doing little more than circumventing the system. In short, it causes a `UNIQUE` constraint to have a slightly different action from that which it would otherwise have.

Normally, a unique constraint, or unique index, does not allow any duplicates of any kind—if a transaction tried to create a duplicate based on a column that is defined as unique, then that transaction would be rolled back and rejected. Once you set the `IGNORE_DUP_KEY` option, however, you'll get something of a mixed behavior. You will still receive an error message, but the error will be only of a warning level—the record is still not inserted.

This last line—"the record is still not inserted"—is a critical concept from an `IGNORE_DUP_KEY` standpoint. A rollback isn't issued for the transaction (the error is a warning error rather than a critical error), but the duplicate row will have been rejected.

Why would you do this? Well, it's a way of storing unique values, but not disturbing a transaction that tries to insert a duplicate. For whatever process is inserting the would-be duplicate, it may not matter at all that it's a duplicate row (no logical error from it). Instead, that process may have an attitude that's more along the lines of, "Well, as long as I know there's one row like that in there, I'm happy—I don't care whether it's the specific row that I tried to insert or not."

DROP_EXISTING

If you specify the `DROP_EXISTING` option, any existing index with the name in question will be dropped prior to construction of the new index. This option is much more efficient than simply dropping and re-creating an existing index when you use it with a clustered index. If you rebuild an exact match of the existing index, SQL Server knows that it need not touch the non-clustered indexes, while an explicit

drop and create would involve rebuilding all of the non-clustered indexes twice in order to accommodate the different row locations. If you change the structure of the index using `DROP_EXISTING`, the NCIs are rebuilt only once instead of twice. Furthermore, you cannot simply drop and re-create an index created by a constraint, for example, to implement a certain fill factor. `DROP_EXISTING` is a workaround to this.

STATISTICS_NORECOMPUTE

By default, SQL Server attempts to automate the process of updating the statistics on your tables and indexes. By selecting the `STATISTICS_NORECOMPUTE` option, you are saying that you will take responsibility for the updating of the statistics. In order to turn this option off, you need to run the `UPDATE STATISTICS` command, but not use the `NORECOMPUTE` option.

I strongly recommend against using this option. Why? Well, the statistics on your index are what the query optimizer uses to figure out just how helpful your index is going to be for a given query. The statistics on an index are changing constantly as the data in your table goes up and down in volume and as the specific values in a column change. When you combine these two facts, you should be able to see that not updating your statistics means that the query optimizer is going to be running your queries based on out of date information. Leaving the automatic statistics feature on means that the statistics will be updated regularly (just how often depends on the nature and frequency of your updates to the table). Conversely, turning automatic statistics off means that you will either be out of date or you will need to set up a schedule to manually run the `UPDATE STATISTICS` command.

SORT_IN_TEMPDB

This option makes sense only when your `tempdb` is stored on a physically separate drive from the database that is to contain the new index. This is largely an administrative function, so I'm not going to linger on this topic for more than a brief overview of what it is and why it makes sense only when `tempdb` is on a separate physical device.

When SQL Server builds an index, it has to perform multiple reads to take care of the various index construction steps:

- 1.** Read through all the data, constructing a leaf row corresponding to each row of actual data. Just like the actual data and final index, these go into pages for interim storage. These intermediate pages are not the final index pages but rather a holding place to temporarily store things every time the sort buffers fill up.
- 2.** A separate run is made through these intermediate pages to merge them into the final leaf pages of the index.
- 3.** Non-leaf pages are built as the leaf pages are being populated.

If the `SORT_IN_TEMPDB` option is not used, then the intermediate pages are written out to the same physical files that the database is stored in. This means that the reads of the actual data have to compete with the writes of the build process. The two cause the disk heads to move to different places from those the other (read versus write) needs. The result is that the disk heads are constantly moving back and forth — this takes time.

If, on the other hand, `SORT_IN_TEMPDB` is used, then the intermediate pages will be written to `tempdb` rather than the database's own file. If they are on separate physical drives, this means that there is no

competition between the read and write operations of the index build. Keep in mind, however, that this works only if `tempdb` is on a separate physical drive from your database file; otherwise, the change is only in name, and the competition for I/O is still a factor.

If you're going to use `SORT_IN_TEMPDB`, make sure that there is enough space in `tempdb` for large files.

ONLINE

If you set this to `ON`, it forces the table to remain available for general access and does not create any locks that block users from the index and/or table. By default, full index operations will grab the locks (eventually a table lock) it needs to have full and efficient access to the table. The side effect, however, is that your users are blocked out. (Yeah, it's a paradox; you're likely building an index to make the database more usable, but you essentially make the table unusable while you do it.)

Now, you're probably thinking something like: "Oh, that sounds like a good idea—I'll do that every time so my users are unaffected." Poor thinking. Keep in mind that any index construction like that is probably a very highly I/O-intensive operation, so it is affecting your users one way or the other. Now, add that there is a lot of additional overhead required in the index build for it to make sure that it doesn't step on the toes of any of your users. If you let SQL Server have free reign over the table while it's building the index, then the index will be built much faster, and the overall time that the build is affecting your system will be much smaller.

ONLINE index operations are supported only in the Enterprise Edition of SQL Server. You can execute the `index` command with the `ONLINE` directive in other editions, but it will be ignored, so don't be surprised if you use `ONLINE` and find your users still being blocked out by the index operation if you're using a lesser edition of SQL Server.

ALLOW ROW/PAGE LOCKS

This is a longer term directive than `ONLINE` is, and is a very, very advanced topic. For purposes of this book and given how much we've introduced so far on locking, I want to stick with a pretty simple explanation.

Through much of the book thus far I have repeatedly used the term *lock*. As explained early on, this is something of a placeholder to avoid conflicts in data integrity. The `ALLOW` settings you're looking at here are setting directives regarding whether this index will allow those styles of locks or not. This falls under the heading of *extreme* performance tweak.

MAXDOP

This is overriding the system setting for the maximum degree of parallelism for purposes of building this index. Parallelism is not something I talk about in this book, so I'll give you a mini-dose of it here.

In short, the degree of parallelism is how many processes are put to use for one database operation (in this case, the construction of an index). There is a system setting called the max degree of parallelism that allows you to set a limit on how many processors per operation. The `MAXDOP` option in the index creation options allows you to set the degree of parallelism to be either higher or lower than the base system setting as you deem appropriate.

Chapter 8

ON

SQL Server gives you the option of storing your indexes separately from the data by using the ON option. This can be nice from a couple of perspectives:

- ❑ The space that is required for the indexes can be spread across other drives.
- ❑ The I/O for index operations does not burden the physical data retrieval.

There's more to this, but this is hitting the area of *highly* advanced stuff. It is very data and use dependent, and so we'll consider it out of the scope of this book.

Creating XML Indexes

XML indexes are new with SQL Server 2005, and I have to admit that I'm mildly amazed that Microsoft pulled it off. I've known a lot of that team for a very long time now, and I have a lot of confidence in them, but the indexing of something as unstructured as XML has been a problem that many have tried to address, but few have done with any real success. Kudos to the SQL Server team for pulling this one off. Enough gushing though—I want to get down to the business of what XML indexes are all about.

This is another of those “chicken or egg?” things, in that I haven't really looked at XML at all in this book thus far. Still, I consider this more of an index topic than an XML topic. Indeed, the XML create syntax supports all the same options you saw in the previous look at the CREATE statement with the exception of IGNORE_DUP_KEY and ONLINE.

So, for a bit of hyperfast background: Unlike the relational data that you've been looking at thus far, XML tends to be very unstructured data. It utilizes tags to identify data and can be associated with what's called a schema to provide type and validation information to that XML-based data. The unstructured nature of XML requires the notion of “navigating” or “path” information to find a data “node” in a XML document. Now indexes, on the other hand, try to provide very specific structure and order to data—this poses something of a conflict.

You can create indexes on columns in SQL Server that are of type XML. The primary requirements of doing this are:

- ❑ The table containing the XML you want to index *must* have a clustered index on it.
- ❑ A “primary” XML index must exist on the XML data column before you can create “secondary” indexes (more on this in a moment).
- ❑ XML indexes can be created only on columns of XML type (and an XML index is the only kind of index you can create on columns of that type).
- ❑ The XML column must be part of a base table—you cannot create the index on a view.

The Primary XML Index

The first index you create on an XML index must be declared as a “primary” index. When you create a primary index, SQL Server creates a new clustered index that combines the clustered index of the base table with data from whatever XML node you specify.

Secondary XML Indexes

Nothing special here — much like non-clustered indexes point to the cluster key of the clustered index, secondary XML indexes point at primary XML indexes in much the same way. Once you create a primary XML index, you can create up to 248 more XML indexes on that XML column.

Implied Indexes Created with Constraints

I guess I call this one “index by accident.” It’s not that the index shouldn’t be there — it has to be there if you want the constraint that created the index. It’s just that I’ve seen an awful lot of situations where the only indexes on the system were those created in this fashion. Usually, this implies that the administrators and/or designers of the system are virtually oblivious to the concept of indexes.

However, you’ll also find yet another bizarre twist on this one — the situation where the administrator or designer knows how to create indexes but doesn’t really know how to tell what indexes are already on the system and what they are doing. This kind of situation is typified by duplicate indexes. As long as they have different names, SQL Server will be more than happy to create them for you.

Implied indexes are created when one of two constraints is added to a table:

- A PRIMARY KEY
- A UNIQUE constraint (a.k.a. an *alternate key*)

You’ve seen plenty of the CREATE syntax up to this point, so I won’t belabor it — however, it should be noted that all the options except for {CLUSTERED | NONCLUSTERED} and FILLFACTOR are not allowed when creating an index as an implied index to a constraint.

ALTER INDEX

The command ALTER INDEX is somewhat deceptive in what it does. Up until now, ALTER commands have always been about changing the definition of your object. You ALTER tables to add or disable constraints and columns for example. ALTER INDEX is different — it is all about maintenance and zero about structure. If you need to change the make-up of your index, you still need either to DROP and CREATE it or to CREATE and use the index with the DROP_EXISTING=ON option.

As you saw earlier in the chapter, SQL Server gives you an option for controlling just how full your leaf-level pages are, and, if you choose, another option to deal with non-leaf level pages. Unfortunately, these are proactive options — they are applied once, and then you need to reapply them as necessary by rebuilding your indexes and reapplying the options.

In the upcoming section on maintenance, you’ll learn more on the wheres and whys of utilizing this command, but for now take it on faith that you’ll use maintenance commands like ALTER INDEX as part of your regular maintenance routine.

The ALTER INDEX syntax looks like this:

```
ALTER INDEX { <name of index> | ALL }
    ON <table or view name>
    { REBUILD
```

```
[ [ WITH (
    [ PAD_INDEX = { ON | OFF } ]
    | [ , ] FILLFACTOR = <fillfactor>
    | [ , ] SORT_IN_TEMPDB = { ON | OFF }
    | [ , ] IGNORE_DUP_KEY = { ON | OFF }
    | [ , ] STATISTICS_NORECOMPUTE = { ON | OFF }
    | [ , ] ONLINE = { ON | OFF }
    | [ , ] ALLOW_ROW_LOCKS = { ON | OFF }
    | [ , ] ALLOW_PAGE_LOCKS = { ON | OFF }
    | [ , ] MAXDOP = <max degree of parallelism>
        )
    | [ PARTITION = <partition number>
        [ WITH ( <partition rebuild index option>
            [ ,...n ] ) ] ]
    | DISABLE
    | REORGANIZE
        [ PARTITION = <partition number> ]
        [ WITH ( LOB_COMPACTION = { ON | OFF } ) ]
    | SET ( [ ALLOW_ROW_LOCKS= { ON | OFF } ]
        | [ , ] ALLOW_PAGE_LOCKS = { ON | OFF }
        | [ , ] IGNORE_DUP_KEY = { ON | OFF }
        | [ , ] STATISTICS_NORECOMPUTE = { ON | OFF }
        )
    } [ ; ]
```

Several of the options are common to the `CREATE INDEX` command, so I will skip redefining those particular ones here. Beyond that, a fair amount of the `ALTER` specific options are fairly detailed and relate to dealing with things like fragmentation (you'll get to fragmentation and maintenance shortly) or are more DBA oriented and usually used on an ad hoc basis to deal with very specific problems. The core elements here should, however, be part of your regular maintenance planning.

You'll start by looking at a couple of top parameters and then look at the options that are part of your larger maintenance planning needs

Index Name

You can name a specific index if you want to maintain one specific index, or use `ALL` to indicate that you want to perform this maintenance on every index associated with the named table.

Table or View Name

Pretty much just what it sounds like—the name of the specific object (table or view) that you want to perform the maintenance on. Note that it needs to be one specific table (you can feed it a list and say “do all of these please!”).

REBUILD

This is the “industrial-strength” approach to fixing an index. If you run `ALTER INDEX` with this option, the old index is completely thrown away and a new one reconstructed from scratch. The result is a truly optimized index, where every page in both the leaf and non-leaf levels of the index has been reconstructed as you have defined it (either the defaults, or using switches to change things like the fill factor). If the index in question is a clustered index, then the physical data is also reorganized.

By default, the pages will be reconstituted to be full minus two records. Just as with the `CREATE TABLE` syntax, you can set the `FILLFACTOR` to be any value between 0 and 100. This number will be the percent full that your pages are once the database reorganization is complete. Remember though that, as your pages split, your data will still be distributed 50-50 between the two pages—you cannot control the fill percentage on an ongoing basis other than regularly rebuilding the indexes.

Careful on this one. As soon as you kick off a REBUILD, the index you are working on is essentially gone until the rebuild is complete. Any queries that relied on that index may become exceptionally slow (potentially by orders of magnitude). This is the sort of thing you want to test on an offline system first to have an idea how long it's going to take, and then schedule to run in off hours (preferably with someone monitoring it to be sure it's back online when peak hours come along).

This one can have major side effects while it runs, and thus it falls squarely in the domain of the database administrator in my not so humble opinion.

DISABLE

This one does what it says, only in somewhat drastic fashion. It would be nice if all this command did was take your index offline until you decided further what you want to do, but instead it essentially marks the index as unusable. Once an index has been disabled, it must be rebuilt (not reorganized, but rebuilt) before it will be active again.

This is one you're very, very rarely going to do yourself (you would more likely just drop the index)—it is far more likely to happen during a SQL Server upgrade or some other oddball situation.

Yet another BE CAREFUL!!! warning on this one. If you disable the clustered index for your table, it has the effect of disabling the table. The data will remain, but will be inaccessible by all indexes (since they all depend on the clustered index) until you rebuild the clustered index.

REORGANIZE

BINGO!!! from the developer perspective. With `REORGANIZE` you hit much more of a happy medium in life. When you reorganize your index, you get a slightly less complete optimization than you get with a full rebuild, but one that occurs online (users can still utilize the index).

This should, if you're paying attention, bring about the question “What exactly do you mean by ‘slightly less complete?’” Well, `REORGANIZE` works only on the leaf level of your index—non-leaf levels of the index go untouched. This means that you’re not quite getting a full optimization, but, for the lion’s share of indexes, that is not where you real cost of fragmentation is (though it can happen and your mileage may vary).

Given its much lower impact on users, this is usually the tool you’ll want to use as part of your regular maintenance plan. We’ll look into this a bit more later when talking fragmentation.

DROP INDEX

This one returns to most of the simplicity of prior `DROP` statements. The only real trick to it is that, since an index is not a standalone object (it is essentially contained within the definition of a table), you must name not only the index but also the table that it belongs to. The syntax looks like this:

```
DROP INDEX <table name>.<index name>
```

As you can see, there's not really much to it. You can use full four-part naming (I guess it turns into five part if you include the index) if you need to.

Choosing Wisely: Deciding What Index Goes Where and When

By now, you're probably thinking to yourself, "Gee, I'm always going to create clustered indexes!" There are plenty of good reasons to think that way. Just keep in mind that there are also some reasons not to.

Choosing what indexes to include and what not to can be a tough process, and, in case that wasn't enough, you have to make some decisions about what type you want them to be. The latter decision is made simultaneously easier and harder in the fact that you can only have one clustered index. It means that you have to choose wisely to get the most out of it.

Selectivity

Indexes, particularly non-clustered indexes, are primarily beneficial in situations where there is a reasonably high level of *selectivity* within the index. By selectivity, I'm referring to the percentage of values in the column that are unique. The higher the percentage of unique values within a column, the higher the selectivity is said to be, and the greater the benefit of indexing.

If you think back to the sections on non-clustered indexes—particularly the section on non-clustered indexes over a clustered index—you will recall that the lookup in the non-clustered index is really only the beginning. You still need to make another loop through the clustered index in order to find the real data. Even with the non-clustered index on a heap, you still end up with multiple physically separate reads to perform.

If one lookup in your non-clustered index is going to generate multiple additional lookups in a clustered index, then you are probably better off with the table scan. The exponential effect that's possible here is actually quite amazing. Consider that the looping process created by the non-clustered index is not worth it if you don't have somewhere in the area of 90–95 percent uniqueness in the indexed column.

Clustered indexes are substantially less affected by this because, once you're at the start of your range of data—unique or not—you're there. There are no additional index pages to read. Still, more than likely, your clustered index has other things that it could be put to greater use on.

One other exception to the rule of selectivity has to do with foreign keys. If your table has a column that is a foreign key, then, in all likelihood, you're going to benefit from having an index on that column.

Why foreign keys and not other columns? Well, foreign keys are frequently the target of joins with the table they reference. Indexes, regardless of selectivity, can be very instrumental in join performance because they allow what is called a merge join. A merge join obtains a row from each table and compares them to see if they match the join criteria (what you're joining on). Since there are indexes on the related columns in both tables, the seek for both rows is very fast.

The point here is that selectivity is not everything, but it is a big issue to consider. If the column in question is not in a foreign key situation, then it is almost certainly second only to the, "How often will this be used?" question in terms of issues you need to consider.

Watching Costs: When Less Is More

Remember that, while indexes speed up performance when reading data, they are actually very costly when modifying data. Indexes are not maintained by magic. Every time that you make a modification to your data, any indexes related to that data also need to be updated.

When you insert a new row, a new entry must be made into every index on your table. Remember, too, that when you update a row, this is handled as a delete and insert—again, your indexes have to be updated. But wait! There's more! (Feeling like a late night infomercial here.) When you delete records—again, you must update all the indexes too—not just the data. For every index that you create, you are creating one more block of entries that has to be updated.

Notice, by the way, that I said entries plural—not just one. Remember that a B-Tree has multiple levels to it. Every time that you make a modification to the leaf level, there is a chance that a page split will occur, and that one or more non-leaf level pages must also be modified to have the reference to the proper leaf page.

Sometimes—quite often actually—not creating that extra index is the thing to do. Sometimes, the best thing to do is choose your indexes based on the transactions that are critical to your system and use the table in question. Does the code for the transaction have a WHERE clause in it? What column(s) does it use? Is there a sorting required?

Choosing That Clustered Index

Remember that you can have only one, so you need to choose it wisely.

By default, your primary key is created with a clustered index. This is often the best place to have it, but not always (indeed, it can seriously hurt you in some situations), and if you leave things this way, you won't be able to use a clustered index anywhere else. The point here is don't just accept the default. Think about it when you are defining your primary key—do you really want it to be a clustered index?

If you decide that you indeed want to change things—that is, you don't want to declare things as being clustered, just add the NONCLUSTERED keyword when you create your table. For example:

```
CREATE TABLE MyTableKeyExample
(
    Column1  int IDENTITY
        PRIMARY KEY NONCLUSTERED,
    Column2  int
)
```

Chapter 8

Once the index is created, the only way to change it is to drop and rebuild it, so you want to get it set correctly up front.

Keep in mind that, if you change which column(s) your clustered index is on, SQL Server will need to do a complete resorting of your entire table (remember, for a clustered index, the table sort order and the index order are the same). Now, consider a table you have that is 5,000 characters wide and has a million rows in it—that is an awful lot of data that has to be reordered. Several questions should come to mind from this:

- ❑ How long will it take? It could be a long time, and there really isn't a good way to estimate that time.
- ❑ Do I have enough space? Figure that in order to do a resort on a clustered index you will, on average, need an *additional* 1.2 times (the working space plus the new index) the amount of space your table is already taking up. This can turn out to be a very significant amount of space if you're dealing with a large table—make sure you have the room to do it in. All this activity will, by the way, happen in the database itself—so this will also be affected by how you have your maximum size and growth options set for your database.
- ❑ Should I use the `SORT_IN_TEMPDB` option? If `tempdb` is on a separate physical array from your main database and it has enough room, then the answer is probably yes.

The Pros

Clustered indexes are best for queries when the column(s) in question will frequently be the subject of a ranged query. This kind of query is typified by use of the `BETWEEN` statement or the `<` or `>` symbols. Queries that use a `GROUP BY` and make use of the `MAX`, `MIN`, and `COUNT` aggregators are also great examples of queries that use ranges and love clustered indexes. Clustering works well here, because the search can go straight to a particular point in the physical data, keep reading until it gets to the end of the range, and then stop. It is extremely efficient.

Clusters can also be excellent when you want your data sorted (using `ORDER BY`) based on the cluster key.

The Cons

There are two situations where you don't want to create that clustered index. The first is fairly obvious—when there's a better place to use it. I know I'm sounding repetitive here, but don't use a clustered index on a column just because it seems like the thing to do (primary keys are the common culprit here)—be sure that you don't have another column that it's better suited to first.

Perhaps the much bigger no-no use for clustered indexes, however, is when you are going to be doing a lot of inserts in a non-sequential order. Remember that concept of page splits? Well, here's where it can come back and haunt you big time.

Imagine this scenario: You are creating an accounting system. You would like to make use of the concept of a transaction number for your primary key in your transaction files, but you would also like those transaction numbers to be somewhat indicative of what kind of transaction it is (it really helps troubleshooting by your accountants). So, you come up with something of a scheme—you'll place a prefix on all the transactions indicating what sub-system they come out of. They will look something like this:

SQL Server—Storage and Index Structures

ARXXXXXX	Accounts Receivable Transactions
GLXXXXXX	General Ledger Transactions
APXXXXXX	Accounts Payable Transactions

where XXXXXX will be a sequential numeric value.

This seems like a great idea, so you implement it, leaving the default of the clustered index going on the primary key.

At first look, everything about this setup looks fine. You're going to have unique values, and the accountants will love the fact that they can infer where something came from based on the transaction number. The clustered index seems to make sense since they will often be querying for ranges of transaction IDs.

Ah, if only it were that simple. Think about your inserts for a bit. With a clustered index, you originally had a nice mechanism to avoid much of the overhead of page splits. When a new record was inserted that was to go after the last record in the table, then, even if there was a page split, only that record would go to the new page—SQL Server wouldn't try to move around any of the old data. Now you've messed things up though.

New records inserted from the General Ledger will wind up going on the end of the file just fine (GL is last alphabetically, and the numbers will be sequential). The AR and AP transactions have a major problem though—they are going to be doing non-sequential inserts. When AP000025 gets inserted and there isn't room on the page, SQL Server is going to see AR000001 in the table, and know that it's not a sequential insert. Half the records from the old page will be copied to a new page before AP000025 is inserted.

The overhead of this can be staggering. Remember that you're dealing with a clustered index, and that the clustered index is the data. The data is in index order. This means that, when you move the index to a new page, you are also moving the data. Now imagine that you're running this accounting system in a typical OLTP environment (you don't get much more OLTP-like than an accounting system) with a bunch of data-entry people keying in vendor invoices or customer orders as fast as they can. You're going to have page splits occurring constantly, and every time you do, you're going to see a brief hesitation for users of that table while the system moves data around.

Fortunately, there are a couple of ways to avoid this scenario:

- Choose a cluster key that is going to be sequential in its inserting. You can either create an identity column for this or you may have another column that logically is sequential to any transaction entered regardless of system.
- Choose not to use a clustered index on this table. This is often the best option in a situation like that in this example, since an insert into a non-clustered index on a heap is usually faster than one on a cluster key.

Even though I've told you to lean toward sequential cluster keys to avoid page splits, you also have to realize that there's a cost there. Among the downsides of sequential cluster keys are concurrency (two or more people trying to get to the same object at the same time). It's all about balancing out what you want, what you're doing, and what it's going to cost you elsewhere.

This is perhaps one of the best examples of why I have gone into so much depth as to how things work. You need to think through how things are actually going to get done before you have a good feel for what the right index to use (or not to use) is.

Column Order Matters

Just because an index has two columns in, it doesn't mean that the index is useful for any query that refers to either column.

An index is considered for use only if the first column listed in the index is used in the query. The bright side is that there doesn't have to be an exact one-for-one match to every column—just the first. Naturally, the more columns that match (in order), the better, but only the first creates a definite do-not-use situation.

Think about things this way. Imagine that you are using a phone book. Everything is indexed by last name and then first name—does this sorting do you any real good if all you know is that the person you want to call is named Fred? On the other hand, if all you know is that his last name is Blake, the index will still serve to narrow the field for you.

One of the more common mistakes that I see in index construction is to think that one index that includes all the columns is going to be helpful for all situations. Indeed, what you're really doing is storing all the data a second time. The index will totally be ignored if the first column of the index isn't mentioned in the `JOIN`, `ORDER BY`, or `WHERE` clauses of the query.

Dropping Indexes

If you're constantly re-analyzing the situation and adding indexes, don't forget to drop indexes, too. Remember the overhead on inserts—it doesn't make much sense to look at the indexes that you need and not also think about which indexes you do not need. Always ask yourself: "Can I get rid of any of these?"

The syntax to drop an index is pretty much the same as that for dropping a table. The only hitch is that you need to qualify the index name with the table or view it is attached to:

```
DROP INDEX <table or view name>.<index name>
```

And it's gone.

Use the Database Engine Tuning Advisor

It would be my hope that you'll learn enough about indexes not to need the *Database Engine Tuning Advisor*, but it still can be quite handy. It works by taking a workload file, which you generate using the SQL Server Profiler (discussed in Chapter 23), and looking over that information for what indexes will work best on your system.

The Database Engine Tuning Advisor is found as part of the Tools menu of the SQL Server Management Studio. It can also be reached as a separate program item in the Start Menu of Windows. As with most tuning tools, I don't recommend using this tool as the sole way you decide what indexes to build, but it can be quite handy in terms of making some suggestions that you may not have thought of.

Maintaining Your Indexes

As developers, we often tend to forget about our product after it goes out the door. For many kinds of software, that's something you can get away with just fine—you ship it and then you move on to the next product or next release. However, with database-driven projects, it's virtually impossible to get away with. You need to take responsibility for the product well beyond the delivery date.

Please don't take me to mean that you have to go serve a stint in the tech support department—I'm actually talking about something even more important: *maintenance planning*.

There are really two issues to be dealt with in terms of the maintenance of indexes:

- Page splits
- Fragmentation

Both are related to page density and, while the symptoms are substantially different, the troubleshooting tool is the same, as is the cure.

Fragmentation

We've already talked about page splits quite a bit, but we haven't really touched on fragmentation. I'm not talking about the fragmentation that you may have heard of with your O/S files and the defrag tool you use, because that won't help with database fragmentation.

Fragmentation happens when your database grows, pages split, and then data is eventually deleted. While the B-Tree mechanism is really not that bad at keeping things balanced from a growth point of view, it doesn't really have a whole lot to offer as you delete data. Eventually, you may get down to a situation where you have one record on this page, a few records on that page—a situation where many of your data pages are holding only a small fraction of the amount of data that they could hold.

The first problem with this is probably the first you would think about—wasted space. Remember that SQL Server allocates an extent of space at a time. If only one page has one record on it, then that extent is still allocated. In the case of the empty pages in the extent, SQL Server will see those pages as available for reuse in the same table or index, but if, for example, that table or index is decreasing in size, the free pages in the extent will remain unused.

The second problem is the one that is more likely to cause you grief—records that are spread out all over the place cause additional overhead in data retrieval. Instead of just loading up one page and grabbing the 10 rows it requires, SQL Server may have to load 10 separate pages in order to get that same information. It isn't just reading the row that causes effort—SQL Server has to read that page in first. More pages = more work on reads.

That being said, database fragmentation does have its good side—OLTP systems positively love fragmentation. Any guesses as to why? Page splits. Pages that don't have much data in them can have data inserted with little or no fear of page splits.

So, high fragmentation equates to poor read performance, but it also equates to excellent insert performance. As you might expect, this means that OLAP systems really don't like fragmentation, but OLTP systems do.

Identifying Fragmentation

SQL Server has always had a command to help you identify just how full the pages and extents in your database are. With SQL Server 2005, Microsoft has greatly expanded the options and, in particular, the usability of management tools for indexes. We can, then, use the information provided by these commands and tools to make some decisions about what we want to do to maintain our database.

The “old standby” command is actually an option for the *Database Consistency Checker*—or DBCC. This is the command you’re likely to find utilized in some fashion in virtually every installation today and for years to come. This is the pre-2005 way of doing things, and any pre-2005 database installation that had any maintenance going at all utilized it. What’s more, there are tons and tons of articles and “how-tos” on the Web that show you how to use this tool.

Before I get too far into extolling the praises of DBCC SHOWCONTIG, let me be forthright and say I don’t think it’s the best tool anymore. Microsoft added some new functionality in the “sys” system functions that is very, very cool in terms of being able to more specifically query data and manage indexes on a more global level. We will look at the index related side of these shortly, and look at them more fully in the chapter on administration. That said, DBCC has done the job for years, and it is the thing to use if you are monitoring indexes in a server environment that contains pre-SQL Server 2005 installations.

The syntax is pretty simple:

```
DBCC SHOWCONTIG  
[ ({<table name>}|<table id>|<view name>|<view id>)  
[, <index name>|<index id>)]  
[WITH { [ ALL_INDEXES ]  
| [, FAST ]  
| [, TABLERESULTS ]  
| [, ALL_LEVELS } ]  
| [, NO_INFOMSGS ]
```

Some of this is self-describing (such as the table name), but I want address the items beyond the names:

table id/view id/ index id	The is the internal object id for the table, view, or index. In prior versions of SQL Server, DBCC SHOWCONTIG operated solely off this identifier, so you had to look it up using the OBJECT_ID() function prior to making your DBCC call.
ALL_INDEXES	This is one of those “what it sounds like” things. If you specify this option, you can skip providing a specific index, as all indexes will be analyzed and data returned.
FAST	This is about getting a return as fast as possible, and it therefore skips analyzing the actual pages of the index and will output only minimal information.

TABLERESULTS	A very cool feature — this one returns the results as a table rather than text. This means it's much easier to parse the results and take automated actions.
ALL_LEVELS	This really only has one relevance in SQL Server 2005, as what it used to do it now ignores. The relevance? Backward compatibility. Basically, you can include this option and the command will still run, but it won't be any different.
NO_INFOMSGS	This just trims out informational only messages. Basically, if you have any significant errors in your table (error level 11 or higher), then messages will still come through, but error level 10 and lower will be excluded.

As an example, to get the information from the `PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID` index in the `Sales.SalesOrderDetail` table, we could run:

```
USE AdventureWorks
GO

DBCC SHOWCONTIG ('Sales.SalesOrderDetail',
    PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID)
```

Notice the single quotation marks around the table name. These are only required because I'm using two-part naming—if I had only specified the name of the table (`SalesOrderDetail`), then the quotation marks would not have been required. The problem here is that, depending on how your user is set up for use of different schemas or the existence of other tables with the same name in a different schema, leaving out the schema name may generate an error or perform the operation on a different table than you expected.

The output is not really all that self-describing:

```
DBCC SHOWCONTIG scanning 'SalesOrderDetail' table...
Table: 'SalesOrderDetail' (610101214); index ID: 1, database ID: 6
TABLE level scan performed.
- Pages Scanned.....: 1235
- Extents Scanned.....: 155
- Extent Switches.....: 154
- Avg. Pages per Extent.....: 8.0
- Scan Density [Best Count:Actual Count].....: 100.00% [155:155]
- Logical Scan Fragmentation .....: 0.32%
- Extent Scan Fragmentation .....: 20.00%
- Avg. Bytes Free per Page.....: 35.0
- Avg. Page Density (full).....: 99.57%
DBCC execution completed. If DBCC printed error messages, contact your system
administrator.
```

Chapter 8

Some of this is probably pretty straightforward, but the following table will walk you through what everything means:

Stat	What It Means
Pages Scanned	The number of pages in the table (for a clustered index) or index.
Extents Scanned	The number of extents in the table or index. This will be a minimum of the number of pages divided by 8 and then rounded up. The more extents for the same number of pages, the higher the fragmentation.
Extent Switches	The number of times DBCC moved from one extent to another as it traversed the pages of the table or index. This is another one for fragmentation — the more switches it has to make to see the same amount of pages, the more fragmented you are.
Avg. Pages per Extent	The average number of pages per extent. A fully populated extent would have eight.
Scan Density [Best Count: Actual Count]	The best count is the ideal number of extent changes if everything is perfectly linked. Actual count is the actual number of extent changes. Scan density is the percentage found by dividing the best count by the actual count.
Logical Scan Fragmentation	The percentage of pages that are out-of-order as checked by scanning the leaf pages of an index. Only relevant to scans related to a clustered table. An out-of-order page is one for which the next page indicated in the index allocation map (IAM) is different from that pointed to by the next page pointer in the leaf page.
Extent Scan Fragmentation	This one is telling you if an extent is not physically located next to the extent that it is logically located next to. This just means that the leaf pages of your index are not physically in order (though they still can be logically), and just what percentage of the extents this problem pertains to.
Avg. Bytes free per page	Average number of free bytes on the pages scanned. This number can get artificially high if you have large row sizes. For example, if your row size was 4,040 bytes, then every page could only hold one row, and you would always have an average number of free bytes of about 4,020 bytes. That would seem like a lot, but, given your row size, it can't be any less than that.
Avg. Page density (full)	Average page density (as a percentage). This value takes into account row size and is, therefore, a more accurate indication of how full your pages are. The higher the percentage, the better.

Now, the question is how do we use this information once we have it? The answer is, of course, that it depends.

Using the output from our `SHOWCONTIG`, we have a decent idea of whether our database is full, fragmented, or somewhere in between (the latter is, most likely, what we want to see). If we're running an

OLAP system, then seeing our pages full would be great—fragmentation would bring on depression. For an OLTP system, we would want much the opposite (although only to a point).

So, how do we take care of the problem? To answer that, we need to look into the concept of index rebuilding and fillfactors.

DBREINDEX—That Other Way of Maintaining Your Indexes

Earlier in the chapter, we looked at the ALTER INDEX command. This should be your first line command for performing index reorganization and managing your fragmentation levels. While I highly recommend the use of ALTER INDEX moving forward, DBREINDEX is the way things have been done in the past, and, much like DBCC SHOWCONTIG, there is far, far too much code and use out there already for me to just skip it.

DBREINDEX is another DBCC command, and the syntax looks like this:

```
DBCC DBREINDEX ( <'database.owner.table_name'> [, <index name>
[, <fillfactor>]]) [WITH NO_INFOMSGS]
```

Executing this command completely rebuilds the requested index. If you supply a table name with no index name, then it rebuilds all the indexes for the requested table. There is no single command to rebuild all the indexes in a database.

Rebuilding your indexes restructures all the information in those indexes, and reestablishes a base percentage that your pages are full. If the index in question is a clustered index, then the physical data is also reorganized.

As with ALTER INDEX, the pages will, by default, be reconstituted to be full minus two records. Just as with the CREATE TABLE syntax, you can set the FILLFACTOR to be any value between 0 and 100. This number will be the percent full that your pages are once the database reorganization is complete. Remember though that, as your pages split, your data will still be distributed 50-50 between the two pages—you cannot control the fill percentage on an ongoing basis other than regularly rebuilding the indexes.

There is something of an exception on the number matching the percent full that occurs if you use zero as your percentage. It will go to full minus two rows (it's a little deceiving—don't you think?).

We use a FILLFACTOR when we need to adjust the page densities. As we've already discussed, lower page densities (and therefore lower FILLFACTORS) are ideal for OLTP systems where there are a lot of insertions—this helps avoid page splits. Higher page densities are desirable with OLAP systems (fewer pages to read, but no real risk of page splitting due to few to no inserts).

If we wanted to rebuild the index that serves as the primary key for the Order Details table we were looking at earlier with a fill factor of 65, we would issue a DBCC command as follows:

```
DBCC DBREINDEX ('Sales.SalesOrderDetail',
PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID, 65)
```

Chapter 8

We can then re-run the DBCC SHOWCONTIG to see the effect:

```
DBCC SHOWCONTIG scanning 'SalesOrderDetail' table...
Table: 'SalesOrderDetail' (610101214); index ID: 1, database ID: 6
TABLE level scan performed.
- Pages Scanned.....: 1883
- Extents Scanned.....: 236
- Extent Switches.....: 235
- Avg. Pages per Extent.....: 8.0
- Scan Density [Best Count:Actual Count].....: 100.00% [236:236]
- Logical Scan Fragmentation .....: 0.05%
- Extent Scan Fragmentation .....: 0.85%
- Avg. Bytes Free per Page.....: 2809.1
- Avg. Page Density (full).....: 65.29%
DBCC execution completed. If DBCC printed error messages, contact your system
administrator.
```

The big one to notice here is the change in Avg. Page Density. The number didn't quite reach 65 percent because SQL Server has to deal with page and row sizing, but it gets as close as it can.

Several things to note about DBREINDEX and FILLFACTOR:

- ❑ If a FILLFACTOR isn't provided, then the DBREINDEX will use whatever setting was used to build the index previously. If one has never been specified, then the fillfactor will make the page full less two records (which is too full for most situations).
- ❑ If a FILLFACTOR is provided, then that value becomes the default FILLFACTOR for that index.
- ❑ While DBREINDEX can be done live, I strongly recommend against it—it locks resources and can cause a host of problems. At the very least, look at doing it at non-peak hours. Better still, if you're going to do it online, use ALTER INDEX instead and just do a REORGANIZE rather than a rebuild.
- ❑ I've said it before, but it bears repeating: DBREINDEX is now considered deprecated, and you should avoid it in situations where you do not need that backward compatibility (use ALTER INDEX instead).

Summary

Indexes are sort of a cornerstone topic in SQL Server or any other database environment, and are not something to be taken lightly. They can drive your performance successes, but they can also drive your performance failures.

Top-level things to think about with indexes:

- ❑ Clustered indexes are usually faster than non-clustered indexes (one could come very close to saying always, but there are exceptions).
- ❑ Only place non-clustered indexes on columns where you are going to get a high level of selectivity (that is, 95 percent or more of the rows are unique).

SQL Server—Storage and Index Structures

- ❑ All Data Manipulation Language (DML: `INSERT`, `UPDATE`, `DELETE`, `SELECT`) statements can benefit from indexes, but inserts, deletes, and updates (remember, they use a delete and insert approach) are slowed by indexes. The lookup part of a query is helped by the index, but anything that modifies data will have extra work to do (to maintain the index in addition to the actual data).
- ❑ Indexes take up space.
- ❑ Indexes are used only if the first column in the index is relevant to your query.
- ❑ Indexes can hurt as much as they help—know why you’re building the index, and don’t build indexes you don’t need.
- ❑ Indexes can provide structured data performance to your unstructured XML data, but keep in mind that, like other indexes, there is overhead involved.

When you’re thinking about indexes, ask yourself these questions:

Question	Response
Are there a lot of inserts or modifications to this table?	If yes, keep indexes to a minimum. This kind of table usually has modifications done through single record lookups of the primary key — usually, this is the only index you want on the table. If the inserts are non-sequential, think about not having a clustered index.
Is this a reporting table? That is, not many inserts, but reports run lots of different ways?	More indexes are fine. Target the clustered index to frequently used information that is likely to be extracted in ranges. OLAP installations will often have many times the number of indexes seen in an OLTP environment.
Is there a high level of selectivity on the data?	If yes, and it is frequently the target of a <code>WHERE</code> clause, then add that index.
Have I dropped the indexes I no longer need?	If not, why not?
Do I have a maintenance strategy established?	If not, why not?

9

Views

Up to this point, we've been dealing with base objects—objects that have some level of substance of their own. In contrast, this chapter goes virtual (well, mostly) to take a look at views.

Since we're assuming, in this book, that you already know something about SQL Server, I am, as I have in most of the previous chapters, going to rip through the basic part of views in more of a review approach. We will, however, be introducing some more advanced concepts—particularly in the area of partitioned and index views.

Views have a tendency to be used either too much, or not enough—rarely just right. When we're done with this chapter, you should be able to use views to:

- Reduce apparent database complexity for end users
- Prevent sensitive columns from being selected, while still affording access to other important data
- Add additional indexing to your database to speed query performance—even when you're not using the view the index is based on
- Understand and utilize the notion of partitioned tables and the early stages of federated servers (often used for very high-end scalability)

A view is, at its core, really nothing more than a stored query. You can create a simple query that selects from only one table and leaves some columns out, or you can create a complex query that joins several tables and makes them appear as one.

Simple Views

The syntax for a view, in its most basic form, is a combination of a couple of things we've already seen in the book—the basic CREATE statement that we saw back in Chapter 4, plus a SELECT statement like the ones we've used over and over again:

```
CREATE VIEW <view name>
AS
<SELECT statement>
```

The preceding syntax just represents the minimum, of course, but it's still all we need in a large percentage of the situations. The more extended syntax looks like this:

```
CREATE VIEW [<schema name>].<view name> [(<column name list>)]
[WITH [ENCRYPTION] [, SCHEMABINDING] [, VIEW_METADATA]]
AS
<SELECT statement>
WITH CHECK OPTION
```

So, an extremely simple view on the Accounting database we used back in Chapter 5 might look something like:

```
USE Accounting
GO

CREATE VIEW CustomerPhoneList_vw
AS
    SELECT CustomerName, Contact, Phone
    FROM Customers
```

So, when you run:

```
SELECT * FROM CustomerPhoneList_vw
```

You get back exactly the same thing as:

```
SELECT CustomerName, Contact, Phone
FROM Customers
```

You are essentially saying to SQL Server: “Give me all of the rows and columns you get when you run the statement `SELECT CustomerName, Contact, Phone FROM Customers`.”

We've created something of a pass-through situation—that is, our view hasn't really changed anything, but rather just “passed through” a filtered version of the data it was accessing. Think about the uses for this a bit, and you should be able to see how this concept can be utilized to do things like simplify the data for inexperienced users (show them only the columns they care about to keep from confusing them) or to proactively hide sensitive data (such as profit or salary numbers) by granting the user rights to a view that doesn't include that data, but not giving them rights to the underlying table.

Be aware that, by default, there is nothing special done for a view. The view runs just as if it were a query run from the command line — there is no pre-optimization of any kind. This means that you are adding one more layer of overhead between the request for data and the data being delivered. That means that a view is never going to run as fast as if you had just run the underlying SELECT statement directly. That said, views exist for a reason — be it security or simplification for the user — balance your need against the overhead as would seem to fit your particular situation.

Let's take this one step further.

You've already seen how to create a simple view — you just use an easy SELECT statement. How do we filter the results of our queries? With a WHERE clause. Views are no different.

More Complex Views

Perhaps one of the most common uses of views is to flatten data — that is, the removal of complexity that we outlined at the beginning of the chapter. Imagine that we are providing a view for management to make it easier to check on sales information. No offense to managers who are reading this book, but managers who write their own complex queries are still a rather rare breed — even in the information age.

For an example, let's briefly go back to using the AdventureWorks database. Our manager would like to be able to do simple queries that will tell him or her what orders have been placed for what items and how many sold on each order and related pricing information. So, we create a view that he or she can perform very simple queries on — remember that we are creating this one in AdventureWorks:

```
USE AdventureWorks
GO

CREATE VIEW CustomerOrders_vw
AS
SELECT    o.SalesOrderID,
          o.OrderDate,
          od.ProductID,
          p.Name,
          od.OrderQty,
          od.UnitPrice,
          od.LineTotal
FROM Sales.SalesOrderHeader AS o
JOIN    Sales.SalesOrderDetail AS od
        ON o.SalesOrderID = od.SalesOrderID
JOIN    Production.Product AS p
        ON od.ProductID = p.ProductID
```

Now do a SELECT:

```
SELECT *
FROM CustomerOrders_vw
```

Chapter 9

You wind up with a bunch of rows—over 100,000—but you also wind up with information that is far simpler for the average manager to comprehend and sort out. What's more, with not that much training, the manager (or whoever the user might be) can get right to the heart of what he or she is looking for:

```
SELECT ProductID, OrderQty, LineTotal  
FROM CustomerOrders_vw  
WHERE OrderDate = '5/15/2003'
```

The user didn't need to know how to do a four-table join—that was hidden in the view. Instead, he or she needs only limited skill (and limited imagination for that matter) in order to get the job done.

ProductID	OrderQty	LineTotal
791	1	2443.350000
781	1	2071.419600
794	1	2181.562500
798	1	1000.437500
783	1	2049.098200
801	1	1000.437500
784	1	2049.098200
779	1	2071.419600
797	1	1000.437500

(9 row(s) affected)

However, we could make our query even more targeted. Let's say that we want our view to return only yesterday's sales. We'll make only slight changes to our query:

```
USE AdventureWorks  
GO  
  
CREATE VIEW YesterdayCustomerOrders_vw  
AS  
SELECT o.SalesOrderID,  
       o.OrderDate,  
       od.ProductID,  
       p.Name,  
       od.OrderQty,  
       od.UnitPrice,  
       od.LineTotal  
  FROM Sales.SalesOrderHeader AS o  
 JOIN Sales.SalesOrderDetail AS od  
    ON o.SalesOrderID = od.SalesOrderID  
 JOIN Production.Product AS p  
    ON od.ProductID = p.ProductID  
 WHERE CONVERT(varchar(12),o.OrderDate,101) =  
       CONVERT(varchar(12),DATEADD(day,-1,GETDATE()),101)
```

All the dates in the AdventureWorks database are old enough that this view wouldn't return any data, so let's add a row to test it. Execute the following script all at one time:

```
USE AdventureWorks

DECLARE @Ident int

INSERT INTO Sales.SalesOrderHeader
    (CustomerID,
     OrderDate,
     DueDate,
     ContactID,
     BillToAddressID,
     ShipToAddressID,
     ShipMethodID)
VALUES
    (1, DATEADD(day,-1,GETDATE()), GETDATE(), 1, 1, 1, 1)

SELECT @Ident = @@IDENTITY

INSERT INTO Sales.SalesOrderDetail
    (SalesOrderID,
     OrderQty,
     ProductID,
     SpecialOfferID,
     UnitPrice,
     UnitPriceDiscount)
VALUES
    (@Ident, 4, 765, 1, 50, 0)

SELECT 'The OrderID of the INSERTed row is ' + CONVERT(varchar(8),@Ident)
```

Most of what's going on in this script shouldn't be a big mystery for non-beginners, but I'll be explaining all of what is going on here in the chapter on scripts and batches. For now, just trust me that we'll need to run all of this in order for us to have a value in AdventureWorks that will come up for our view. You should see a result from the Management Studio that looks something like this:

```
(1 row(s) affected)

(1 row(s) affected)

-----
The OrderID of the INSERTed row is 75132

(1 row(s) affected)
```

Be aware that some of the messages shown in the preceding code will appear only on the Messages tab if you are using the Management Studio's Results In Grid mode.

The SalesOrderID might vary, but the rest should hold pretty true.

Now let's run a query against our view and see what we get:

```
SELECT SalesOrderID, OrderDate FROM YesterdayCustomerOrders_vw
```

You can see that the 75132 does indeed show up:

```
SalesOrderID OrderDate  
-----  
75132      2006-03-12 15:28:52.903  
  
(1 row(s) affected)
```

Don't get stuck on the notion that your SalesOrderID numbers are going to be the same as mine — these are set by the system (since SalesOrderID is an identity column) and are dependent on just how many rows have already been inserted into the table. As such, your numbers will vary.

Using a View to Change Data — Before INSTEAD OF Triggers

As we've said before, a view works *mostly* like a table does from an in-use perspective (obviously, creating them works quite a bit differently). Now we're going to come across some differences, however.

It's surprising to many, but you can run `INSERT`, `UPDATE`, and `DELETE` statements against a view successfully. There are several things, however, that you need to keep in mind when changing data through a view:

- ❑ If the view contains a join, you won't, in most cases, be able to `INSERT` or `DELETE` data unless you make use of an `INSTEAD OF` trigger. An `UPDATE` can, in some cases (as long as you are only updating columns that are sourced from a single table), work without `INSTEAD OF` triggers, but it requires some planning, or you'll bump into problems very quickly.
- ❑ If your view references only a single table, then you can `INSERT` data using a view without the use of an `INSTEAD OF` trigger provided all the required fields in the table are exposed in the view or have defaults. Even for single-table views, if there is a column not represented in the view that does not have a default value, then you must use an `INSTEAD OF` trigger if you want to allow an `INSERT`.
- ❑ You can, to a limited extent, restrict what is and isn't inserted or updated in a view.

Now, I've already mentioned `INSTEAD OF` triggers several times. `INSTEAD OF` triggers are a special, fairly complex kind of trigger we will look at extensively in Chapter 13. The problem here is that we haven't discussed triggers to any significant extent yet. As is often the case in SQL Server items, we have something of the old chicken versus egg thing going ("Which came first?"). I need to discuss `INSTEAD OF` triggers because of their relevance to views, but we're also not ready to talk about `INSTEAD OF` triggers unless we understand both of the objects (tables and views) that they can be created against.

The way we are going to handle things for this chapter is to address views the way they used to be — before there was such a thing as `INSTEAD OF` triggers. While we won't deal with the specifics of `INSTEAD`

OF triggers in this chapter, we'll make sure we understand when they must be used. We'll then come back and address these issues more fully when we look at INSTEAD OF triggers in Chapter 13.

Having said that, I will provide this bit of context—an Instead Of trigger is a special kind of trigger that essentially runs “instead” of whatever statement caused the trigger to fire. The result is that it can see what you’re statement would have done, and then make decisions right in the trigger about how to resolve any conflicts or other issues that might have come up. It’s very powerful but also fairly complex stuff, which is why we defer it for now.

Dealing with Changes in Views with Joined Data

If the view has more than one table, then using a view to modify data is, in many cases, out—sort of anyway—unless you use an INSTEAD OF trigger. Since it creates some ambiguities in the key arrangements, Microsoft locks you out by default when there are multiple tables. To resolve this, you can use an INSTEAD OF trigger to examine the altered data and explicitly tell SQL Server what you want to do with it.

Required Fields Must Appear in the View or Have the Default Value

By default, if you are using a view to insert data (there must be a single table SELECT in the underlying query or at least you must limit the insert to affecting just one table and have all required columns represented), then you must be able to supply some value for all required fields (fields that don't allow NULLS). Note that by “supply some value” I don't mean that it has to be in the SELECT list—a default covers the bill rather nicely. Just be aware that any columns that do not have defaults and do not accept NULL values will need to appear in the view in order to perform INSERTS through the view. The only way to get around this is—you guessed it—with an INSTEAD OF trigger.

Limit What's Inserted into Views — WITH CHECK OPTION

The WITH CHECK OPTION is one of those lesser-known to almost completely unknown features in SQL Server. The rules are simple—in order to update or insert data using the view, the resulting row must qualify to appear in the view results. Restated, the inserted or updated row must meet any WHERE criterion that's used in the SELECT statement that underlies your view.

Editing Views with T-SQL

The main thing to remember when you edit views with T-SQL is that you are completely replacing the existing view. The only differences between using the ALTER VIEW statement and the CREATE VIEW statement are:

- ALTER VIEW expects to find an existing view, whereas CREATE doesn't.
- ALTER VIEW retains any permissions that have been established for the view.
- ALTER VIEW retains any dependency information.

The second of these is the biggie. If you perform a DROP and then use a CREATE, you have *almost* the same effect as using an ALTER VIEW statement. The problem is that you will need to entirely reestablish your permissions on who can and can't use the view.

Dropping Views

It doesn't get much easier than this:

```
DROP VIEW <view name>, [<view name>, [ . . .n ] ]
```

And it's gone.

Auditing: Displaying Existing Code

What do you do when you have a view, but you're not sure what it does? The first option should be easy at this point—just go into the Management Studio as if you're going to edit the view. Go to the Views sub-node, select the view you want to edit, right-click, and choose Modify View. You'll see the code behind the view complete with color-coding.

Unfortunately, we don't always have the option of having the Management Studio around to hold our hand through this stuff (we may be using a lighter-weight tool of some sort). The bright side is that we have two ways of getting at the actual view definition:

- sp_helptext
- The syscomments system table

Using `sp_helptext` is highly preferable, because when new releases come out, it will automatically be updated for changes to the system tables.

Let's run `sp_helptext` against one of the supplied views in the AdventureWorks database—`vStateProvinceCountryRegion`:

```
EXEC sp_helptext 'Person.vStateProvinceCountryRegion'
```

Note the quotes. This is because this stored proc expects only one argument, and the period is a delimiter of sorts—if you pass `Person.vStateProvinceCountryRegion` in without the quotes, it sees the period and isn't sure what to do with it and therefore errors out.

SQL Server obliges us with the code for the view:

Text

```
CREATE VIEW [Person].[vStateProvinceCountryRegion]
WITH SCHEMABINDING
AS
SELECT
    sp.[StateProvinceID]
    ,sp.[StateProvinceCode]
    ,sp.[IsOnlyStateProvinceFlag]
    ,sp.[Name] AS [StateProvinceName]
    ,sp.[TerritoryID]
    ,cr.[CountryRegionCode]
```

```
,cr.[Name] AS [CountryRegionName]
FROM [Person].[StateProvince] sp
INNER JOIN [Person].[CountryRegion] cr
ON sp.[CountryRegionCode] = cr.[CountryRegionCode];
```

Now let's try it the other way—using `syscomments`. Beyond the compatibility issues with using system tables, using `syscomments` (and most other system tables for that matter) comes with the extra added hassle of everything being coded in object IDs.

Object IDs are SQL Server's internal way of keeping track of things. They are integer values rather than the names that you're used to for your objects. In general, they are outside the scope of this book, but it is good to realize they are there, as you will find them used by scripts you may copy from other people or just bump into them later in your SQL endeavors.

Fortunately, you can get around this by joining to the `sysobjects` table:

```
SELECT sc.text
FROM sys.syscomments sc
JOIN sys.objects so
    ON sc.id = so.object_id
JOIN sys.schemas ss
    ON so.schema_id = ss.schema_id
WHERE so.name = 'vStateProvinceCountryRegion'
    AND ss.name = 'Person'
```

Again, you get the same block of code (indeed, all `sp_helptext` does is run what amounts to this same query).

I've actually deliberately mixed some new and old concepts in the preceding code.

In years gone by, we would freely refer to what were called system objects. These most frequently included `sysobjects`, `sysdepends`, `syscomments`, or one of a few others. Microsoft has warned us over and over that those would be going away and not to depend on them, but, of course, our needs are our needs, and we have probably done it anyway.

With SQL Server 2005, Microsoft begins a journey towards migrating us away from the base system tables (many of which have been replaced with views over the years anyway) and onto a set of table valued functions that should be more stable over time. In the preceding query, I have combined a query against the old `syscomments` table with queries against the replacements for some other system tables. I highly recommend transitioning to these system functions as you are able.

Protecting Code: Encrypting Views

If you're building any kind of commercial software product, odds are that you're interested in protecting your source code. Views are the first place we see the opportunity to do just that.

Chapter 9

All you have to do to encrypt your view is use the `WITH ENCRYPTION` option. This one has a couple of tricks to it if you're used to the `WITH CHECK OPTION` clause:

- `WITH ENCRYPTION` goes after the name of the view, but *before* the `AS` keyword.
- `WITH ENCRYPTION` does not use the `OPTION` keyword.

In addition, remember that if you use an `ALTER VIEW` statement, you are entirely replacing the existing view except for access rights. This means that the encryption is also replaced. If you want the altered view to be encrypted, then you must use the `WITH ENCRYPTION` clause in the `ALTER VIEW` statement.

Let's do an `ALTER VIEW` on the `CustomerOrders_vw` view that we created earlier in the chapter. If you haven't yet created the `CustomerOrders_vw` view, then just change the `ALTER` to `CREATE` (don't forget to run this against AdventureWorks):

```
ALTER VIEW CustomerOrders_vw
WITH ENCRYPTION
AS
SELECT    o.SalesOrderID,
          o.OrderDate,
          od.ProductID,
          p.Name,
          od.OrderQty,
          od.UnitPrice,
          od.LineTotal
FROM Sales.SalesOrderHeader AS o
JOIN Sales.SalesOrderDetail AS od
  ON o.SalesOrderID = od.SalesOrderID
JOIN Production.Product AS p
  ON od.ProductID = p.ProductID
```

Now do an `sp_helptext` on our `CustomerOrders_vw`:

```
EXEC sp_helptext CustomerOrders_vw
```

SQL Server promptly tells us that it can't do what we're asking:

```
The text for object 'CustomerOrders_vw' is encrypted.
```

The heck you say, and promptly go to the `syscomments` table:

```
SELECT sc.text
FROM syscomments sc
JOIN sys.objects so
  ON sc.id = so.object_id
JOIN sys.schemas ss
  ON so.schema_id = ss.schema_id
WHERE so.name = 'CustomerOrders_vw'
  AND ss.name = 'dbo'
```

But that doesn't get you very far either—SQL Server recognizes that the table was encrypted and will give you a `NULL` result.

In short — your code is safe and sound. Even if you pull it up in other viewers (such as Management Studio, which actually won't even give you the Modify option on an encrypted table), you'll find it useless.

Make sure you store your source code somewhere before using the WITH ENCRYPTION option. Once it's been encrypted, there is no way to get it back. If you haven't stored your code away somewhere and you need to change it, then you may find yourself rewriting it from scratch.

About Schema Binding

Schema binding essentially takes the things that your view is dependent upon (tables or other views), and "binds" them to that view. The significance of this is that no one can make alterations to those objects (CREATE, ALTER) unless they drop the schema-bound view first.

Why would you want to do this? Well, there are a few reasons why this can come in handy:

- ❑ It prevents your view from becoming "orphaned" by alterations in underlying objects. Imagine, for a moment, that someone performs a DROP or makes some other change (even deleting a column could cause your view grief) but doesn't pay attention to your view. Oops. If the view is schema bound, then this is prevented from happening.
- ❑ To allow indexed views. If you want an index on your view, you *must* create it using the SCHEMABINDING option. (We'll look at indexed views just a few paragraphs from now.)
- ❑ If you are going to create a schema-bound user-defined function (and there are instances where your UDF *must* be schema bound) that references your view, then your view must also be schema bound.

Keep these in mind as you are building your views.

Making Your View Look Like a Table with VIEW_METADATA

This option has the effect of making your view look very much like an actual table to DB-LIB, ODBC, and OLE-DB clients. Without this option, the metadata passed back to the client API is that of the base table(s) that your view relies on.

Providing this metadata information is required to allow for any client-side cursors (cursors your client applications manages) to be updatable. Note that, if you want to support such cursors, you're also going to need to use an INSTEAD OF trigger.

Indexed (Materialized) Views

In SQL Server 2000, this one was supported only in the Enterprise Edition (okay, the Developer and Evaluation Editions also supported it, but you aren't allowed to use test and development editions in production systems). It is, however, supported in all editions of SQL Server 2005.

When a view is referred to, the logic in the query that makes up the view is essentially incorporated into the calling query. Unfortunately, this means that the calling query just gets that much more complex. The extra overhead of figuring out the impact of the view (and what data it represents) on the fly can actually get very high. What's more, you're often adding additional joins into your query in the form of the tables that are joined in the view. Indexed views give you a way of taking care of some of this impact before the query is ever run.

An indexed view is essentially a view that has had a set of unique values "materialized" into the form of a clustered index. The advantage of this is that it provides a very quick lookup in terms of pulling the information behind a view together. After the first index (which must be a clustered index against a unique set of values), SQL Server can also build additional indexes on the view using the cluster key from the first index as a reference point. That said, nothing comes for free—there are some restrictions about when you can and can't build indexes on views (I hope you're ready for this one—it's an awfully long list!):

- ❑ The view must use the SCHEMABINDING option.
- ❑ If it references any user-defined functions (more on these later in the book), then these must also be schema bound.
- ❑ The view must not reference any other views—just tables and UDFs.
- ❑ All tables and UDFs referenced in the view must utilize a two-part (not even three-part and four-part names are allowed) naming convention (for example `dbo.Customers`, `BillyBob.SomeUDF`) and must also have the same owner as the view.
- ❑ The view must be in the same database as all objects referenced by the view.
- ❑ The `ANSI_NULLS` and `QUOTED_IDENTIFIER` options must have been turned on (using the `SET` command) at the time the view and all underlying tables were created.
- ❑ Any functions referenced by the view must be deterministic.

To create an example indexed view, let's start by reviewing the `CustomerOrders_vw` object that we created earlier in the chapter. I'm showing this using the `ALTER` statement we used in the section on encryption, but, really, it could just as easily be the original version we created very early in the chapter as long as the `WITH SCHEMABINDING` is properly added.

```
ALTER VIEW CustomerOrders_vw
WITH SCHEMABINDING
AS
SELECT    o.SalesOrderID,
          o.OrderDate,
          od.ProductID,
          p.Name,
          od.OrderQty,
          od.UnitPrice,
          od.LineTotal
FROM Sales.SalesOrderHeader AS o
```

```

JOIN Sales.SalesOrderDetail AS od
ON o.SalesOrderID = od.SalesOrderID
JOIN Production.Product AS p
ON od.ProductID = p.ProductID

```

Some important things to notice here are:

- ❑ We had to make our view use the SCHEMABINDING option.
- ❑ In order to utilize the SCHEMABINDING option, we must have two-part naming for the objects (in this case, all tables) that we reference (in this case, we did anyway, but not all views you come across will already be configured that way).

This is really just the beginning — we don't have an indexed view as yet. Instead, what we have is a view that *can* be indexed. When we create the index, the first index created on the view must be both clustered and unique.

```

CREATE UNIQUE CLUSTERED INDEX ivCustomerOrders
ON CustomerOrders_vw(SalesOrderID, ProductID, Name)

```

Once this command has executed, we have a clustered view. We also, however, have a small problem that will become clear in just a moment.

Let's test our view by running a simple SELECT against it:

```
SELECT * FROM CustomerOrders_vw
```

If you execute this, you'll see that the graphical showplan as shown in Figure 9-1 (display Estimated Execution Plan is the tooltip for this, and you'll find it toward the center of the toolbar; you can also find it in the menus at Query→Display Estimated Execution Plan) shows us using our new index:

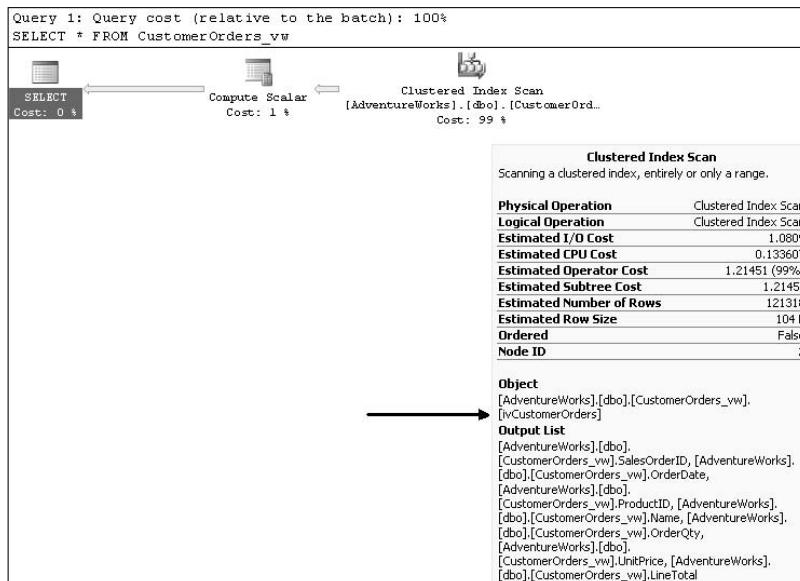


Figure 9-1

The index supporting an indexed view may be utilized by SQL Server even if you do not explicitly use the view. For example, if you are performing joins that are similar to those the index is supporting for the view, SQL Server may recognize this and utilize the index.

Partitioned Views

These have been in use just since SQL Server 2000, but they are already being deprecated by Microsoft. I bring them up here because they were one of the leading scalability options put forth by Microsoft for many years, and you need to know how they work in case you run into them in legacy code. With SQL Server 2005, they are being replaced by partitioned tables.

A partitioned view is a view that unifies multiple identical (in terms of structure—not actual data) tables and makes them appear to be a single table. At first, this seems like an easy thing to do with simple `UNION` clauses, but the concept actually becomes somewhat tricky when you go to handle insert and update scenarios.

With partitioned views, we define a constraint on one of the tables in our view. We then define a similar, but mutually exclusive, constraint on a second (and possibly many more) table. When you build the view that unifies these mutually exclusive tables, SQL Server is able to sort out the exclusive nature of the tables in a logical manner. By doing this, SQL Server can determine exactly which table is to get the new data (by determining which table *can* accept the data—if you created them as mutually exclusive as you should have, then the data will be able to get into only one table and there is no conflict).

Summary

Views tend to be either the most overused or most underused tools in most of the databases I've seen. Some people like to use them to abstract seemingly everything (often forgetting that they are adding another layer to the process when they do this). Others just seem to forget that views are even an option. Personally, like most things, I think you should use a view when it's the right tool to use—not before, not after.

Common uses for views include:

- ❑ Filtering rows
- ❑ Protecting sensitive data
- ❑ Reducing database complexity
- ❑ Abstracting multiple physical databases into one logical database

Things to remember with views include:

- ❑ Stay away from building views based on views—instead, adapt the appropriate query information from the first view into your new view.

-
- ❑ Remember that a view using the WITH CHECK OPTION provides some flexibility that can't be duplicated with a normal CHECK constraint.
 - ❑ Encrypt views when you don't want others to be able to see your source code — either for commercial products or general security reasons.
 - ❑ Using an ALTER VIEW completely replaces the existing view other than permissions. This means you must include the WITH ENCRYPTION and WITH CHECK OPTION clauses in the ALTER statement if you want encryption and restrictions to be in effect in the altered view.
 - ❑ Use `sp_helptext` to display the supporting code for a view — avoid using the system tables.
 - ❑ Minimize the use of views for production queries — they add additional overhead and hurt performance.

In our next chapter, we'll take a look at batches and scripting. We got a brief taste when we ran the `INSERT` script in this chapter to insert a row into the `SalesOrderHeader` table and then used information from the freshly inserted row in an insert into the `SalesOrderDetail` table. Batches and scripting will lead us right into stored procedures — the closest thing that SQL Server has to its own programs.

10

Scripts and Batches

Geez. I've been writing too long. For some reason when I see the phrase "Scripts and Batches" it reminds me of the old song "Love and Marriage." While scripts and batches do go together like a horse and carriage, they are hardly as lyrical—but I digress. . . .

We have, of course, already been writing SQL scripts. Every CREATE statement that you write, every ALTER, every SELECT is all (if you're running a single statement) or part (multiple statements) of a script. It's hard to get excited, though, over a script with one line in it—could you imagine Hamlet's "To be, or not to be . . . ?" if it had never had the following lines—we wouldn't have any context for what he was talking about.

SQL scripts are much the same way. Things get quite a bit more interesting when we string several commands together into a longer script—a full play or at least an act to finish our Shakespeare analogy. Now imagine that we add a more rich set of language elements from .NET to the equation—now we're ready to write an epic!

Scripts generally have a unified goal. That is, all the commands that are in a script are usually building up to one overall purpose. Examples include scripts to build a database (these might be used for a system installation), scripts for system maintenance (backups, Database Consistency Checker utilities (DBCCs)—scripts for anything where several commands are usually run together.

We will be looking into scripts during this chapter, and adding in the notion of *batches*—which control how SQL Server groups your commands together. In addition, we will take a look at *SQLCMD*—the command-line utility—and how it relates to scripts.

SQLCMD is new with SQL Server 2005. For backward compatibility only, SQL Server also supports osql.exe (the previous tool that did command-line work). You may also see references to isql.exe (do not confuse this with isqlw.exe), which served this same function in earlier releases. Isql.exe is no longer supported as of SQL Server 2005.

Script Basics

A script technically isn't a script until you store it in a file where it can be pulled up and reused. SQL scripts are stored as text files. SQL Server Management Studio provides many tools to help you with your script writing, but you can do the writing in any text editor (keep in mind, however, that in order to actually test your script, it's going to have to be something that can connect). Indeed, I frequently use a highly robust text editor for its ability to handle regular expressions and other text-editing features that Management Studio, and even Visual Studio, will never have.

Scripts are usually treated as a unit. That is, you are normally executing the entire script or nothing at all. They can make use of both system functions and local variables. As an example, let's look at a simple script that could be used to `INSERT` order records into a typical order header and order detail table scenario:

```
USE SomeDatabase

DECLARE @Ident int

INSERT INTO Orders
(CustomerID, OrderDate)
VALUES
(25, DATEADD(day, -1, GETDATE())) -- this always sets the OrderDate to yesterday

SELECT @Ident = @@IDENTITY

INSERT INTO Details
(OrderID, ProductID, UnitPrice, Quantity)
VALUES
(@Ident, 1, 50, 25)

SELECT 'The OrderID of the INSERTed row is ' + CONVERT(varchar(8),@Ident)
```

We have six distinct commands working here, covering a range of different things that we might do in a script. We're using both system functions and local variables, the `USE` statement, `INSERT` statements, and both assignment and regular versions of the `SELECT` statement. They are all working in unison to accomplish one task—to insert complete orders into the database.

The `USE` Statement

The `USE` statement sets the current database. This affects any place where we are making use of default values for the database portion of our fully qualified object name. In this particular example, we have not indicated what database the tables in our `INSERT` or `SELECT` statements are from, but, since we've included a `USE` statement prior to our `INSERT` and `SELECT` statements, they will use that database (in this case, `SomeDatabase`). Without our `USE` statement, we would be at the mercy of whoever executes the script to make certain that the correct database was current when the script was executed.

Don't take this as meaning that you should always include a `USE` statement in your script—it depends on what the purpose of the script is. If your intent is to have a general-purpose script, then leaving out the `USE` statement might actually be helpful.

Usually, if you are naming database-specific tables in your script (that is, non-system tables), then you want to use the `USE` command. I also find it very helpful if the script is meant to modify a specific database—as I've said in prior chapters, I can't tell you how many times I've accidentally created a large number of tables in the `master` database that were intended for a user database.

Declaring Variables

The `DECLARE` statement has a pretty simple syntax:

```
DECLARE @<variable name> <variable type>[,  
        @<variable name> <variable type>[,  
        @<variable name> <variable type>]]
```

You can declare just one variable at a time or several. It's common to see people reuse the `DECLARE` statement with each variable they declare, rather than using the comma separated method. It's up to you, but no matter which method you choose, the value of your variable will always be `NULL` until you explicitly set it to some other value.

In our case, we've declared a local variable called `@ident` as an integer.

I like to move a value I'm taking from a system function into my own variable. That way I can safely use the value and know that it's only being changed when I change it. With the system function itself, you sometimes can't be certain when it's going to change because most system functions are not set by you, but by the system. That creates a situation where it would be very easy to have the system change a value at a time you weren't expecting it, and wind up with the most dreaded of all computer terms: unpredictable results.

Setting the Value in Your Variables

Well, we now know how to declare our variables, but the question that follows is, "How do we change their values?" There are currently two ways to set the value in a variable. You can use a `SELECT` statement or a `SET` statement. Functionally, they work almost the same, except that a `SELECT` statement has the power to have the source value come from a column within the `SELECT` statement.

To be honest, I can't tell you why there are two ways of doing this, but that said, there are some differences in the way they are typically put to use.

Chapter 10

Setting Variables Using SET

SET is usually used for setting variables in the fashion that you would see in more procedural languages. Examples of typical uses are:

```
SET @TotalCost = 10  
SET @TotalCost = @UnitCost * 1.1
```

Notice that these are all straight assignments that use either explicit values or another variable. With a SET, you cannot assign a value to a variable from a query — you have to separate the query from the SET. For example:

```
USE AdventureWorks  
  
DECLARE @Test money  
  
SET @Test = MAX(UnitPrice) FROM SalesOrderDetail  
SELECT @Test
```

causes an error, but:

```
USE AdventureWorks  
  
DECLARE @Test money  
  
SET @Test = (SELECT MAX(UnitPrice) FROM SalesOrderDetail)  
SELECT @Test
```

works just fine.

Although this latter syntax works, by convention, code is never implemented this way. Again, I don't know for sure why it's "just not done that way," but I suspect that it has to do with readability — you want a SELECT statement to be related to retrieving table data, and a SET to be about simple variable assignments.

Setting Variables Using SELECT

SELECT is usually used to assign variable values when the source of the information you're storing in the variable is from a query. For example, the last of our preceding illustrations would be far more typically done using a SELECT:

```
USE AdventureWorks  
  
DECLARE @Test money  
  
SELECT @Test = MAX(UnitPrice) FROM SalesOrderDetail  
  
SELECT @Test
```

Notice that this is a little cleaner (it takes less verbiage to do the same thing).

So again, the convention on when to use which goes like this:

- Use `SET` when you are performing a simple assignment of a variable — where your value is already known in the form of an explicit value or some other variable.
- Use `SELECT` when you are basing the assignment of your variable on a query.

I'm not going to pick any bones about the fact that you'll see me violate this last convention in many places in this book. Using `SET` for variable assignment first appeared in version 7.0, and I must admit that, even 7+ years after that release, I still haven't completely adapted yet. Nonetheless, this seems to be something that's really being pushed by Microsoft and the SQL Server community, so I strongly recommend that you start out on the right foot and adhere to the convention.

Reviewing System Functions

There are over 30 parameterless system functions available. Some of the ones you should be most concerned with are in the table that follows:

Variable	Purpose	Comments
<code>@@ERROR</code>	Returns the error number of the last T-SQL statement executed on the current connection. Returns 0 if no error.	Is reset with each new statement. If you need the value preserved, move it to a local variable immediately after the execution of the statement for which you want to preserve the error code.
<code>@@FETCH_STATUS</code>	Used in conjunction with a <code>FETCH</code> statement.	Returns 0 for valid fetch, % for beyond end of cursor set, -2 for a missing (deleted) row. Typical gotcha is to assume that any non-zero value means you are at the end of the cursor — a -2 may just mean one missing record. This is covered in far more detail in Chapter 15.
<code>@@IDENTITY</code>	Returns the last identity value inserted as a result of the last <code>INSERT</code> or <code>SELECT INTO</code> statement.	Is set to <code>NULL</code> if no identity value was generated. This is true even if the lack of an identity value was due to a failure of the statement to run. If multiple inserts are performed by just one statement, then only the last identity value is returned.
<code>@@ROWCOUNT</code>	One of the most used system functions. Returns the number of rows affected by the last statement.	Commonly used in nonruntime error checking. For example, if you try to <code>DELETE</code> a row using a <code>WHERE</code> clause, and no rows are affected, then that would imply that something unexpected happened. You can then raise an error manually.

Table continued on following page

Variable	Purpose	Comments
@@SERVERNAME	Returns the name of the local server that the script is running from.	Can be changed by using <code>sp_addserver</code> and then restarting SQL Server, but rarely required.
@@TRANCOUNT	Returns the number of active transactions — essentially the transaction nesting level — for the current connection.	A <code>ROLLBACK TRAN</code> statement decrements <code>@@TRANCOUNT</code> to 0 unless you are using savepoints. <code>BEGIN TRAN</code> increments <code>@@TRANCOUNT</code> by 1, <code>COMMIT TRAN</code> decrements <code>@@TRANCOUNT</code> by 1.

Don't worry if you don't recognize some of the terms in a few of these. They will become clear in due time, and you will have this table to look back on for reference at a later date. The thing to remember is that there are sources you can go to in order to find out a whole host of information about the current state of your system and your activities.

Using @@IDENTITY

`@@IDENTITY` is one of the most important of all the system functions. Remember when we saw identity values all the way back in Chapter 4? An identity column is one where we don't supply a value, and SQL Server inserts a numbered value automatically.

In our example case earlier in the chapter, we obtain the value of `@@IDENTITY` right after performing an insert into the parent (in this case, `Orders`) table. The issue is that we don't supply the key value for that table — it's automatically created as we do the insert. Now we want to insert a record into the `Details` table, but we need to know the value of the primary key in the associated record in the `Orders` table (In this kind of scenario, there will almost certainly be a foreign key constraint on the child table that references the parent table). Since SQL Server generated that value instead of us supplying it, we need to have a way to retrieve that value for use in our dependent inserts later on in the script. `@@IDENTITY` gives us that automatically generated value since it was the last statement run.

In the case of our example, we could have easily gotten away with not moving `@@IDENTITY` to a local variable — we could have just referenced it explicitly in our next `INSERT` query. I make a habit of always moving it to a local variable though, to avoid errors on the occasions when I do need to keep a copy. An example of this kind of situation would be if we had yet another `INSERT` that was dependent on the identity value from the `INSERT` into the `Orders` table. If I hadn't moved it into a local variable, then it would be lost when I did the next `INSERT`, because it would have been overwritten with the value from the `Details` table. In the case where the next table doesn't have an identity column (as most child tables in this scenario won't), `@@IDENTITY` would be set to `NULL`. Moving the value of `@@IDENTITY` to a local variable also let me keep the value around for the statement where I printed out the value for later reference.

Using @@ROWCOUNT

In the many queries that we've run up to this point, it's always been pretty easy to tell how many rows a statement affected — Management Studio tells us. For example, if we run:

```
USE AdventureWorks  
  
SELECT * FROM HumanResources.Employee
```

then we see all the rows in `Employee`, but we also see a count on the number of rows affected by our query (in this case, it's all the rows in the table):

```
(290 row(s) affected)
```

But what if we need to *programmatically* know how many rows were affected? Much the same way `@@IDENTITY`, `@@ROWCOUNT` is an invaluable tool in the fight to know what's going on as your script runs—but this time the value is how many rows were affected rather than our identity value.

Let's examine this just a bit further with an example:

```
USE AdventureWorks
GO

DECLARE @RowCount int -- Notice the single @ sign

SELECT * FROM HumanResources.Employee

SELECT @RowCount = @@ROWCOUNT

PRINT 'The value of @@ROWCOUNT was ' + CAST(@RowCount AS varchar(5))
```

This again shows us all the rows, but notice the new line that we got back:

```
The value of @@ROWCOUNT was 290
```

If you look through the example, you might notice that, much as I did with `@@IDENTITY`, I chose to move the value off to a holding variable. `@@ROWCOUNT` will be reset with a new value the very next statement, so, if you're going to be doing multiple activities with the `@@ROWCOUNT` value, you should move it into a safekeeping area.

Batches

A *batch* is a grouping of T-SQL statements into one logical unit. All of the statements within a batch are combined into one execution plan, so all statements are parsed together and must pass a validation of the syntax or none of the statements will execute. Note, however, that this does not prevent runtime errors from happening. In the event of a runtime error, any statement that has been executed prior to the runtime error will still be in effect. To summarize, if a statement fails at parse-time, then nothing runs—if a statement fails at runtime, then all statements until the statement that generated the error have already run.

All the scripts we have run up to this point are made up of one batch each. Even the script we've been analyzing so far this in chapter is just one batch. To separate a script into multiple batches, we make use of the `GO` statement. The `GO` statement:

- Must be on its own line (nothing other than a comment can be on the same line); there is an exception to this discussed shortly, but think of a `GO` as needing to be on a line to itself

Chapter 10

- ❑ Causes all statements since the beginning of the script or the last GO statement (whichever is closer) to be compiled into one execution plan and sent to the server independently of any other batches
- ❑ Is not a T-SQL command, but, rather, a command recognized by the various SQL Server command utilities (OSQL, ISQL, and the Query Analyzer)

A Line to Itself

The GO command should stand alone on its own line. Technically, you can start a new batch on the same line after the GO command, but you'll find this puts a serious damper on readability. T-SQL statements cannot precede the GO statement, or the GO statement will often be misinterpreted and cause either a parsing error or some other unexpected result. For example, if I use a GO statement after a WHERE clause:

```
SELECT * FROM Customers WHERE CustomerID = 2 GO
```

The parser becomes somewhat confused:

```
Msg 102, Level 15, State 1, Line 1  
Incorrect syntax near 'GO'.
```

Each Batch Is Sent to the Server Separately

Because each batch is processed independently, an error in one batch does not preclude another batch from running. To illustrate, take a look at some code:

```
USE AdventureWorks

DECLARE @MyVarchar varchar(50) --This DECLARE only lasts for this batch!

SELECT @MyVarchar = 'Honey, I ''m home...'

PRINT 'Done with first Batch...'

GO

PRINT @MyVarchar --This generates an error since @MyVarchar
--isn't declared in this batch
PRINT 'Done with second Batch'

GO

PRINT 'Done with third batch' -- Notice that this still gets executed
-- even after the error

GO
```

If there were any dependencies between these batches, then either everything would fail—or, at the very least, everything after the point of error would fail—but it doesn't. Look at the results if you run the preceding script:

```
Done with first Batch...
Msg 137, Level 15, State 2, Line 2
```

```
Must declare the scalar variable "@MyVarchar".  
Done with third batch
```

Again, each batch is completely autonomous in terms of runtime issues. Keep in mind though that you can build in dependencies in the sense that one batch may try to perform work that depends on the first batch being complete—we'll see some of this in the next section when we talk about what can and can't span batches.

GO Is Not a T-SQL Command

Thinking that GO is a T-SQL command is a common mistake. GO is a command that is recognized only by the editing tools (Management Studio, SQLCMD). If you use a third-party tool, then it may or may not support the GO command, but most that claim SQL Server support will.

When the editing tool encounters a GO statement, it sees it as a flag to terminate that batch, package it up, and send it as a single unit to the server—*without* including the GO. That's right; the server itself has absolutely no idea what GO is supposed to mean.

If you try to execute a GO command in a pass-through query using ODBC, OLE DB, ADO, ADO.NET, or any other access method, you'll get an error message back from the server. The GO is merely an indicator to the tool that it is time to end the current batch, and time, if appropriate, to start a new one. In the case of the aforementioned access methods, they each have the concept of a "command" object. That command object may include multiple statements, but each execution of the command object is implied to represent exactly one batch.

Errors in Batches

Errors in batches fall into two categories:

- Syntax errors
- Runtime errors

If the query parser finds a *syntax error*, processing of that batch is canceled immediately. Since syntax checking happens before the batch is compiled or executed, a failure during the syntax check means none of the batch will be executed—regardless of the position of the syntax error within the batch.

Runtime errors work quite a bit differently. Any statement that has already executed before the runtime error was encountered is already done, so anything that statement did will remain intact unless it is part of an uncommitted transaction (transactions are covered in Chapter 12, but the relevance here is that they imply an all or nothing situation). What happens beyond the point of the runtime error depends on the nature of the error. Generally speaking, runtime errors will terminate execution of the batch from the point where the error occurred to the end of the batch. Some runtime errors, such as a referential-integrity violation will prevent only the offending statement from executing—all other statements in the batch will still be executed. This later scenario is why error checking is so important—we will cover error checking in full in our chapter on stored procedures (Chapter 11).

When to Use Batches

Batches have several purposes, but they all have one thing in common—they are used when something has to happen either before or separately from everything else in your script.

Statements That Require Their Own Batch

There are several commands that absolutely must be part of their own batch. These include:

- CREATE DEFAULT
- CREATE PROCEDURE
- CREATE RULE
- CREATE TRIGGER
- CREATE VIEW

If you want to combine any of these statements with other statements in a single script, then you will need to break them up into their own batch by using a GO statement.

Note that, if you DROP an object, you may want to place the DROP in its own batch or at least with a batch of other DROP statements. Why? Well, if you're going to later create an object with the same name, the CREATE will fail during the parsing of your batch unless the DROP has already happened. That means you need to run the DROP in a separate and prior batch so it will be complete when the batch with the CREATE statement executes.

Using Batches to Establish Precedence

Perhaps the most likely scenario for using batches is when precedence is required—that is, you need one task to be completely done before the next task starts. Most of the time, SQL Server deals with this kind of situation just fine—the first statement in the script is the first executed, and the second statement in the script can rely on the server being in the proper state when the second statement runs. There are times, however, when SQL Server can't resolve this kind of issue.

Let's take the example of creating a database together with some tables:

```
CREATE DATABASE Test

CREATE TABLE TestTable
(
    col1    int,
    col2    int
)
```

Execute this and, at first, it appears that everything has gone well:

```
Command(s) completed successfully.
```

However, things are not as they seem—check out the INFORMATION_SCHEMA in the Test database, and you'll notice something is missing:

```
SELECT TABLE_CATALOG FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_NAME =
    'TestTable'
```

```
TABLE_CATALOG
```

```
-----  
master
```

```
(1 row(s) affected)
```

Hey! Why was the table created in the wrong database? The answer lies in what database was current when we ran the CREATE TABLE statement. In our case, it happened to be the master database, so that's where our table was created.

Note that you may have been somewhere other than the master database when you ran this, so you may get a different result. That's kind of the point though—you could be in pretty much any database. That's why making use of the USE statement is so important.

When you think about it, this seems like an easy thing to fix—just make use of the USE statement, but before we test our new theory, we have to get rid of the old (okay, not that old) database.

```
USE MASTER
DROP DATABASE Test
```

We can then run our newly modified script:

```
CREATE DATABASE Test
```

```
USE Test
```

```
CREATE TABLE TestTable
(
    col1    int,
    col2    int
)
```

Unfortunately, this has its own problems:

```
Msg 911, Level 16, State 1, Line 3
Could not locate entry in sysdatabases for database 'Test'. No entry found with
that name. Make sure that the name is entered correctly.
```

The parser tries to validate our code and finds that we are referencing a database with a USE command that doesn't exist. Ahh, now we see the need for our batches. We need the CREATE DATABASE statement to be completed before we try to use the new database:

Chapter 10

```
CREATE DATABASE Test  
GO
```

```
USE Test  
  
CREATE TABLE TestTable  
(  
    col1    int,  
    col2    int  
)
```

Now things work a lot better. Our immediate results look the same:

```
Command(s) completed successfully.
```

But when we run our INFORMATION_SCHEMA query, things are confirmed:

```
TABLE_CATALOG  
-----  
Test  
(1 row(s) affected)
```

Let's move on to another example that shows an even more explicit need for precedence.

When you use an ALTER TABLE statement that significantly changes the type of a column or adds columns, you cannot make use of those changes until the batch that makes the changes has completed.

If we add a column to our TestTable table in our Test database and then try to reference that column without ending the first batch:

```
USE Test  
  
ALTER TABLE TestTable  
    ADD col3 int  
  
INSERT INTO TestTable  
(col1, col2, col3)  
VALUES  
(1,1,1)
```

We get an error message—SQL Server cannot resolve the new column name and therefore complains:

```
Msg 207, Level 16, State 1, Line 6  
Invalid column name 'col3'.
```

Add one simple GO statement after the ADD col3 int though, and everything is working fine:

```
(1 row(s) affected)
```

SQLCMD

SQLCMD is a utility that allows you to run scripts from a command prompt in a Windows command box. This can be very nice for executing conversion or maintenance scripts, as well as a quick-and-dirty way to capture a text file.

SQLCMD replaces the older OSQL. OSQL is still included with SQL Server for backward compatibility only. An even older command-line utility—ISQL—is no longer supported.

The syntax for running SQLCMD from the command line includes a large number of different switches, and looks like this:

```
sqlcmd
[
{ { -U <login id> [ -P <password> ] } | -E }
]
[-S <server name> [ \<instance name> ] ] [ -H <workstation name> ] [ -d <db name> ]
[ -l <time out> ] [ -t <time out> ] [ -h <headers> ]
[ -s <col separator> ] [ -w <col width> ] [ -a <packet size> ]
[ -e ] [ -I ]
[ -c <cmd end> ] [ -L [ c ] ] [ -q "<query>" ] [ -Q "<query>" ]
[ -m <error level> ] [ -V ] [ -W ] [ -u ] [ -r [ 0 | 1 ] ]
[ -i <input file> ] [ -o <output file> ]
[ -f <codepage> | i:<codepage> | <, o:<codepage> ]
[ -k [ 1 | 2 ] ]
[ -y <display width> ] [ -Y <display width> ]
[ -p [ 1 ] ] [ -R ] [ -b ] [ -v ] [ -A ] [ -X [ 1 ] ] [ -x ]
[ -? ]
]
```

The single biggest thing to keep in mind with these flags is that many of them (but, oddly enough, not all of them) are case sensitive. For example, both `-Q` and `-q` will execute queries, but the first will exit SQLCMD when the query is complete, and the second won't.

So, let's try a quick query direct from the command line. Again, remember that this is meant to be run from the Windows command prompt (don't use the Management Console):

```
SQLCMD -Usa -Pmypassword -Q "SELECT * FROM AdventureWorks.HumanResources.Employee"
```

The `-P` is the flag that indicates the password. If your server is configured with something other than a blank password (and it should be!), then you'll need to provide that password immediately following the `-P` with no space in between.

If you run this from a command prompt, you should get something like 290 rows back. Now, let's create a quick text file to see how it works when including a file. At the command prompt, type the following:

```
C:\>copy con testsql.sql
```

Chapter 10

This should take you down to a blank line (with no prompt of any kind), where you can enter this:

```
SELECT * FROM AdventureWorks.HumanResources.Employee
```

Then press F6 and Return (this ends the creation of our text file). You should get back a message like:

```
1 file(s) copied.
```

Now let's retry our earlier query, using a script file this time. The command line at the prompt only has a slight change to it:

```
C:\>sqlcmd -Usa -Pmypass -i testsql.sql
```

This should get us exactly the same results as when we ran the query using `-Q`. The major difference is, of course, that we took the command from a file. The file could have had hundreds—if not thousands—of different commands in it.

There are a wide variety of different parameters for SQLCMD, but the most important are the login, the password, and the one that says what you want to do (straight query or input file). You can mix and match many of these parameters to obtain fairly complex behavior from this seemingly simple command-line tool.

Dynamic SQL: Generating Your Code on the Fly with the EXEC Command

Okay, so all this saving stuff away in scripts is all fine and dandy, but what if you don't know what code you need to execute until runtime?

As a side note, notice that we are done with SQLCMD for now—the following examples should be run utilizing the Management Console.

SQL Server allows us, with a few gotchas, to build our SQL statement on the fly using string manipulation. The need to do this usually stems from not being able to know the details about something until runtime. The syntax looks like this:

```
EXEC ({<string variable>}|'<literal command string>' )
```

Or:

```
EXECUTE ({<string variable>}|'<literal command string>' )
```

As with executing a stored proc, whether you use the `EXEC` or `EXECUTE` makes no difference.

Let's build an example in the `AdventureWorks` database by creating a dummy table to grab our dynamic information out of:

```

USE AdventureWorks
GO

--Create The Table. We'll pull info from here for our dynamic SQL
CREATE TABLE DynamicSQLExample
(
    TableID      int      IDENTITY   NOT NULL
    CONSTRAINT PKDynamicSQLExample
        PRIMARY KEY,
    SchemaName  varchar(128)     NOT NULL,
    TableName    varchar(128)     NOT NULL
)
GO

/* Populate the table. In this case, We're grabbing every user
** table object in this database */
INSERT INTO DynamicSQLExample
SELECT s.name AS SchemaName, t.name AS TableName
    FROM sys.schemas s
    JOIN sys.tables t
        ON s.schema_id = t.schema_id

```

This should get us a response something like:

```
(75 row(s) affected)
```

To quote the old advertising disclaimer: "actual results may vary." It's going to depend on which examples you've already followed along with in the book, which ones you haven't, and for which ones you took the initiative and did a DROP on once you were done with them. In any case, don't sweat it too much.

Okay, so what we now have is a list of all the tables in our current database. Now let's say that we wanted to select some data from one of the tables, but we wanted to identify the table only at runtime by using its ID. For example, I'll pull out all the data for the table with an ID of 1:

```

DECLARE @SchemaName      varchar(128)
DECLARE @TableName        varchar(128)

-- Now, grab the table name that goes with our ID
SELECT @SchemaName = SchemaName, @TableName = TableName
    FROM DynamicSQLExample
    WHERE TableID = 13

-- Finally, pass that value into the EXEC statement
EXEC ('SELECT * FROM ' + @SchemaName + '.' + @TableName)

```

If your table names went into the `DynamicSQLExample` table the way mine did, then a `TableID` of 13 should equate to the `Categories` table. If so, you should wind up with something like this (the right-most columns have been snipped for brevity):

VendorID	AccountNumber	Name	...
1	INTERNAT0001	International	...
2	ELECTRON0002	Electronic Bike Repair & Supplies	...
3	PREMIER0001	Premier Sport, Inc.	...
4	COMFORT0001	Comfort Road Bicycles	...
...

The Gotchas of EXEC

Like most things that are of interest, using `EXEC` is not without its little trials and tribulations. Among the gotchas of `EXEC` are:

- ❑ It runs under a separate scope than the code that calls it—that is, the calling code can't reference variables inside the `EXEC` statement, and the `EXEC` can't reference variables in the calling code after they are resolved into the string for the `EXEC` statement.
- ❑ It runs under the same security context as the current user—not that of the calling object.
- ❑ It runs under the same connection and transaction context as the calling object (we'll discuss this further in Chapter 12).
- ❑ Concatenation that requires a function call must be performed on the `EXEC` string prior to actually calling the `EXEC` statement—you can't do the concatenation of function in the same statement as the `EXEC` call.
- ❑ `EXEC` cannot be used inside a user-defined function.

Each of these can be a little difficult to grasp, so let's look at each individually.

The Scope of EXEC

Determining variable scope with the `EXEC` statement is something less than intuitive. The actual statement line that calls the `EXEC` statement has the same scope as the rest of the batch or procedure that the `EXEC` statement is running in, but the code that is performed as a result of the `EXEC` statement is considered to be in its own batch. As is so often the case, this is best shown with an example:

```
USE AdventureWorks

/* First, we'll declare to variables. One for stuff we're putting into
** the EXEC, and one that we think will get something back out (it won't)
*/
DECLARE @InVar    varchar(50)
DECLARE @OutVar   varchar(50)

-- Set up our string to feed into the EXEC command
SET @InVar = 'SELECT @OutVar = FirstName FROM Person.Contact
              WHERE ContactID = 1'

-- Now run it
```

```
EXEC (@Invar)

-- Now, just to show there's no difference, run the select without using a in
variable
EXEC ('SELECT @OutVar = FirstName FROM Person.Contact WHERE ContactID = 1')

-- @OutVar will still be NULL because we haven't been able to put anything in it
SELECT @OutVar
```

Now, look at the output from this:

```
Msg 137, Level 15, State 1, Line 1
Must declare the scalar variable '@OutVar'.
Msg 137, Level 15, State 1, Line 1
Must declare the scalar variable '@OutVar'.

-----
NULL
(1 row(s) affected)
```

SQL Server wastes no time in telling us that we are scoundrels and clearly don't know what we're doing. Why do we get a "Must Declare" error message when we have already declared @OutVar? Because we've declared it in the outer scope—not within the EXEC itself.

Let's look at what happens if we run things a little differently:

```
USE AdventureWorks

-- This time, we only need one variable. It does need to be longer though.
DECLARE @InVar      varchar(200)

/* Set up our string to feed into the EXEC command. This time we're going
** to feed it several statements at a time. They will all execute as one
** batch.
*/
SET @InVar = 'DECLARE @OutVar varchar(50)
              SELECT @OutVar = FirstName FROM Person.Contact
              WHERE ContactID = 1
              SELECT ''The Value Is '' + @OutVar'

-- Now run it
EXEC (@Invar)
```

This time we get back results closer to what we expect:

```
-----
The Value Is Gustavo
```

Notice the way that I'm using two quotation marks right next to each other to indicate that I really want a quotation mark rather than to terminate my string.

Chapter 10

So, what we've seen here is that we have two different scopes operating, and nary the two shall meet. There is, unfortunately, no way to pass information between the inside and outside scopes without using an external mechanism such as a temporary table. If you decide to use a temp table to communicate between scopes, just remember that any temporary table created within the scope of your `EXEC` statement will only live for the life of that `EXEC` statement.

This behavior of a temp table only lasting the life of the `EXEC` procedure will show up again when we are dealing with triggers and sprocs.

A Small Exception to the Rule

There is one thing that happens inside the scope of the `EXEC` that can be seen after the `EXEC` is done—system functions. So, things like `@@ROWCOUNT` can still be used. Again, let's look at a quick example:

```
USE AdventureWorks

EXEC ('SELECT * FROM Sales.Customer')
SELECT 'The Rowcount is ' + CAST(@@ROWCOUNT as varchar)
```

the yields us (after the result set):

```
The Rowcount is 19185
```

Security Contexts and `EXEC`

When you give someone the right to run a stored procedure, you imply that he or she also gains the right to perform the actions called for within the sproc. For example, let's say we had a stored procedure that lists all the employees hired within the last year. Someone who has rights to execute the sproc can do so (and get results back) even if he or she does not have rights to the `HumanResources.Employee` table directly. This is really handy for reasons we will explore later in Chapter 11.

Developers usually assume that this same implied right is valid for an `EXEC` statement also—not necessarily. Indeed, by default, any reference made inside an `EXEC` statement will be run under the security context of the current user. So, let's say I have the right to run a procedure called `spNewEmployee`, but I do not have rights to the `Employee` table. If `spNewEmployee` gets the values by running a simple `SELECT` statement, then everything is fine. If, however, `spNewEmployee` uses an `EXEC` statement to execute that `SELECT` statement, the `EXEC` statement will fail because I don't have the rights to perform a `SELECT` on the `Employee` table.

Fortunately, we can get around this. We'll discuss the specifics of how to do so as we work with stored procedures, functions, and triggers later in the book. I bring this up now because this is where I chose to discuss `EXEC` (because it can be run in scripts, not just stored procs and other executable objects). We'll discuss how to explicitly state what rights to run under when we discuss our compiled objects later in the book.

The security context of an `EXEC` statement ran within a stored procedure, user-defined function, or trigger can be overridden using the `EXECUTE AS` clause within the sproc, function, or trigger. `EXECUTE AS` will be discussed more fully when we look at sprocs, functions, and triggers later in the book.

Use of Functions in Concatenation and EXEC

This one is actually more of a nuisance than anything else, since there is a reasonably easy workaround. Simply put, you can't run a function against your `EXEC` string in the argument for an `EXEC`. For example:

```
USE AdventureWorks

-- This won't work
DECLARE @NumberOfLetters int
SET @NumberOfLetters = 3
EXEC('SELECT LEFT(LastName,' + CAST(@NumberOfLetters AS varchar) + ') AS FilingName
FROM Person.Contact')
GO

-- But this does
DECLARE @NumberOfLetters AS int
SET @NumberOfLetters = 3
DECLARE @str AS varchar(255)
SET @str = 'SELECT LEFT(LastName,' + CAST(@NumberOfLetters AS varchar) + ') AS
FilingName FROM Person.Contact'
EXEC(@str)
```

The first instance gets us an error message because the `CAST` function needs to be fully resolved prior to the `EXEC` line:

```
Msg 102, Level 15, State 1, Line 6
Incorrect syntax near 'CAST'.
```

But the second line works just fine because it is already a complete string:

```
FilingName
-----
Ach
Abe
...
Zhe
Hu

(19972 row(s) affected)
```

EXEC and UDFs

In short, you can't get there from here. You are not allowed to use `EXEC` to run dynamic SQL within a UDF—period (using `EXEC` to run a sproc is, however, legal in a few cases).

Control-of-Flow Statements

Control-of-flow statements are a veritable must for any programming language these days. I can't imagine having to write my code where I couldn't change what commands to run depending on a condition.

Chapter 10

Given that we're assuming at least an intermediate knowledge of both programming and SQL, we're not going to dwell on these a lot, but since "intermediate" means different things to different people, we had best give these the once over.

T-SQL offers most of the classic choices for control-of-flow situations, including:

- IF...ELSE
- GOTO
- WHILE
- WAITFOR
- TRY/CATCH

We also have the CASE statement (a.k.a. SELECT CASE, DO CASE, and SWITCH/BREAK in other languages), but it doesn't have quite the level of control of flow capabilities that you've come to expect from other languages.

The **IF . . . ELSE** Statement

IF...ELSE statements work much as they do in any language, although I equate them most closely to C in the way they are implemented. The basic syntax is:

```
IF <Boolean Expression>
    <SQL statement> | BEGIN <code series> END
[ELSE
    <SQL statement> | BEGIN <code series> END]
```

The expression can be pretty much any expression that evaluates to a Boolean.

This brings us back to one of the most common traps that I see SQL programmers fall into—improper user of NULLs. I can't tell you how often I have debugged stored procedures only to find a statement like:

```
IF @myvar = NULL
```

This will, of course, never be true on most systems (see the exception shortly) and will wind up bypassing all their NULL values. Instead, it needs to read:

```
IF @myvar IS NULL
```

Don't forget that NULL doesn't equate to anything—not even NULL. Use IS instead of =

The exception to this is dependent on whether you have set the ANSI_NULLS option ON or OFF. The default is that this is ON, in which case you'll see the behavior described previously. You can change this behavior by setting ANSI_NULLS to OFF. I strongly recommend against this since it violates the ANSI standard (it's also just plain wrong).

Note that only the very next statement after the IF will be considered to be conditional (as per the IF). You can include multiple statements as part of your control-of-flow block using BEGIN...END, but we'll discuss that one a little later in the chapter.

To show off a simple version of this, let's run an example that's very common to build scripts. Imagine for a moment that we want to CREATE a table if it's not there, but to leave it alone if it already exists. We could make use of the EXISTS operator (you may recall my complaint that the Books Online calls EXISTS a keyword when I consider it an operator).

```
-- We'll run a SELECT looking for our table to start with to prove it's not there
SELECT 'Found Table ' + s.name + '.' + t.name
    FROM sys.schemas s
    JOIN sys.tables t
        ON s.schema_id = t.schema_id
    WHERE s.name = 'dbo'
        AND t.name = 'OurIFTest'

-- Now we're run our conditional CREATE statement
IF NOT EXISTS (
    SELECT s.name AS SchemaName, t.name AS TableName
        FROM sys.schemas s
        JOIN sys.tables t
            ON s.schema_id = t.schema_id
    WHERE s.name = 'dbo'
        AND t.name = 'OurIFTest'
)
    CREATE TABLE OurIFTest(
        Col1      int      PRIMARY KEY
    )

-- And now look again to prove that it's been created.
SELECT 'Found Table ' + s.name + '.' + t.name
    FROM sys.schemas s
    JOIN sys.tables t
        ON s.schema_id = t.schema_id
    WHERE s.name = 'dbo'
        AND t.name = 'OurIFTest'
```

The meat of this is in the middle—notice that our CREATE TABLE statement runs only if no matching table already exists.

(0 row(s) affected)

Found Table dbo.OurIFTest

(1 row(s) affected)

The ELSE Clause

Now this thing about being able to run a statement conditionally is just great, but it doesn't really deal with all the scenarios we might want to deal with. Quite often—indeed, most of the time—when we deal with an IF condition, we have specific statements we want to execute not just for the true condition, but also a separate set of statements that we want to run if the condition is false—or the ELSE condition.

You will run into situations where a Boolean cannot be evaluated—that is, the result is unknown (for example, if you are comparing to a `NULL`). Any expression that returns a result that would be considered as an unknown result will be treated as `FALSE`.

The `ELSE` statement works pretty much as it does in any other language. The exact syntax may vary slightly, but the nuts and bolts are still the same—the statements in the `ELSE` clause are executed if the statements in the `IF` clause are not.

To expand our earlier example just a bit, let's actually print a warning message out if we do not create our table:

```
-- Now we're run our conditional CREATE statement
IF NOT EXISTS (
    SELECT s.name AS SchemaName, t.name AS TableName
        FROM sys.schemas s
        JOIN sys.tables t
            ON s.schema_id = t.schema_id
    WHERE s.name = 'dbo'
        AND t.name = 'OurIFTTest'
    )
CREATE TABLE OurIFTTest(
    Col1      int          PRIMARY KEY
)
ELSE
    PRINT 'WARNING: Skipping CREATE as table already exists'
```

If you have already run the preceding example, then the table will already exist, and running this second example should get you the warning message:

```
WARNING: Skipping CREATE as table already exists
```

Grouping Code into Blocks

Sometimes you need to treat a group of statements as though they were all one statement (if you execute one, then you execute them all—otherwise, you don't execute any of them). For instance, the `IF` statement will, by default, consider only the very next statement after the `IF` to be part of the conditional code. What if you want the condition to require several statements to run? Life would be pretty miserable if you had to create a separate `IF` statement for each line of code you wanted to run if the condition holds.

Thankfully, like most any language with an `IF` statement, SQL Server gives us a way to group code into blocks that are considered to all belong together. The block is started when you issue a `BEGIN` statement and continues until you issue an `END` statement. It works like this:

```
IF <Expression>
BEGIN  --First block of code starts here—executes only if
      --expression is TRUE
    Statement that executes if expression is TRUE
    Additional statements
    ...
    ...
```

```

Still going with statements from TRUE expression
IF <Expression>    --Only executes if this block is active
BEGIN
    Statement that executes if both outside and inside
    expressions are TRUE
    Additional statements
    ...
    ...
Still statements from both TRUE expressions
END
Out of the condition from inner condition, but still
part of first block
END    --First block of code ends here
ELSE
BEGIN
    Statement that executes if expression is FALSE
    Additional statements
    ...
    ...
Still going with statements from FALSE expression
END

```

Notice our ability to nest blocks of code. In each case, the inner blocks are considered to be part of the outer block of code. I have never heard of there being a limit to how many levels deep you can nest your BEGIN...END blocks, but I would suggest that you minimize them. There are definitely practical limits to how deep you can keep them readable—even if you are particularly careful about the formatting of your code.

Just to put this notion into play, let's make yet another modification to table creation. This time, we're going to provide an informational message regardless of whether the table was created or not.

```

-- This time we're adding a check to see if the table DOES already exist
-- We'll remove it if it does so that the rest of our example can test the
-- IF condition. Just remove this first IF EXISTS block if you want to test
-- the ELSE condition below again.
IF EXISTS (
    SELECT s.name AS SchemaName, t.name AS TableName
    FROM sys.schemas s
    JOIN sys.tables t
        ON s.schema_id = t.schema_id
    WHERE s.name = 'dbo'
        AND t.name = 'OurIFTTest'
    )
DROP TABLE OurIFTTest

-- Now we're run our conditional CREATE statement
IF NOT EXISTS (
    SELECT s.name AS SchemaName, t.name AS TableName
    FROM sys.schemas s
    JOIN sys.tables t
        ON s.schema_id = t.schema_id
    WHERE s.name = 'dbo'
        AND t.name = 'OurIFTTest'
    )

```

```
BEGIN
    PRINT 'Table dbo.OurIFTTest not found.'
    PRINT 'CREATING: Table dbo.OurIFTTest'
    CREATE TABLE OurIFTTest(
        Col1      int      PRIMARY KEY
    )
END
ELSE
    PRINT 'WARNING: Skipping CREATE as table already exists'
```

Now, I've mixed all sorts of uses of the `IF` statement there. I have the most basic `IF` statement—with no `BEGIN...END` or `ELSE`. In my other `IF` statement, the `IF` portion uses a `BEGIN...END` block, but the `ELSE` does not.

I did this one this way just to illustrate how you can mix them. That said, I recommend you go back to my old axiom of “be consistent.” It can be really hard to deal with what statement is being controlled by what `IF...ELSE` condition if you are mixing the way you group things. In practice, if I’m using `BEGIN...END` on any statement within a given `IF`, then I use them for every block of code in that `IF` statement even if there is only one statement for that particular condition.

The CASE Statement

The `CASE` statement is, in some ways, the equivalent of one of several different statements depending on the language from which you’re coming. Statements in procedural programming languages that work in a similar way to `CASE` include:

- Switch: C, C++, C#, Delphi
- Select Case: Visual Basic
- Do Case: Xbase
- Evaluate: COBOL

I’m sure there are others—these are just from the languages that I’ve worked with in some form or another over the years. The big drawback in using a `CASE` statement in T-SQL is that it is, in many ways, more of a substitution operator than a control-of-flow statement.

There is more than one way to write a `CASE` statement—with an input expression or a Boolean expression. The first option is to use an input expression that will be compared with the value used in each `WHEN` clause. The SQL Server documentation refers to this as a *simple CASE*:

```
CASE <input expression>
WHEN <when expression> THEN <result expression>
[...n]
[ELSE <result expression>]
END
```

Option number two is to provide an expression with each `WHEN` clause that will evaluate to TRUE/FALSE. The docs refer to this as a *searched CASE*:

```
CASE
WHEN <Boolean expression> THEN <result expression>
[...n]
[ELSE <result expression>]
END
```

Perhaps what's nicest about `CASE` is that you can use it "inline" with (that is, as an integral part of) a `SELECT` statement. This can actually be quite powerful.

A Simple CASE

A simple `CASE` takes an expression that equates to a Boolean result. Let's get right to an example:

```
USE AdventureWorks
GO

SELECT TOP 10 SalesOrderID, SalesOrderID % 10 AS 'Last Digit', Position =
CASE SalesOrderID % 10
    WHEN 1 THEN 'First'
    WHEN 2 THEN 'Second'
    WHEN 3 THEN 'Third'
    WHEN 4 THEN 'Fourth'
    ELSE 'Something Else'
END
FROM Sales.SalesOrderHeader
```

For those of you who aren't familiar with it, the `%` operator is for a *modulus*. A modulus works in a similar manner to the divide by `(/)`, but it gives you only the remainder — therefore, $16 \% 4 = 0$ (4 goes into 16 evenly), but $16 \% 5 = 1$ (16 divided by 5 has a remainder of 1). In the example, since we're dividing by 10, using the modulus is giving us the last digit of the number we're evaluating.

Let's see what we got with this:

OrderID	Last Digit	Position
10249	9	Something Else
10251	1	First
10258	8	Something Else
10260	0	Something Else
10265	5	Something Else
10267	7	Something Else
10269	9	Something Else
10270	0	Something Else
10274	4	Fourth
10275	5	Something Else
(10 row(s) affected)		

Notice that whenever there is a matching value in the list, the `THEN` clause is invoked. Since we have an `ELSE` clause, any value that doesn't match one of the previous values will be assigned whatever we've put in our `ELSE`. If we had left the `ELSE` out, then any such value would be given a `NULL`.

Chapter 10

Let's go with one more example that expands on what we can use as an expression. This time, we'll use another column from our query:

```
USE AdventureWorks
GO

SELECT TOP 10 SalesOrderID % 10 AS 'OrderLastDigit',
       ProductID % 10 AS 'ProductLastDigit',
       "How Close?" = CASE SalesOrderID % 10
        WHEN ProductID % 1 THEN 'Exact Match!'
        WHEN ProductID % 1 - 1 THEN 'Within 1'
        WHEN ProductID % 1 + 1 THEN 'Within 1'
        ELSE 'More Than One Apart'
      END
FROM Sales.SalesOrderDetail
ORDER BY SalesOrderID DESC
```

Notice that we've used equations at every step of the way on this one, yet it still works. . . .

OrderLastDigit	ProductLastDigit	How Close?
2	5	More Than One Apart
3	2	More Than One Apart
3	9	More Than One Apart
3	8	More Than One Apart
2	2	More Than One Apart
2	8	More Than One Apart
1	7	Within 1
1	0	Within 1
1	1	Within 1
0	2	Exact Match!

(10 row(s) affected)

As long as the expression evaluates to a specific value that is of compatible type to the input expression, it can be analyzed, and the proper `THEN` clause applied.

A Searched CASE

This one works pretty much the same as a simple `CASE`, with only two slight twists:

- There is no input expression (remember that's the part between the `CASE` and the first `WHEN`).
- The `WHEN` expression must evaluate to a Boolean value (whereas in the simple `CASE` examples we've just looked at, we used values such as 1, 3, and `ProductID + 1`).

Perhaps what I find the coolest about this kind of `CASE` is that we can completely change around what is forming the basis of our expression—mixing and matching column expressions, depending on our different possible situations.

As usual, I find the best way to get across how this works is via an example:

```
SELECT TOP 10 SalesOrderID % 10 AS 'OrderLastDigit',
       ProductID % 10 AS 'ProductLastDigit',
       "How Close?" = CASE
           WHEN (SalesOrderID % 10) < 3 THEN 'Ends With Less Than Three'
           WHEN ProductID = 6 THEN 'ProductID is 6'
           WHEN ABS(SalesOrderID % 10 - ProductID) <= 1 THEN 'Within 1'
           ELSE 'More Than One Apart'
       END
FROM Sales.SalesOrderDetail
ORDER BY SalesOrderID DESC
```

This is substantially different from our simple CASE examples, but it still works:

OrderLastDigit	ProductLastDigit	How Close?
2	5	Ends With Less Than Three
3	2	More Than One Apart
3	9	More Than One Apart
3	8	More Than One Apart
2	2	Ends With Less Than Three
2	8	Ends With Less Than Three
1	7	Ends With Less Than Three
1	0	Ends With Less Than Three
1	1	Ends With Less Than Three
0	2	Ends With Less Than Three

(10 row(s) affected)

There are a few of things to pay particular attention to in how SQL Server evaluated things:

- ❑ Even when two conditions evaluate to TRUE, only the first condition is used. For example, the second-to-last row meets both the first (the last digit is smaller than 3) and third (the last digit is within 1 of the ProductID) conditions. For many languages, including Visual Basic, this kind of statement always works this way. If you're from the C world, however, you'll need to remember this when you are coding; no "break" statement is required — it always terminates after one condition is met.
- ❑ You can mix and match what fields you're using in your condition expressions. In this case, we used SalesOrderID, ProductID, and both together.
- ❑ You can perform pretty much any expression as long as, in the end, it evaluates to a Boolean result.

Let's try this out with a slightly more complex example. In this example, we're not going to do the mix-and-match thing — instead, we'll stick with just the one column we're looking at (we could change columns being tested — but, most of the time, we won't need to). Instead, we're going to deal with a more real-life scenario that I helped solve for a rather large e-commerce site.

Chapter 10

The scenario is this: Marketing people really like nice clean prices. They hate it when you apply a 10 percent markup over cost, and start putting out prices like \$10.13, or \$23.19. Instead, they like slick prices that end in numbers like 49, 75, 95, or 99. In our scenario, we're supposed to create a possible new price list for analysis, and they want it to meet certain criteria.

If the new price ends with less than 50 cents (such as our previous \$10.13 example), then marketing would like the price to be bumped up to the same dollar amount but ending in 49 cents (\$10.49 for our example). Prices ending with 50 cents to 75 cents should be changed to end in 75 cents, and prices ending with more than 75 cents should be changed to end with 95 cents. Let's take a look at some examples of what they want:

If the New Price Would Be	Then It Should Become
\$10.13	\$10.49
\$17.57	\$17.75
\$27.75	\$27.75
\$79.99	\$79.95

Technically speaking, we could do this with nested `IF...ELSE` statements, but:

- It would be much harder to read — especially if the rules were more complex.
- We would have to implement the code using a cursor (*bad!*) and examine each row one at a time.

In short — *yuck!*

A `CASE` statement is going to make this process relatively easy. What's more, we're going to be able to place our condition inline to our query and use it as part of a set operation — this almost always means that we're going to get much better performance than we would with a cursor.

Our marketing department has decided they would like to see what things would look like if we increased prices by 10 percent, so we'll plug a 10 percent markup into a `CASE` statement, and, together with a little extra analysis, we'll get the numbers we're looking for:

```
USE AdventureWorks
GO

/* I'm setting up some holding variables here. This way, if we get asked
** to run the query again with a slightly different value, we'll only have
** to change it in one place.
*/
DECLARE @Markup      money
DECLARE @Multiplier   money

SELECT @Markup = .10          -- Change the markup here
SELECT @Multiplier = @Markup + 1    -- We want the end price, not the amount
                                    -- of the increase, so add 1
```

```

/* Now execute things for our results. Note that we're limiting things
** to the top 10 items for brevity—in reality, we either wouldn't do this
** at all, or we would have a more complex WHERE clause to limit the
** increase to a particular set of products
*/
SELECT TOP 10 ProductID, Name, ListPrice,
       ListPrice * @Multiplier AS "Marked Up Price", "New Price" =
CASE WHEN FLOOR(ListPrice * @Multiplier + .24)
      > FLOOR(ListPrice * @Multiplier)
           THEN FLOOR(ListPrice * @Multiplier) + .95
WHEN FLOOR(ListPrice * @Multiplier + .5) >
      FLOOR(ListPrice * @Multiplier)
           THEN FLOOR(ListPrice * @Multiplier) + .75
ELSE FLOOR(ListPrice * @Multiplier) + .49
END
FROM Production.Product
WHERE ProductID % 10 = 0 -- this is just to help the example
ORDER BY ProductID DESC

```

The FLOOR function you see here is a pretty simple one—it takes the value supplied and rounds down to the nearest integer.

Now, I don't know about you, but I get very suspicious when I hear the word “analysis” come out of someone's lips—particularly if that person is in a marketing or sales role. Don't get me wrong—those people are doing their jobs just like I am. The thing is, once they ask a question one way, they usually want to ask the same question another way. That being the case, I went ahead and set this up as a script—now all we need to do when they decide they want to try it with 15 percent is make a change to the initialization value of @Markup. Let's see what we got this time with that 10 percent markup though:

ProductID	Name	ListPrice	Marked Up Price	New Price
990	Mountain-500 Black, 42	539.99	593.989	593.9500
980	Mountain-400-W Silver, 38	769.49	846.439	846.4900
970	Touring-2000 Blue, 46	1214.85	1336.335	1336.4900
960	Touring-3000 Blue, 62	742.35	816.585	816.7500
950	ML Crankset	256.49	282.139	282.4900
940	HL Road Pedal	80.99	89.089	89.4900
930	HL Mountain Tire	35.00	38.50	38.7500
920	LL Mountain Frame - Silver, 52	264.05	290.455	290.4900
910	HL Mountain Seat/Saddle	52.64	57.904	57.9500
900	LL Touring Frame - Yellow, 50	333.42	366.762	366.9500

(10 row(s) affected)

Look these over for a bit, and you'll see that the results match what we were expecting. What's more, we didn't have to build a cursor to do it.

Looping with the WHILE Statement

The WHILE statement works much as it does in other languages to which you have probably been exposed. Essentially, a condition is tested each time you come to the top of the loop. If the condition is still TRUE, then the loop executes again—if not, you exit.

Chapter 10

The syntax looks like this:

```
WHILE <Boolean expression>
    <sql statement> |
[BEGIN
    <statement block>
    [BREAK]
    <sql statement> | <statement block>
    [CONTINUE]
END]
```

While you can just execute one statement (much as you do with an `IF` statement), you'll almost never see a `WHILE` that isn't followed by a `BEGIN...END` with a full statement block.

The `BREAK` statement is a way of exiting the loop without waiting for the bottom of the loop to come and the expression to be re-evaluated.

I'm sure I won't be the last to tell you this, but using a `BREAK` is generally thought of as something of bad form in the classical sense. I tend to sit on the fence on this one. I avoid using them if reasonably possible. Most of the time, I can indeed avoid them just by moving a statement or two around, while still coming up with the same results. The advantage of this is usually more readable code. It is simply easier to handle a looping structure (or any structure for that matter) if you have a single point of entry and a single exit. Using a `BREAK` violates this notion.

All that being said, sometimes you can actually make things worse by reformatting the code to avoid a `BREAK`. In addition, I've seen people write much slower code for the sake of not using a `BREAK` statement — bad idea.

The `CONTINUE` statement is something of the complete opposite of a `BREAK` statement. In short, it tells the `WHILE` loop to go back to the beginning. Regardless of where you are in the loop, you immediately go back to the top and re-evaluate the expression (exiting if the expression is no longer `TRUE`).

We'll go ahead and do something of a short example here just to get our feet wet. As I mentioned before, `WHILE` loops tend to be rare in non-cursor situations, so forgive me if this example seems lame.

What we're going to do is create something of a monitoring process using our `WHILE` loop and a `WAITFOR` command (we'll look at the specifics of `WAITFOR` in our next section). We're going to be automatically updating our statistics once per day:

```
WHILE 1 = 1
BEGIN
    WAITFOR TIME '01:00'
    EXEC sp_updatestats
    RAISERROR('Statistics Updated for Database', 1, 1) WITH LOG
END
```

This would update the statistics for every table in our database every night at 1 AM and write a log entry of that fact to both the SQL Server log and the Windows application log. If you want check to see if this works, leave this running all night and then check your logs in the morning.

Note that using an infinite loop like this isn't the way that you would normally want to schedule a task. If you want something to run every day, set up a job using Management Studio. In addition to not keeping a connection open all the time (which the preceding example would do), you also get the capability to make follow up actions dependent on the success or failure of your script. Also, you can e-mail or netsend messages regarding the completion status.

The WAITFOR Statement

There are often things that you either don't want to or simply can't have happen right this moment, but you also don't want to have to hang around waiting for the right time to execute something.

No problem—use the WAITFOR statement and have SQL Server wait for you. The syntax is incredibly simple:

```
WAITFOR
    DELAY <'time'> | TIME <'time'>
```

The WAITFOR statement does exactly what it says it does—that is, it waits for whatever you specify as the argument to occur. You can specify either an explicit time of day for something to happen, or you can specify an amount of time to wait before doing something.

The **DELAY** Parameter

The DELAY parameter choice specifies an amount of time to wait. You cannot specify a number of days—just time in hours, minutes, and seconds. The maximum allowed delay is 24 hours. So, for example:

```
WAITFOR DELAY '01:00'
```

would run any code prior to the WAITFOR, then reach the WAITFOR statement, and stop for one hour, after which execution of the code would continue with whatever the next statement was.

The **TIME** Parameter

The TIME parameter choice specifies to wait until a specific time of day. Again, we cannot specify any kind of date—just the time of day using a 24-hour clock. Once more, this gives us a one-day time limit for the maximum amount of delay. For example:

```
WAITFOR TIME '01:00'
```

would run any code prior to the WAITFOR, then reach the WAITFOR statement, and stop until 1 AM, after which execution of the code would continue with whatever the next statement was after the WAITFOR.

TRY/CATCH Blocks

In days of yore (meaning anything before SQL Server 2005), our error-handling options were pretty limited. Indeed, we could check for error conditions, but we had to do so proactively. Indeed, in some cases we could have errors that would cause us to leave our procedure or script without an opportunity to trap it at all (indeed, this can still happen). We're going to save a more full discussion of error handling for our stored procedures discussion in Chapter 11, but we'll touch on the fundamentals of the new TRY/CATCH blocks here.

Chapter 10

A TRY/CATCH block in SQL Server works remarkably similarly to those used in any C derived language (C, C++, C#, Delphi, and a host of others). The syntax looks like this:

```
BEGIN TRY
    { <sql statement(s)> }
END TRY
BEGIN CATCH
    { <sql statement(s)> }
END CATCH [ ; ]
```

In short, SQL Server will “try” to run anything within the BEGIN...END that goes with your TRY block. If, and only if, you have an error condition that has an error level of 11–19 occurs, then SQL Server will exit the TRY block immediately and begin with the first line in your CATCH block. Since there are more possible error levels than just 11–19, take a look at what we have there:

Error Level	Nature
1–10	Informational only. This would include things like context changes such as settings being adjusted or NULL values found while calculating aggregates. These will not trigger a CATCH block, so if you need to test for this level of error, you'll need to do so manually by checking @@ERROR.
11–19	Relatively severe errors, but ones that can be handled by your code (foreign key violations, as an example). Some of these can be severe enough that you are unlikely to want to continue processing (such as a memory exceeded error), but at least you can trap them and exit gracefully.
20–25	Very severe. These are generally system-level errors. Your server-side code will never know this kind of error happened, as the script and connection will be terminated immediately.

Keep these in mind—if you need to handle errors outside the 11–19 level range, then you'll need to make other plans.

Now, to test this out, we'll make some alterations to our CREATE script that we built back when we were looking at IF...ELSE statements. You may recall that part of the reason for our original test to see whether the table already existed was to avoid creating an error condition that might have caused our script to fail. That kind of test is the way things have been done historically (and there really wasn't much in the way of other options). With the advent of TRY/CATCH blocks, we could just try the CREATE and then handle the error if one were given:

```
BEGIN TRY
    -- Try and create our table
    CREATE TABLE OurIFTTest(
        Col1      int          PRIMARY KEY
    )
END TRY
BEGIN CATCH
    -- Uh oh, something went wrong, see if it's something
    -- we know what to do with
    DECLARE @ErrorNo      int,
            @Severity     tinyint,
```

```

        @State      smallint,
        @LineNo     int,
        @Message    nvarchar(4000)
SELECT
        @ErrorNo = ERROR_NUMBER(),
        @Severity = ERROR_SEVERITY(),
        @State = ERROR_STATE(),
        @LineNo = ERROR_LINE (),
        @Message = ERROR_MESSAGE()

IF @ErrorNo = 2714 -- Object exists error, we knew this might happen
    PRINT 'WARNING: Skipping CREATE as table already exists'
ELSE -- hmm, we don't recognize it, so report it and bail
    RAISERROR(@Message, 16, 1 )
END CATCH

```

Notice I used some special functions to retrieve the error condition, so let's take a look at those.

Also note that I moved them into variables that were controlled by me so they would not be lost.

Function	Returns
ERROR_NUMBER()	The actual error number. If this is a system error, there will be a entry in the sysmessages table (use sys.messages to look it up) that matches to that error and contains some of the information you'll get from the other error-related functions.
ERROR_SEVERITY()	This equates to what is sometimes called "error level" in other parts of this book and Books Online. My apologies for the inconsistency—I'm guilty of perpetuating something that Microsoft started doing a version or two ago.
ERROR_STATE()	I use this as something of a place mark. This will always be 1 for system errors. When I discuss error handling in more depth in the next chapter, you'll see how to raise your own errors. At that point, you can use state to indicate things like at what point in your stored procedure, function, or trigger the error occurred (this helps with situations where a given error can be handled in any one of many places).
ERROR_PROCEDURE()	We did not use this in the preceding example, as it is only relevant to stored procedures, functions, and triggers. This supplies the name of the procedure that caused the error—very handy if your procedures are nested at all, as the procedure that causes the error may not be the one to actually handle that error.
ERROR_LINE()	Just what it says—the line number of the error.
ERROR_MESSAGE()	The text that goes with the message. For system messages, this is the same as what you'll see if you select the message from the sys.messages function. For user-defined errors, it will be the text supplied to the RAISERROR function.

In our example, I utilized a known error id that SQL Server raises if we attempt to create an object that already exists. You can see all system error messages by selecting them from the sys.messages table function.

Particularly with SQL Server 2005, the sys.messages output has gotten so lengthy that it's hard to find what you're looking for by just scanning it. My solution is less than elegant but is rather effective—I just artificially create the error I'm looking for and see what error number it gives me (simple solutions for simple minds like mine!).

I simply execute the code I want to execute (in this case, the CREATE statement) and handle the error if there is one—there really isn't much more to it than that.

We will look at error handling in a far more thorough fashion in Chapter 11. In the meantime, you can use TRY/CATCH to give basic error handling to your scripts.

Summary

Understanding scripts and batches is the cornerstone to an understanding of programming with SQL Server. The concepts of scripts and batches lay the foundation for a variety of functions from scripting complete database builds to programming stored procedures and triggers.

Local variables have scope for only one batch. Even if you have declared the variable within the same overall script, you will still get an error message if you don't re-declare it (and start over with assigning values) before referencing it in a new batch.

There are over 30 system functions. We provided a listing of some of the most useful system functions, but there are many more. Try checking out the Books Online or Appendix A at the back of this book for some of the more obscure ones. System functions do not need to be declared, and are always available. Some are scoped to the entire server, while others return values specific to the current connection.

You can use batches to create precedence between different parts of your scripts. The first batch starts at the beginning of the script and ends at the end of the script or the first GO statement—whichever comes first. The next batch (if there is another) starts on the line after the first one ends and continues to the end of the script or the next GO statement—again, whichever comes first. The process continues to the end of the script. The first batch from the top of the script is executed first, the second is executed second, and so on. All commands within each batch must pass validation in the query parser, or none of that batch will be executed; however, any other batches will be parsed separately and will still be executed (if they pass the parser).

In addition, we saw how we do have constructs to deal with control of flow and error-handling conditions. We can use this to build complex scripts that are able to adapt to different runtime environments (such as recognizing that it needs to process an upgrade of a database instead of an installation, or even determine what version of your schema it is upgrade from).

Finally, we also saw how we can create and execute SQL dynamically. This can afford us the opportunity to deal with scenarios that aren't always 100 percent predictable or situations where something we need to construct our statement is actually itself a piece of data.

In the next couple of chapters, we will take the notions of scripting and batches to the next level, and apply them to stored procedures and triggers—the closest things that SQL Server has to actual programs. We will also see how we can utilize any .NET language to add more complex language functionality to our stored procedures, functions, and triggers.

11

Getting Procedural: Stored Procedures and User-Defined Functions

OK, so here we are. We added a little bit of meat to the matter in terms of being able to actually code things in the last chapter, but this is where we start to get serious about the “programmer” aspect of things.

Let me digress for a moment though—temper your excitement about procedural work as relates to SQL. The reality is that the most important part of your work is probably already done—the design of your database. The way you build your schema is the very foundation of everything else that you do, so be careful to take the time you need in that seemingly simple stuff. Far too many database developers just throw together some tables and focus their time on the procedural logic—bad choice. As in all things in life, balance is the key.

Things have gotten more exciting from a “what you can” do perspective. In addition to the T-SQL programming constructs that we’ve always had, we now add .NET to the picture. In general, T-SQL will be the way we want to do things, but now we have the flexibility of adding .NET assemblies to not only create more complex procedures but also to create our own complex data types.

In this chapter, we’re going to review how to create a basic stored procedure (sproc) and user-defined function (UDF) from the core elements of SQL Server, and then move quickly onto more complex procedural constructs, debugging. After we finish with the basic objects of SQL Server, we will be set to add in the .NET element in Chapter 14 and get an idea of some of the new power it makes available to us.

Creating the Sproc: Basic Syntax

Creating a sproc works pretty much the same as creating any other object in a database, except that it uses the AS keyword we used with views. The basic syntax looks like this:

```
CREATE PROCEDURE|PROC <sproc name>
    [<parameter name> [schema.]<data type> [VARYING] [= <default value>] [OUT
    [PUT]] [,]
        <parameter name> [schema.]<data type> [VARYING] [= <default value>]
    [OUT[PUT]] [,]
        ...
        ...
    ]
    [WITH
        RECOMPILE| ENCRYPTION | [EXECUTE AS { CALLER|SELF|OWNER|<'user name'>}]
    [FOR REPLICATION]
    AS
        <code> | EXTERNAL NAME <assembly name>.<assembly class>
```

As you can see, you still have the basic CREATE <Object Type> <Object Name> syntax that is the backbone of every CREATE statement. The only oddity here is the choice between PROCEDURE and PROC. Either option works fine, but as always, I recommend that you be consistent regarding which one you choose. (Personally, I like the saved keystrokes of PROC, and in my experience, that's the way most people do it.) The name of your sproc must follow the rules for naming as outlined in Chapter 1.

After the name comes a list of parameters. Parameterization is optional, and I defer that discussion until a little later in the chapter.

Last, but not least, comes your actual code following the AS keyword.

An Example of a Basic Sproc

Perhaps the best example of basic sproc syntax is found in the most basic of sprocs—a sproc that returns all the columns in all the rows on a table—in short, everything to do with a table's data.

I hope that, by now, you have the query that returns all the contents of a table down cold (Hint: SELECT * FROM....)

```
USE AdventureWorks
GO
CREATE PROC spEmployee
AS
    SELECT * FROM HumanResources.Employee
```

Not too rough, eh?

Getting Procedural: Stored Procedures and User-Defined Functions

Now that you have your sproc created, execute it to see what you get:

```
EXEC spEmployee
```

You get exactly what you would have gotten if you had run the `SELECT` statement that's embedded in the sproc.

Changing Stored Procedures with ALTER

`ALTER` statements for sprocs work almost identically to views from the standpoint of what an `ALTER` statement does.

The main thing to remember when you edit sprocs with T-SQL is that you are completely replacing the existing sproc. The only differences between using the `ALTER PROC` statement and the `CREATE PROC` statement are as follows:

- ❑ `ALTER PROC` expects to find an existing sproc, whereas `CREATE` doesn't.
- ❑ `ALTER PROC` retains any permissions that have been established for the sproc. It keeps the same object ID within system objects and allows the dependencies to be kept. For example, if procedure A calls procedure B and you drop and re-create procedure B, you no longer see the dependency between the two. If you use `ALTER`, it's all still there.
- ❑ `ALTER PROC` retains any dependency information on other objects that may call the sproc being altered.

If you perform a `DROP` and then use a `CREATE`, you have almost the same effect as using an `ALTER PROC` statement with one rather big difference: if you `DROP` and `CREATE`, you need to reestablish your permissions for who can and can't use the sproc. In addition, SQL Server loses track of the dependency information for any procedures, views, triggers, and functions that depended on the sproc before you dropped it.

Dropping Sprocs

It doesn't get much easier than this:

```
DROP PROC | PROCEDURE <sproc name>
```

And it's gone.

Parameterization

A stored procedure gives you some (or in the case of .NET, a lot of) procedural capability, and also gives you a performance boost in the form of precompiled code that has already been run through the optimizer, but it wouldn't be much help in most circumstances if it couldn't accept some data to tell it what to do. Likewise, you often want to get information out of the sproc — not just one or more recordsets of table data but also information that is more direct. An example here might be if you update several records in a table and you want to know just how many you updated. Often, this isn't easily handed back in recordset form, so you make use of an *output parameter*.

From outside the sproc, parameters can be passed in either by position or by reference. From the inside, it doesn't matter which way they come in. They are declared the same either way.

Declaring Parameters

Declaring a parameter requires two to four of these pieces of information:

- The name
- The data type
- The default value
- The direction

The syntax follows:

```
@parameter_name [AS] data_type [= default|NULL] [VARYING] [OUTPUT|OUT]
```

The rules for naming parameters are basically the same as those for a variable. The data type also matches the rules of a variable.

One special thing in declaring the data type is to remember that, when declaring a parameter of type CURSOR, you must also use the VARYING and OUTPUT options. The use of this type of parameter is pretty unusual, but keep it in mind.

The default is the first place you start to see any real divergence from variables. Whereas variables are always initialized to a NULL value, parameters aren't. Indeed, if you don't supply a default value, the parameter is assumed to be required, and an initial value must be supplied when the sproc is called or an error will be generated. To supply a default, you simply add an = sign after the data type and provide the default value. After you do this, the users of your sproc can decide to supply no value for that parameter, or they can provide their own value.

So, for example, if we wanted to optionally limit our previous example to just those employees whose last name started with a set of provided characters, we could do that by accepting a `Lastname` parameter:

```
ALTER PROC spEmployee
    @LastName nvarchar(50) = NULL
AS
    IF @LastName IS NULL
```

Getting Procedural: Stored Procedures and User-Defined Functions

```
SELECT * FROM HumanResources.Employee
ELSE
    SELECT c.LastName, c.FirstName, e.*
    FROM Person.Contact c
    JOIN HumanResources.Employee e
        ON c.ContactID = e.ContactID
    WHERE c.LastName LIKE @LastName + '%'
```

Notice how I have made use of the control of flow constructs we learned in the previous chapter on scripting to decide what is the better query to run. In this case, we might consider the procedure to be overloaded in the sense that it really will produce totally different output (the column lists are different) depending on whether a parameter was supplied or not.

Creating Output Parameters

Sometimes, you want to pass non-recordset information out to whatever called your sproc.

Perhaps one of the most common uses for this is with sprocs that do inserts into tables with identity values. Often the code calling the sproc wants to know what the identify value was when the process is complete.

To show this off, we'll utilize a stored procedure that is already in the AdventureWorks database—`uspLogError`. It looks like this:

```
-- uspLogError logs error information in the ErrorLog table about the
-- error that caused execution to jump to the CATCH block of a
-- TRY...CATCH construct. This should be executed from within the scope
-- of a CATCH block otherwise it will return without inserting error
-- information.
CREATE PROCEDURE [dbo].[uspLogError]
    @ErrorLogID [int] = 0 OUTPUT -- contains the ErrorLogID of the row inserted
AS
    -- by uspLogError in the ErrorLog table
BEGIN
    SET NOCOUNT ON;

    -- Output parameter value of 0 indicates that error
    -- information was not logged
    SET @ErrorLogID = 0;

    BEGIN TRY
        -- Return if there is no error information to log
        IF ERROR_NUMBER() IS NULL
            RETURN;

        -- Return if inside an uncommittable transaction.
        -- Data insertion/modification is not allowed when
        -- a transaction is in an uncommittable state.
        IF XACT_STATE() = -1
            BEGIN
                PRINT 'Cannot log error since the current transaction is in an
uncommittable state. '
```

Chapter 11

```
+ 'Rollback the transaction before executing uspLogError in order
to successfully log error information.';
      RETURN;
END

INSERT [dbo].[ErrorLog]
(
[UserName],
[ErrorNumber],
[ErrorSeverity],
[ErrorState],
[ErrorProcedure],
[ErrorLine],
[ErrorMessage]
)
VALUES
(
CONVERT(sysname, CURRENT_USER),
ERROR_NUMBER(),
ERROR_SEVERITY(),
ERROR_STATE(),
ERROR_PROCEDURE(),
ERROR_LINE(),
ERROR_MESSAGE()
);

-- Pass back the ErrorLogID of the row inserted
SET @ErrorLogID = @@IDENTITY;
END TRY
BEGIN CATCH
    PRINT 'An error occurred in stored procedure uspLogError: ';
    EXECUTE [dbo].[uspPrintError];
    RETURN -1;
END CATCH
END;
```

Note the sections that I've highlighted here—these are the core to our output parameter. The first declares the parameter as being an output parameter. The second makes the insert that utilizes the identity value, and finally the SET statement captures the identity value. When the procedure exists, the value in @ErrorLogID is passed to the calling script.

Let's utilize our TRY/CATCH example from the tail end of the last chapter, but this time we'll make the call to uspLogError:

```
USE AdventureWorks

BEGIN TRY
-- Try and create our table
CREATE TABLE OurIFTTest(
    Col1    int      PRIMARY KEY
)
END TRY
BEGIN CATCH
```

Getting Procedural: Stored Procedures and User-Defined Functions

```
-- Uh oh, something went wrong, see if it's something
-- we know what to do with
DECLARE @MyOutputParameter int

IF ERROR_NUMBER() = 2714 -- Object exists error, we knew this might happen
BEGIN
    PRINT 'WARNING: Skipping CREATE as table already exists'
    EXEC dbo.uspLogError @ErrorLogID = @MyOutputParameter OUTPUT
    PRINT 'A error was logged. The Log ID for our error was '
        + CAST(@MyOutputParameter AS varchar)
END
ELSE      -- hmm, we don't recognize it, so report it and bail
    RAISERROR('something not good happened this time around', 16, 1 )
END CATCH
```

If you run this in a database that does not already have the `OurIFTTest` table, then you will get a simple:

```
Command(s) completed successfully.
```

But run it where the `OurIFTTest` table already exists (for example, run it twice if you haven't run the `CREATE` code before), and you get something to indicate the error:

```
WARNING: Skipping CREATE as table already exists
A error was logged. The Log ID for our error was 3
```

Now run a little select against the error log table:

```
SELECT *
FROM ErrorLog
WHERE ErrorLogID = 3 -- change this value to whatever your
                      -- results said it was logged as
```

And you can see that the error was indeed properly logged:

ErrorLogID	UserName	ErrorMessage
3	dbo	There is already an object named 'OurIFTTest' ...

```
(1 row(s) affected)
```

There are several things that you should take note of between the sproc itself and the usage of it by the calling script:

- ❑ The `OUTPUT` keyword is required for the output parameter in the sproc declaration.
- ❑ You must use the `OUTPUT` keyword when you call the sproc, much as you did when you declared the sproc. This gives SQL Server advance warning about the special handling that parameter will require. Be aware, however, that forgetting to include the `OUTPUT` keyword won't create a runtime error. You won't get any messages about it, but the value for the output parameter won't be moved into your variable. You wind up with what was already there, most likely a `NULL` value. This means that you'll have what I consider to be the most dreadful of all computer terms: unpredictable results.

- ❑ The variable you assign the output result to does *not* have to have the same name as the internal parameter in the sproc. For example, in the previous sproc, the internal parameter was called @LogErrorID, but the variable the value was passed to was called @MyOutputVariable.
- ❑ The EXEC (or EXECUTE) keyword was required because the call to the sproc wasn't the first thing in the batch. You can leave off the EXEC if the sproc call is the first thing in a batch, but I recommend that you train yourself to use it regardless.

Confirming Success or Failure with Return Values

Return values are used in a couple of different ways. The first is to actually return data, such as an identity value or the number of rows that the sproc affected. Consider this an evil practice from the dark ages. Instead, move on to the way that return values should be used and what they are really there for: determining the execution status of your sproc.

If it sounds like I have an opinion about how return values should be used, it's because I most definitely do. I was actually originally taught to use return values as a "trick" to get around having to use output parameters, in effect, as a shortcut. Happily, I overcame this training. The problem is that, like most shortcuts, you're cutting something out, and, in this case, what you're cutting out is rather important.

Using return values as a means of returning data to your calling routine clouds the meaning of the return code when you need to send back honest-to-goodness error codes. In short, don't go there!

Return values are all about indicating success or failure of the sproc and even the extent or nature of that success or failure. For the C programmers among you, this should be a fairly easy strategy to relate to. It's common practice to use a function's return value as a success code, with any non-zero value indicating some sort of problem. If you stick with the default return codes in SQL Server, you'll find that the same rules hold true.

How to Use RETURN

Actually, your program will receive a return value whether you supply one or not. By default, SQL Server automatically returns a value of zero when your procedure is complete.

To pass a return value from your sproc to the calling code, you simply use the RETURN statement:

```
RETURN [<integer value to return>]
```

Note that the return value must be an integer.

Perhaps the biggest thing to understand about the RETURN statement is that it unconditionally exits from your sproc. No matter where you are in your sproc, not one single more line of code executes after you issue a RETURN statement.

To illustrate this idea of how a RETURN statement affects things, write a very simple test sproc:

Getting Procedural: Stored Procedures and User-Defined Functions

```
USE AdventureWorks
GO

CREATE PROC spTestReturns
AS
    DECLARE @MyMessage      varchar(50)
    DECLARE @MyOtherMessage varchar(50)

    SELECT @MyMessage = 'Hi, it''s that line before the RETURN'
    PRINT @MyMessage
    RETURN
    SELECT @MyOtherMessage = 'Sorry, but we won''t get this far'
    PRINT @MyOtherMessage
    RETURN
```

Now you have a sproc, but you need a small script to test a couple of things:

- What gets printed?
- What value does the RETURN statement return?

To capture the value of a RETURN statement, you need to assign it to a variable during your EXEC statement. For example, the following code would assign whatever the return value is to @ReturnVal:

```
EXEC @ReturnVal = spMySproc
```

Now put this into a more useful script to test your sproc:

```
DECLARE @Return int

EXEC @Return = spTestReturns
SELECT @Return
```

Short but sweet. When you run it, you'll see that the RETURN statement did indeed terminate the code before anything else could run:

```
Hi, it's that line before the RETURN
-----
0
(1 row(s) affected)
```

You also got the return value for your sproc, which was zero. Notice that the value was zero even though you didn't specify a specific return value. That's because the default is always zero.

Think about this for a minute. If the return value is zero by default, the default return is also, in effect, "No Errors." This has some serious dangers to it. Make sure that you always explicitly define your return values. That way, you are reasonably certain to be returning the value you intended rather than something by accident.

Chapter 11

Now, just for grins, alter that sproc to verify that you can send any integer value you want as the return value:

```
USE AdventureWorks
GO

ALTER PROC spTestReturns
AS
    DECLARE @MyMessage      varchar(50)
    DECLARE @MyOtherMessage varchar(50)

    SELECT @MyMessage = 'Hi, it''s that line before the RETURN'
    PRINT @MyMessage
    RETURN 100
    SELECT @MyOtherMessage = 'Sorry, but we won''t get this far'
    PRINT @MyOtherMessage
RETURN
```

Now rerun your test script, and you'll get the same result save for that change in return value:

```
Hi, it's that line before the RETURN
-----
100
(1 row(s) affected)
```

Dealing with Errors

Sure, you don't need this section. I mean, your code never has errors, and you never run into problems, right? OK, well, now that you've had your moment of fantasy for today, get down to reality. Things go wrong. It's just the way that life works in the wonderful world of software engineering. Fortunately, you can do something about it. Unfortunately, you're often not going to be happy with the tools you have—SQL Server now has much improved error handling in the form of TRY/CATCH, but it still isn't quite everything you might be accustomed to from procedural languages. Fortunately again, there are ways to make the most out of what you have, and ways to hide many of the inadequacies of error handling in the SQL world.

Four common types of errors can happen in SQL Server:

- ❑ Errors that create runtime errors and stop your code from proceeding further.
- ❑ Errors that informational in nature and do not create runtime errors. A non-zero error number is returned (if you ask), but no error is raised (and so no error trapping will be activated unless you are testing for that specific error).
- ❑ Errors that create runtime errors but continue execution within SQL Server such that you can trap them and respond in the manner of your choosing.
- ❑ Errors that are more logical in nature and to which SQL Server is essentially oblivious.

Now, here things get a bit sticky, and versions become important, so hang with me as I lead you down a winding road.

Getting Procedural: Stored Procedures and User-Defined Functions

Error handling in the SQL Server 2005 era is a fair bit different from in prior versions. Today, we have genuine error traps in the form of TRY/CATCH blocks. There is, as you might expect, backward compatibility, but you can do much more in SQL Server 2005 than you could before. One thing remains common between the old and new error-handling models: higher-level runtime errors.

Some general errors cause SQL Server to terminate the script immediately. This was true prior to TRY/CATCH, and it remains true even in the TRY/CATCH era. Errors that have enough severity to generate a runtime error are problematic from the SQL Server side of the equation. The new TRY/CATCH logic is a bit more flexible for some errors than the logic that preceded it, but even now your sproc won't necessarily know when something bad happens (it just depends how bad "bad" is). On the bright side, all the current data access object models pass through the message on such errors, so you know about them in your client application and can do something about them there.

The Way We Were

In prior versions of SQL Server, there was no formal error handler. You didn't have an option that essentially said, "If any error happens, go run this code over in this other spot." Instead, you had to monitor for error conditions within your own code and then decide what to do at the point you detected the error — possibly well after the actual error occurred.

Handling Inline Errors

Inline errors are those pesky little things where SQL Server keeps running as such, but hasn't, for some reason, succeeded in doing what you wanted it to do. For example, try to insert a record into the Sales.StoreContact table that doesn't have a corresponding record in the Customers or Contacts table:

```
USE AdventureWorks
GO

INSERT INTO Sales.StoreContact
    (CustomerID, ContactID, ContactTypeID)
VALUES
    (0, 0, 11)
```

SQL Server won't perform this insert for you because there is a FOREIGN KEY constraint on both CustomerID and ContactID that references other tables. Since there is not a matching record in both tables, the record we are trying to insert into Sales.StoreContact violates both of those foreign key constraints and is rejected:

```
Msg 547, Level 16, State 0, Line 2
The INSERT statement conflicted with the FOREIGN KEY constraint
"FK_Individual_Contact_ContactID". The conflict occurred in database
"AdventureWorks", table "Person.Contact", column 'ContactID'.
The statement has been terminated.
```

Pay attention to that error 547 up there. That's something you can use.

It's worth noting that just the first foreign key violation shows up in the error SQL Server provided — this is because SQL Server got to that error, saw it, and knew there was no point in going further. If we fixed the first error, then the second would be detected and we would again error out.

Using @@ERROR

`@@ERROR` contains the error number of the last T-SQL statement executed. If the value is zero, then no error occurred. This is somewhat similar to the `ERROR_NUMBER()` function we saw in the last chapter when we first discussed TRY/CATCH blocks. While `ERROR_NUMBER()` is only valid and remains the same regardless of where you are within a CATCH block, `@@ERROR` receives a new value with each statement you execute.

The caveat with `@@ERROR` is that it is reset with each new statement. This means that if you want to defer analyzing the value, or you want to use it more than once, you need to move the value into some other holding bin—a local variable that you have declared for this purpose.

Play with this just a bit using the `INSERT` example from before:

```
USE AdventureWorks
GO

DECLARE    @Error      int

-- Bogus INSERT - there is no CustomerID or ContactID of 0. Either of
-- these could cause the error we see when running this statement.
INSERT INTO Sales.Individual
    (CustomerID, ContactID)
VALUES
    (0,0)

-- Move our error code into safekeeping. Note that, after this statement,
-- @Error will be reset to whatever error number applies to this statement
SELECT @Error = @@ERROR

-- Print out a blank separator line
PRINT ''

-- The value of our holding variable is just what we would expect
PRINT 'The Value of @Error is ' + CONVERT(varchar, @Error)

-- The value of @@ERROR has been reset - it's back to zero
PRINT 'The Value of @@ERROR is ' + CONVERT(varchar, @@ERROR)
```

Now execute your script, and you can examine how `@@ERROR` is affected:

```
Msg 547, Level 16, State 0, Line 6
The INSERT statement conflicted with the FOREIGN KEY constraint
"FK_Individual_Contact_ContactID". The conflict occurred in database
"AdventureWorks", table "Person.Contact", column 'ContactID'.
The statement has been terminated.
```

```
The Value of @Error is 547
The Value of @@ERROR is 0
```

Getting Procedural: Stored Procedures and User-Defined Functions

This illustrates pretty quickly the issue of saving the value from @@ERROR. The first error statement is only informational in nature. SQL Server has thrown that error, but hasn't stopped the code from executing. Indeed, the only part of that message that your sproc has access to is the error number. That error number resides in @@ERROR for just that next T-SQL statement; after that it's gone.

Notice that @Error and @@ERROR are two separate and distinct variables, and can be referred to separately. This isn't just because of the case difference. (Depending on how you have your server configured, case sensitivity can affect your variable names.) It's because of the difference in scope. The @ or @@ is part of the name, so the number of @ symbols on the front makes each one separate and distinct from the other.

Using @@ERROR in a Sproc

OK, so let's start with an assumption here: If you're using @@ERROR, then the likelihood is that you are not using TRY/CATCH blocks. If you have not made this choice for backward compatibility reasons, I'm going to bop you upside the head and suggest you reconsider — TRY/CATCH is the much cleaner and all around better way.

TRY/CATCH will handle varieties of errors that in previous versions would have ended your script execution.

That said, TRY/CATCH is out of the equation if backward compatibility with SQL Server 2000 or prior is what you need, so let's take a quick look.

What we're going to do is look at two short procedures. Both are based on things we have already done in scripting or in earlier stored procedure examples, but we want to take a look at how inline error checking works when it works, and how it doesn't when it doesn't (in particular, when inline does not work, but TRY/CATCH would).

Let's start with the referential integrity example we did earlier in this chapter:

```
USE AdventureWorks
GO

INSERT INTO Sales.StoreContact
    (CustomerID, ContactID, ContactTypeID)
VALUES
    (0, 0, 11)
```

You may recall this got us a simple 547 error. This is one of those that is trappable. We could trap this in a simple script, but let's do it as a sproc since procedural stuff is supposedly what we're working on here....

```
USE AdventureWorks
GO

CREATE PROC spInsertValidatedStoreContact
```

Chapter 11

```
@CustomerID int,
@ContactID int,
@ContactTypeID int
AS
BEGIN

    DECLARE @Error int

    INSERT INTO Sales.StoreContact
        (CustomerID, ContactID, ContactTypeID)
    VALUES
        (@CustomerID, @ContactID, @ContactTypeID)

    SET @Error = @@ERROR

    IF @Error = 0
        PRINT 'New Record Inserted'
    ELSE
        BEGIN
            IF @Error = 547 -- Foreign Key violation. Tell them about it.
                PRINT 'At least one provided parameter was not found. Correct and
retry'
            ELSE -- something unknown
                PRINT 'Unknown error occurred. Please contact your system admin'
        END
    END
END
```

Now try executing this with values that work:

```
EXEC spInsertValidatedStoreContact 1, 1, 11
```

Our insert happens correctly, so no error condition is detected (because there isn't one).

```
(1 row(s) affected)
New Record Inserted
```

Now, try something that should blow up:

```
EXEC spInsertValidatedStoreContact 0, 1, 11
```

And you see not only the actual SQL Server message but the message from our error trap (note that there is no way of squelching the SQL Server message).

```
Msg 547, Level 16, State 0, Procedure spInsertValidatedStoreContact, Line 11
The INSERT statement conflicted with the FOREIGN KEY constraint
"FK_StoreContact_Store_CustomerID". The conflict occurred in database
"AdventureWorks", table "Sales.Store", column 'CustomerID'.
The statement has been terminated.
At least one provided parameter was not found. Correct and retry
```

As you can see, we were able to detect our error without a TRY/CATCH block.

Getting Procedural: Stored Procedures and User-Defined Functions

Now, let's move on to an example of why TRY/CATCH is better—a situation where a TRY/CATCH works fine, but where inline error checking fails. To show this one off, all we need to do is use our example for TRY/CATCH that we used in the scripting chapter. It looked like this:

```
BEGIN TRY
    -- Try and create our table
    CREATE TABLE OurIFTest(
        Col1      int          PRIMARY KEY
    )
END TRY
BEGIN CATCH
    -- Uh oh, something went wrong, see if it's something
    -- we know what to do with
    DECLARE @ErrorNo      int,
            @Severity     tinyint,
            @State        smallint,
            @LineNo       int,
            @Message      nvarchar(4000)
    SELECT
        @ErrorNo = ERROR_NUMBER(),
        @Severity = ERROR_SEVERITY(),
        @State = ERROR_STATE(),
        @LineNo = ERROR_LINE (),
        @Message = ERROR_MESSAGE()

    IF @ErrorNo = 2714 -- Object exists error, we knew this might happen
        PRINT 'WARNING: Skipping CREATE as table already exists'
    ELSE -- hmm, we don't recognize it, so report it and bail
        RAISERROR(@Message, 16, 1 )
END CATCH
```

It worked just fine. But if I try and do this using inline error checking, I have a problem:

```
CREATE TABLE OurIFTest(
    Col1      int          PRIMARY KEY
)
IF @@ERROR != 0
    PRINT 'Problems!'
ELSE
    PRINT 'Everything went OK!'
```

Run this (you'll need to run it twice to generate the error if the table isn't already there), and we quickly find out that, without the TRY block, SQL Server aborts the script entirely on the particular error we're generating here:

```
Msg 2714, Level 16, State 6, Line 2
There is already an object named 'OurIFTest' in the database.
```

Notice that our PRINT statements never got a chance to execute—SQL Server had already terminated processing. With TRY/CATCH we were able to trap and handle this error, but using inline error checking, our attempts to trap an error like this fail.

Manually Raising Errors

Sometimes you have errors that SQL Server doesn't really know about, but you wish it did. For example, perhaps in the previous example you don't want to return -1000. Instead, you'd like to be able to create a runtime error at the client end that the client would then use to invoke an error handler and act accordingly. To do this, you use the RAISERROR command in T-SQL. The syntax is pretty straightforward:

```
RAISERROR (<message ID | message string>, <severity>, <state>
[, <argument>
[,<...n>]] )
[WITH option[,...n]]
```

Message ID/Message String

The message ID or message string you provide determines which message is sent to the client.

Using a message ID creates a manually raised error with the ID that you specified and the message that is associated with that ID as found in the sysmessages table in the master database.

*If you want to see what your SQL Server has as predefined messages, you can always perform a SELECT * FROM master..sysMessages. This includes any messages you've manually added to your system using the sp_addmessage stored procedure or through Enterprise Manager.*

You can also just supply a message string in the form of ad hoc text without creating a more permanent message in sysmessages:

```
RAISERROR ('Hi there, I''m an error', 1, 1)
```

This raises a rather simple error message:

```
Msg 50000, Level 1, State 50000
Hi there, I'm an error
```

Notice that the assigned message number, even though you didn't supply one, is 50000. This is the default error value for any ad hoc error. It can be overridden using the WITH SETERROR option.

Severity

We got a quick overview of this when looking at TRY/CATCH in the chapter on scripting. For those of you already familiar with Windows servers, severity should be an old friend. Severity is an indication of just how bad things really are based on this error. For SQL Server, however, what severity codes mean can get a little bizarre. They can range from informational (severities 1–18), to system level (19–25), and even catastrophic (20–25). If you raise an error of severity 19 or higher (system level), the WITH LOG option must also be specified. 20 and higher automatically terminates the users' connections. (They *hate* that!)

So, get back to what I meant by bizarre. SQL Server actually varies its behavior into more ranges than NT does, or even than the Books Online will tell you about. Errors fall into six major groupings:

Getting Procedural: Stored Procedures and User-Defined Functions

1–10	Purely informational but will return the specific error code in the message information.
11–16	If you do not have a TRY/CATCH block set up, then these terminate execution of the procedure and raise an error at the client. The state is shown to be whatever value you set it to. If you have a TRY/CATCH block defined, then that handler will be called rather than raising an error at the client.
17	Usually, only SQL Server should use this severity. Basically, it indicates that SQL Server has run out of resources—for example tempdb was full—and can't complete the request. Again, a TRY/CATCH block will get this before the client does.
18–19	Both of these are severe errors and imply that the underlying cause requires system administrator attention. With 19, the WITH LOG option is required, and the event will show up in the NT or Windows Event Log if you are using that OS family. These are the final levels at which you can trap the error with a TRY/CATCH block—after this, it will go straight to the client.
20–25	Your world has just caved in as has the user's connection. Essentially, this is a fatal error. The connection is terminated. As with 19, you must use the WITH LOG option, and a message will, if applicable, show up in the Event Log.

State

State is an ad hoc value. It's something that recognizes that exactly the same error may occur at multiple places within your code. The notion is that this gives you an opportunity to send something of a place marker for where exactly the error occurred.

State values can be between 1 and 127. If you are troubleshooting an error with Microsoft tech support, they apparently have some arcane knowledge that hasn't been shared with us about what some of these mean. I'm told that if you make a tech support call to Microsoft, they are likely to ask about and make use of this state information.

Error Arguments

Some predefined errors accept arguments. These allow you to make the error to be somewhat more dynamic by changing to the specific nature of the error. You can also format your error messages to accept arguments.

When you want to use dynamic information in what is otherwise a static error message, you need to format the fixed portion of your message so that it leaves room for the parameterized section of the message. You do so by using placeholders. If you're coming from the C or C++ world, then you'll recognize the parameter placeholders immediately; they are similar to the printf command arguments. If you're not from the C world, these may seem a little odd to you. All the placeholders start with the % sign and are then coded for the kind of information you'll be passing to them, as shown in the following table.

Chapter 11

Placeholder Type Indicator	Type of Value
D	Signed integer. Books Online indicates that i is an acceptable choice, but I've had problems getting it to work as expected.
O	Unsigned octal.
P	Pointer.
S	String.
U	Unsigned integer.
x or X	Unsigned hexadecimal.

In addition, there is the option to prefix any of these placeholder indicators with some additional flag and width information:

Flag	What It Does
- (dash or minus sign)	Left-justify. Only makes a difference when you supply a fixed width.
+ (plus sign)	Indicates the positive or negative nature if the parameter is a signed numeric type.
0	Tells SQL Server to pad the left side of a numeric value with zeroes until it reaches the width specified in the width option.
# (pound sign)	Applies only to octal and hex values. Tells SQL Server to use the appropriate prefix (0 or 0x) depending on whether it is octal or hex.
' '	Pads the left of a numeric value with spaces if positive.

Last, but not least, you can also set the width, precision, and long/short status of a parameter:

- ❑ **Width**—Set by simply supplying an integer value for the amount of space you want to hold for the parameterized value. You can also specify a *, in which case SQL Server will automatically determine the width according to the value you've set for precision.
- ❑ **Precision**—Determines the maximum number of digits output for numeric data.
- ❑ **Long/Short**—Set by using an h (short) or l (long) when the type of the parameter is an integer, octal, or hex value.

Use this in an example:

```
RAISERROR ('This is a sample parameterized %s, along with a zero
padding and a sign%+010d',1,1, 'string', 12121)
```

Getting Procedural: Stored Procedures and User-Defined Functions

If you execute this, you get back something that looks a little different from what's in the quotation marks:

```
This is a sample parameterized string, along with a zero
padding and a sign+000012121
Msg 50000, Level 1, State 1
```

The extra values supplied were inserted, in order, into your placeholders, with the final value being reformatted as specified.

WITH <option>

Currently, you can mix and match three options when you raise an error:

- LOG
- SETERROR
- NOWAIT

WITH LOG

This tells SQL Server to log the error to the SQL Server error log and the Windows Application Log. This option is required with severity levels that are 19 or higher.

WITH SETERROR

By default, a RAISERROR command doesn't set @@ERROR with the value of the error you generated. Instead, @@ERROR reflects the success or failure of your actual RAISERROR command. SETERROR overrides this and sets the value of @@ERROR to be equal to your error ID.

WITH NOWAIT

Immediately notifies the client of the error.

Adding Your Own Custom Error Messages

You can make use of a special system stored procedure to add messages to the system. The sproc is called `sp_addmessage`, and the syntax looks like this:

```
sp_addmessage [@msgnum =] <message id>,
[@severity =] <severity>,
[@msgtext =] <'msg'>
[, [@lang =] <'language'>]
[, [@with_log =] [TRUE|FALSE]]
[, [@replace =] 'replace']
```

All the parameters mean pretty much the same thing that they did with RAISERROR, except for the addition of the language and replace parameters and a slight difference with the WITH LOG option.

@lang

This specifies the language to which this message applies. What's cool here is that you can specify a separate version of your message for any language supported. This equates to the alias if you select the language list from `sys.syslanguages`.

@with_log

This works just the same as it does in `RAISERROR` in that, if set to `TRUE` the message will be automatically logged to the SQL Server error log and the Windows Application Log when raised. The only trick here is that you indicate that you want this message to be logged by setting this parameter to `TRUE` rather than using the `WITH LOG` option.

Be careful of this one in the Books Online. Depending on how you read it, it would be easy to interpret it as saying that you should set @with_log to a string constant of 'WITH_LOG', when you should set it to TRUE. Perhaps even more confusing is that the REPLACE option looks much the same, and it must be set to the string constant rather than TRUE.

@replace

If you are editing an existing message rather than creating a new one, you must set the `@replace` parameter to `'REPLACE'`. If you leave this off, you'll get an error if the message already exists.

Creating a set list of additional messages for use by your applications can greatly enhance reuse, but more importantly, it can significantly improve readability of your application. Imagine if every one of your database applications made use of a constant list of custom error codes. You could then easily establish a constants file (a resource or include library, for example) that had a listing of the appropriate errors. You could even create an include library that had generic handling of some or all of the errors. In short, if you're going to be building multiple SQL Server applications in the same environment, consider using a set list of errors that is common to all your applications.

Using sp_addmessage

As has already been indicated, `sp_addmessage` creates messages in much the same way as you create ad hoc messages using `RAISERROR`.

As an example, add your own custom message that tells users about the issues with their order date:

```
sp_addmessage
    @msgnum = 60000,
    @severity = 10,
    @msgtext = '%s is not a valid Order date.
Order date must be within 7 days of current date.'
```

Execute the sproc and it confirms the addition of the new message:

```
(1 row(s) affected)
```

No matter what database you're working with when you run `sp_addmessage`, the actual message is added to the `sysmessages` table in the master database. The significance of this is that, if you migrate your database to a new server, the messages need to be added again to that new server. The old ones will still be in the master database of the old server. As such, I strongly recommend keeping all your custom messages stored in a script somewhere, so they can easily be added into a new system.

Removing an Existing Custom Message

To get rid of the custom message, use the following:

```
sp_dropmessage <msg num>
```

What a Sproc Offers

Now that you spent some time looking at how to build a sproc, you probably want to ask the question about why you'd use them. Some of the reasons are pretty basic; others may not come to mind right away if you're new to the RDBMS world. Using sprocs has the following primary benefits:

- Enables you to create processes that require procedural action callable
- Enhances security
- Increases performance

Creating Callable Processes

As I've already indicated, a sproc is sort of a script that is stored in the database. The nice thing is that, because it's a database object, you can call to it. You don't have to manually load it from a file before executing it.

Sprocs can call to other sprocs (called *nesting*). With SQL Server 2005, you can nest up to 32 levels deep. This gives you the capability to reuse separate sprocs much as you would use a subroutine in a classic procedural language. The syntax for calling one sproc from another sproc is exactly the same as it is calling the sproc from a script.

Note that local variables are just that: local to each sproc. You can have five different copies of `@MyDate`, one each in five different sprocs, and they are all independent of each other.

Using Sprocs for Security

Many people don't realize the full use of sprocs as a tool for security. As with views, you can create a sproc that returns a recordset without having to give the user authority to the underlying table. Granting someone the right to execute a sproc implies that they can perform any action within the sproc, provided that the action is taken within the context of the sproc. That is, if you grant someone authority to execute

a sproc that returns all the records in the `Customers` table, but not access to the actual `Customers` table, the user will still be able to get data out of the `Customers` table provided he or she uses the sproc to do it. Trying to access the table directly won't work.

What can be really handy here is that you can give someone access to modify data through the sproc but give only read access to the underlying table. The user will be able use the sproc to modify data in the table, which will likely be enforcing some business rules. The user can then hook directly up to your SQL Server using Excel, Access, or whatever to build a custom reports with no risk of "accidentally" modifying the data.

Setting users up to directly link to a production database via Access or Excel has to be one of the most incredibly powerful and yet stupid things you can do to your system. While you are empowering your users, you are also digging your own grave in terms of the resources they will use and the long-running queries they will execute. (Naturally, they will be oblivious to the havoc this causes your system.)

If you really must give users direct access, consider using replication or backup and restores to create a completely separate copy of the database for them to use. This helps ensure against record locks, queries that bog down the system, and a whole host of other problems.

Sprocs and Performance

Generally speaking, sprocs can do a lot to help the performance of your system. Keep in mind though that, like most things in life, there are no guarantees. Indeed, some processes can be created in sprocs that will substantially slow the process if the sproc hasn't been designed intelligently.

Where does that performance come from? Well, when you create a sproc, the process works as diagrammed in Figure 11-1.

You start by running your `CREATE PROC` procedure. This parses the query to make sure that the code should actually run. The one difference between this and running the script directly is that the `CREATE PROC` command can make use of deferred name resolution. *Deferred name resolution* ignores the fact that you may have some objects that don't exist yet. This gives you the chance to create these objects later.

After the sproc has been created, it waits for the first time that it is executed. At that time, the sproc is optimized, and a query plan is compiled and cached on the system. Subsequent times that you run it, your sproc uses that cached query plan rather than create a new one, unless you specify otherwise using the `WITH RECOMPILE` option. This means that whenever the sproc is used, it can skip much of the optimization and compilation process. Exactly how much time this saves varies depending on the complexity of the batch, the size of the tables involved in the batch, and the number of indexes on each table. Usually, the amount of time saved is seemingly small — say, perhaps one second for most scenarios — yet that difference can really add up in terms of percentage. (One second is 100 percent faster than two seconds.) The difference can become even more extreme when you need to make several calls or when you are in a looping situation.

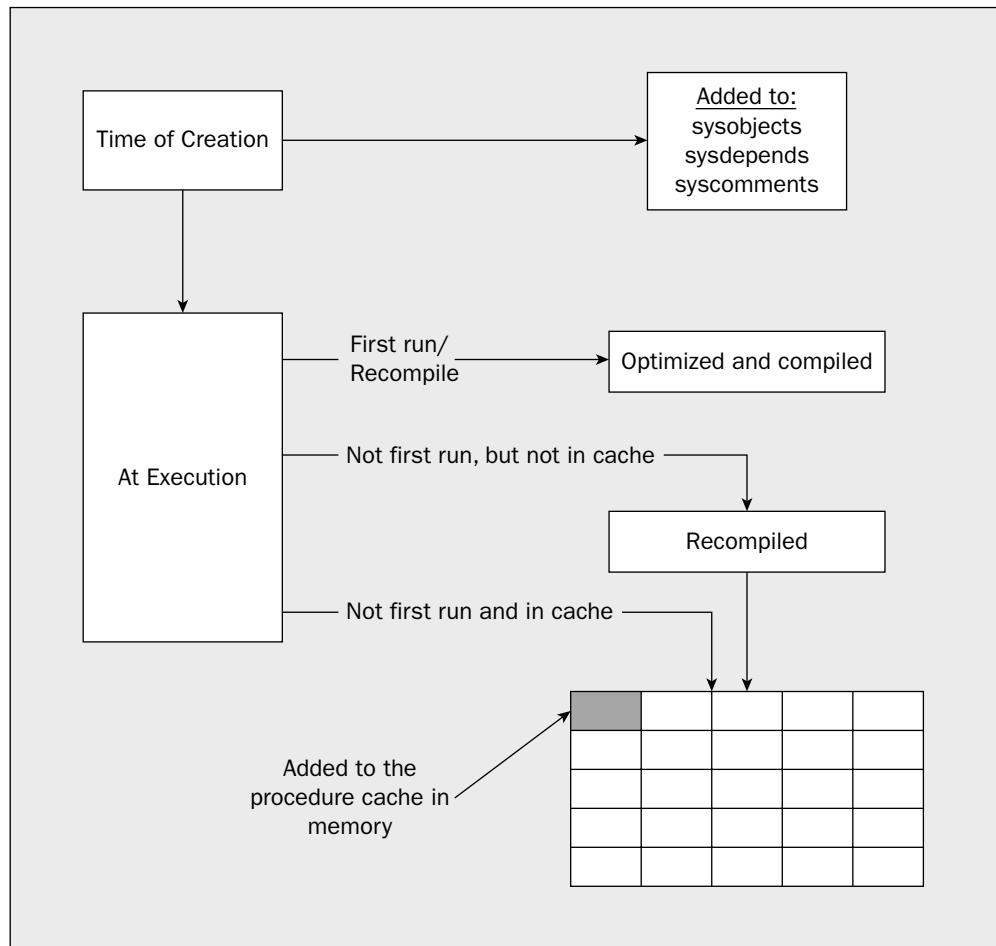


Figure 11-1

When a Good Sproc Goes Bad

Perhaps one of the most important things to recognize on the downside of sprocs is that unless you manually interfere (using the `WITH RECOMPILE` option) they are optimized based on either the first time that they run, or when the statistics have been updated on the table(s) involved in any queries.

That “optimize once, use many times” strategy saves the sproc time, but it’s a double-edged sword. If your query is dynamic in nature (the query is built up as it goes using the `EXEC` command), the sproc may be optimized for the way things ran the first time, only to find that things never run that way again. In short, it may be using the wrong plan!

It’s not just dynamic queries in sprocs that can cause this scenario either. Imagine a Web page that lets you mix and match several criteria for a search. For example, say that you wanted to add a sproc to the

Chapter 11

AdventureWorks database that would support a Web page that enables users to search for an order based on the following criteria:

- Customer ID
- Order ID
- Product ID
- Order date

The user is allowed to supply any mix of the information, with each new piece of information supplied making the search a little more restricted and theoretically faster.

The approach you would probably take to this would be to have more than one query and select the right query to run depending on what was supplied by the user. The first time that you execute your sproc, it's going to run through a few IF . . . ELSE statements and pick the right query to run. Unfortunately, it's just the right query for that particular time and for an unknown percentage of the other times. Any time after that first time the sproc selects a different query to run, it will still be using the query plan based on the first time it ran. In short, the query performance is really going to suffer.

Using the WITH RECOMPILE Option

You can choose to use the security and compartmentalization of code benefits of a sproc but still ignore the precompiled code side of things. This lets you get around this issue of not using the right query plan because you're certain that a new plan was created just for this run. To do this, you use the WITH RECOMPILE option, which can be included in two different ways.

First, you can include WITH RECOMPILE at runtime. You simply include it with your execution script:

```
EXEC spEmployee  
    WITH RECOMPILE
```

This tells SQL Server to throw away the existing execution plan, and create a new one, but just this once—that is, just for this time that you executed the sproc using the WITH RECOMPILE option.

You can also make things more permanent by including the WITH RECOMPILE option within the sproc. If you do things this way, you add the WITH RECOMPILE option immediately before your AS statement in your CREATE PROC or ALTER PROC statements.

If you create your sproc with this option, the sproc is recompiled each time it runs, regardless of other options chosen at runtime.

Extended Stored Procedures (XPs)

The advent of .NET in SQL Server has really changed the area of extended stored procedures. These used to be the bread and butter of the “hard core” code scenarios, useful when you hit those times where basic T-SQL and the other features of SQL Server just wouldn't give you what you needed.

Getting Procedural: Stored Procedures and User-Defined Functions

With the advent of .NET to deal with things like O/S file access and other external communication or complex formulas, the day of the XP would seem to be waning. XPs still have their teeny tiny place in the world for times where performance is so critical that you want the code running genuinely in process to SQL Server, but this is truly a *radical* approach in the .NET era.

A Brief Look at Recursion

Recursion is one of those things that aren't used very often in programming. Still when you need it, there never seems to be anything else that will quite do the trick. As a just-in-case, a brief review of what recursion is seems in order.

The brief version is that *recursion* is the situation where a piece of code calls itself. The dangers here should be fairly self-evident. If it calls itself once, then what's to keep it from calling itself over and over again? The answer to that is *you*. That is, *you* need to make sure that if your code is going to be called recursively, you provide a *recursion check* to make sure you bail out when it's appropriate.

I'd love to say that the example I'm going to use is neat and original, but it isn't. Indeed, for an example, I'm going to use the classic recursion example that's used with about every textbook recursion discussion I've ever seen. Please accept my apologies; it's just that it's an example that just about anyone can understand, so here we go.

The classic example uses factorials. A *factorial* is the value you get when you take a number and multiply it successively by that number less one, then the next value less one, and so on until you get to 1. For example, the factorial of 5 is 120—that's $5*4*3*2*1$.

Look at an implementation of such a recursive sproc:

```
CREATE PROC spFactorial
@ValueIn int,
@ValueOut int OUTPUT
AS
DECLARE @InWorking int
DECLARE @OutWorking int
IF @ValueIn != 1
BEGIN
    SELECT @InWorking = @ValueIn - 1
    EXEC spFactorial @InWorking, @OutWorking OUTPUT
    SELECT @ValueOut = @ValueIn * @OutWorking
END
ELSE
BEGIN
    SELECT @ValueOut = 1
END
RETURN
GO
```

Chapter 11

So, what you're doing is accepting a value in (that's the value you want a factorial of) and providing a value back out (the factorial value you've computed). The surprising part is that your sproc doesn't, in one step, do everything it needs to calculate the factorial. Instead, it just takes one number's worth of the factorial and then turns around and calls itself. The second call deals with just one number's worth and then again calls itself. This can go on and on up to a limit of 32 levels of recursion. When SQL Server gets 32 levels deep, it raises an error and ends processing.

Note that any calls into .NET assemblies count as an extra level in your recursion count, but anything you do within those assemblies doesn't count against the recursion limit. If you consider this for a moment, you should be able to see how you could use .NET assemblies to get around the recursion limit in some scenarios. For example, if the core of this sproc was moved into the .NET assembly, the number of times you recurse within the .NET side of things would be limited only by memory considerations.

Try out the recursive sproc with a little script:

```
DECLARE @WorkingOut int  
DECLARE @WorkingIn int  
SELECT @WorkingIn = 5  
EXEC spFactorial @WorkingIn, @WorkingOut OUTPUT  
  
PRINT CAST(@WorkingIn AS varchar) + ' factorial is ' + CAST(@WorkingOut AS varchar)
```

This gets the expected result of 120:

```
5 factorial is 120
```

You can try different values for @WorkingIn, and things should work just fine with two rather significant hitches:

- ❑ Arithmetic overflow when your factorial grows too large for the int (or even bigint) data type
- ❑ The 32-level recursion limit

You can test the arithmetic overflow easily by putting any large number in; anything bigger than about 13 will work for this example.

Testing the 32-level recursion limit takes a little bit more modification to your sproc. This time, you'll determine the *triangular* of the number. This is similar to finding the factorial, except that you use addition rather than multiplication. Therefore, 5 triangular is 15 ($5 + 4 + 3 + 2 + 1$). Create a new sproc to test this one. It will look almost just like the factorial sproc with only a few small changes:

```
CREATE PROC spTriangular  
@ValueIn int,  
@ValueOut int OUTPUT  
AS  
DECLARE @InWorking int  
DECLARE @OutWorking int
```

Getting Procedural: Stored Procedures and User-Defined Functions

```
IF @ValueIn != 1
BEGIN
    SELECT @InWorking = @ValueIn - 1

    EXEC spTriangular @InWorking, @OutWorking OUTPUT

    SELECT @ValueOut = @ValueIn + @OutWorking
END
ELSE
BEGIN
    SELECT @ValueOut = 1
END
RETURN
GO
```

As you can see, there weren't that many changes to be made. Similarly, you only need to change your sproc call and the `PRINT` text for your test script:

```
DECLARE @WorkingOut int
DECLARE @WorkingIn int
SELECT @WorkingIn = 5
EXEC spTriangular @WorkingIn, @WorkingOut OUTPUT

PRINT CAST(@WorkingIn AS varchar) + ' Triangular is ' + CAST(@WorkingOut AS
varchar)
```

Running this with a `@ValueIn` of 5 gets the expected 15:

```
5 Triangular is 15
```

However, if you try to run it with a `@ValueIn` of more than 32, you get an error:

```
Msg 217, Level 16, State 1, Procedure spTriangular, Line 12
Maximum stored procedure, function, trigger, or view nesting level exceeded (limit
32).
```

I'd love to say there's some great workaround to this, but, unless you can somehow segment your recursive calls (run it 32 levels deep, come all the way back out of the call stack, and run down it again), you're pretty much out of luck. Just keep in mind that most recursive functions can be rewritten to be a more standard looping construct, which doesn't have any hard limit. Be sure you can't use a loop before you force yourself into recursion.

User-Defined Functions (UDFs)

Well, here we are already at one of my favorite topics. Five years after their introduction, user-defined functions—or UDFs—remain one of the more underutilized and misunderstood objects in SQL Server. In short, these were awesome when Microsoft first introduced them in SQL Server 2000, and the advent of .NET just adds some extra “oomph” to them.

What a UDF Is

A *user-defined function* is, much like a sproc, an ordered set of T-SQL statements that are pre-optimized and compiled and can be called to work as a single unit. The primary difference between them is how results are returned. Because of things that need to happen in order to support these different kinds of returned values, UDFs have a few more limitations to them than sprocs do.

OK, so I've said what a UDF is, so I suspect I ought to take a moment and say what it is not. A UDF is definitely NOT a replacement for a sproc—they are just a different option that offers us yet one more form of code flexibility.

There are two types of UDFs:

- ❑ Those that return a scalar value
- ❑ Those that return a table

Let's take a look at the general syntax for creating a UDF:

```
CREATE FUNCTION [<schema name>.]<function name>
    ( [ <@parameter name> [AS] [<schema name>.]<scalar data type> [ = <default
value>]
    [ ,...n ] ] )
RETURNS {<scalar type>|TABLE [(<Table Definition>)]}
    [ WITH [ENCRYPTION] | [SCHEMABINDING] |
    [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ] | [EXECUTE AS {
CALLER|SELF|OWNER|<'user name'>} ]
]
[AS] { EXTERNAL NAME <external method> |
BEGIN
    [<function statements>]
    {RETURN <type as defined in RETURNS clause>|RETURN (<SELECT statement>)}
END }[;]
```

This is kind of a tough one to explain because parts of the optional syntax are dependent on the choices you make elsewhere in your CREATE statement. The big issues here are whether you are returning a scalar data type or a table and whether you're doing a T-SQL based function or doing something utilizing the CLR and .NET. Let's look at the core SQL Server type individually (we will explore .NET further in Chapter 14).

UDFs Returning a Scalar Value

This type of UDF is probably the most like what you might expect a function to be. Much like most of SQL Server's own built-in functions, they will return a scalar value to the calling script or procedure; functions such as GETDATE() or USER() return scalar values.

Since with a UDF you are highly focused on the return value (instead of a result set or output variable), you are not limited to an integer for a return value—instead, it can be of any valid SQL Server data type (including user-defined data types!), except for BLOBs, cursors, and timestamps. Even if you wanted to return an integer, a UDF should look very attractive to you for two different reasons:

Getting Procedural: Stored Procedures and User-Defined Functions

- ❑ Unlike sprocs, the whole purpose of the return value is to serve as a meaningful piece of data — for sprocs, a return value is meant as an indication of success or failure, and, in the event of failure, to provide some specific information about the nature of that failure.
- ❑ You can perform functions in-line to your queries (for instances, include it as part of your SELECT statement) — you can't do that with a sproc.

So, that said, let's create a simple UDF.

One of the most common function-like requirements I see is a desire to see if an entry in a datetime field occurred on a specific day. The usual problem here is that your datetime field has specific time-of-day information that prevents it from easily being compared with just the date. Indeed, we've already seen this problem in some of our comparisons in previous chapters.

Let's go back to the Accounting database that we created in an earlier chapter. Imagine for a moment that we want to know all the orders that came in today. Let's start by adding a few orders in with today's date. We'll just pick customer and employee IDs we know already exist in their respective tables (if you don't have any records there, you'll need to insert a couple of dummy rows to reference). I'm also going to create a small loop to add in several rows:

```
USE Accounting

DECLARE @Counter int

SET @Counter = 1
WHILE @Counter <= 10
BEGIN
    INSERT INTO Orders
        VALUES (1, DATEADD(mi,@Counter,GETDATE()), 1)
    SET @Counter = @Counter + 1
END
```

Note that you must have matching rows in the Customers and Employees tables. This shouldn't be an issue if you did all the examples in Chapters 4 and 5, but unless you have a CustomerNo of 1 in the Customers table and a EmployeeID of 1 in the Employees table, you will get foreign key violations (from the foreign keys we added in Chapter 5).

So, this gets us 10 rows inserted, with each row being inserted with today's date, but one minute apart from each other.

OK, if you're running this a split second before midnight, some of the rows may dribble over into the next day, so be careful — but it will work fine for everyone except the night owls.

So, now we're ready to run a simple query to see what orders we have today. We might try something like:

```
SELECT *
FROM Orders
WHERE OrderDate = GETDATE()
```

Unfortunately, this query will not get us anything back at all. This is because GETDATE() gets the current time down to the millisecond — not just the day. This means that any query based on GETDATE() is very

Chapter 11

unlikely to return us any data—even if it happened on the same day (it would have had to have happened within in the same minute for a `smalldatetime`, and within a millisecond for a full `datetime` field).

The typical solution is to convert the date to a string and back in order to truncate the time information, and then perform the comparison.

It might look something like:

```
SELECT *
FROM Orders
WHERE CONVERT(varchar(12), OrderDate, 101) = CONVERT(varchar(12), GETDATE(), 101)
```

This time, we will get back every row with today's date in the `OrderDate` column—regardless of what time of day the order was taken. Unfortunately, this isn't exactly the most readable code. Imagine you had a large series of dates you needed to perform such comparisons against—it can get very ugly indeed.

So now let's look at doing the same thing with a simple user-defined function. First, we'll need to create the actual function. This is done with the new `CREATE FUNCTION` command, and it's formatted much like a sproc. For example, we might code this function like this:

```
CREATE FUNCTION dbo.DayOnly(@Date datetime)
RETURNS varchar(12)
AS
BEGIN
    RETURN CONVERT(varchar(12), @Date, 101)
END
```

where the date returned from `GETDATE()` is passed in as the parameter, and the task of converting the date is included in the function body and the truncated date is returned.

To see this function in action, let's reformat our query slightly:

```
SELECT *
FROM Orders
WHERE dbo.DayOnly(OrderDate) = dbo.DayOnly(GETDATE())
```

We get back the same set as with the standalone query. Even for a simple query like this one, the new code is quite a bit more readable. The call works pretty much as it would from most languages that support functions. There is, however, one hitch—the schema is required. SQL Server will, for some reason, not resolve functions the way it does with other objects.

As long as you're returning a scalar value, functions are very flexible as to how you obtain that value. You could, for example, use a function to encapsulate a subquery that does a single value lookup.

UDFs That Return a Table

User-defined functions in SQL Server are not limited to just returning scalar values. They can return something far more interesting—tables. Now, while the possible impacts of this are sinking in on you,

Getting Procedural: Stored Procedures and User-Defined Functions

I'll go ahead and add that the table that is returned is, for the most part, usable much as any other table is. You can perform a `JOIN` against it and even apply `WHERE` conditions against the results. It's *very* cool stuff indeed.

To make the change to using a table as a return value is not hard at all—a table is just like any other SQL Server data type as far as a UDF is concerned. To illustrate this, we'll build a relatively simple one to start:

```
USE AdventureWorks
GO

CREATE FUNCTION dbo.fnContactList()
RETURNS TABLE
AS
RETURN (SELECT ContactID,
            LastName + ', ' + FirstName AS Name,
            EmailAddress AS email
       FROM Person.Contact)
GO
```

This function returns a table of SELECTED records and does a little formatting: joining the last and first names, separating them with a comma.

At this point, we're ready to use our function just as we would use a table:

```
SELECT *
FROM fnContactList()
```

Now, let's add a bit more fun into things. What we did with this table up to this point could have been done just as easily—easier in fact—with a view. But what if we wanted to parameterize a view? What if, for example, we wanted to accept last name input to filter our results (without having to manually put in our own `WHERE` clause)? It might look something like this:

```
CREATE FUNCTION dbo.fnContactSearch(@LastName nvarchar(50))
RETURNS TABLE
AS
RETURN (SELECT ContactID,
            LastName + ', ' + FirstName AS Name,
            EmailAddress AS email
       FROM Person.Contact
      WHERE LastName LIKE @LastName + '%')
GO
```

Now if I feed it the first part of a last name I'm interested in, I get relevant results:

```
SELECT *
FROM fnContactSearch('Ad')
```

This yields us approximately 87 rows (your numbers may vary depending on how much you've experimented with this data). Indeed, with the default data, we come up with 87 different variations of "Adams" for a last name—your parameterization has worked!

Chapter 11

Well, all this would probably be exciting enough, but sometimes we need more than just a single SELECT statement. Sometimes, we want more than just a parameterized view. Indeed, much as we saw with some of our scalar functions, we may need to execute multiple statements in order to achieve the results that we want. User-defined functions support this notion just fine. Indeed, they can return tables that are created using multiple statements—the only big difference when using multiple statements is that you must both name and define the metadata (much as you would a temporary table) for what you'll be returning.

For this example, we'll deal with a very common problem in the relational database world—hierarchies. Imagine for a moment that you are working in the Human Resources department. You have an Employee table, and it has a unary relationship that relates each employee to their boss through the ManagerID column—that is, the way you know who someone's boss is by relating the ManagerID column back to another EmployeeID. A very common need in a scenario like this is to be able to create a reporting “tree”—that is, a list of all of the people who exist below a given manager in an organization chart.

At first blush, this would seem pretty easy. If we wanted to know all the people who report to Terri Duffy, we might write a query that would join the Employee table back to itself and also join in the Contacts table to get names instead of just IDs—something like:

```
Use AdventureWorks

SELECT e.EmployeeID, ec.LastName, ec.FirstName, e.ManagerID
FROM HumanResources.Employee AS e
JOIN Person.Contact AS ec
    ON e.ContactID = ec.ContactID
JOIN HumanResources.Employee AS m
    ON m.EmployeeID = e.ManagerID
JOIN Person.Contact AS mc
    ON m.ContactID = mc.ContactID
WHERE mc.LastName = 'Duffy'
    AND mc.FirstName = 'Terri'
```

Again, at first glance, this might appear to give us what we want:

EmployeeID	LastName	FirstName	ManagerID
3	Tamburello	Roberto	12

(1 row(s) affected)

But, in reality, we have a bit of a problem here. At issue is that we want all of the people in Terri's reporting chain—not just those who report to Terri, but those who report to people who report to Terri, and so on. You see that if you look at all the records in AdventureWorks, you'll find a number of employees who report to Robert Tamburello, but they don't appear in the results of this query.

OK, so some of the quicker or more experienced among you may now be saying something like, “Hey, no problem! I'll just join back to the Employees table one more time!” You could probably make this work for such a small data set, or any situation where the number of levels of your hierarchy is fixed—but what if the number of hierarchy levels isn't fixed? What if people are reporting to Steven Buchanan, and still others report to people under Steven Buchanan—it could go on virtually forever. Now what? Glad you asked. . . .

Getting Procedural: Stored Procedures and User-Defined Functions

What we really need is a function that will return all the levels of the hierarchy below whatever EmployeeID (and, therefore, ManagerID) we provide. To do this, we have a classic example of *recursion*. A block of code is said to recurse any time it calls itself. We saw an example of this earlier in the chapter with our spTriangular stored procedure. Let's think about this scenario for a moment:

1. We need to figure out all the people who report to the manager that we want.
2. For each person in Step 1, we need to know who reports to them.
3. Repeat Step 2 until there are no more subordinates.

This is recursion all the way. What this means is that we're going to need several statements to make our function work: some statements to figure out the current level and at least one more to call the same function again to get the next lowest level.

Keep in mind that UDFs are going to have the same recursion limits that sprocs had — that is, you can only go to 32 levels of recursion, so, if you have a chance of running into this limit, you'll want to get creative in your code to avoid errors.

Let's put it together. Notice the couple of changes in the declaration of our function. This time, we need to associate a name with the return value (in this case, @Reports) — this is required any time you're using multiple statements to generate your result. Also, we have to define the table that we will be returning — this allows SQL Server to validate whatever we try to insert into that table before it is returned to the calling routine.

```
CREATE FUNCTION dbo.fnGetReports
    (@EmployeeID AS int)
RETURNS @Reports TABLE
(
    EmployeeID      int          NOT NULL,
    ManagerID      int          NULL
)
AS
BEGIN

    /* Since we'll need to call this function recursively - that is once for each
       reporting
    ** employee (to make sure that they don't have reports of their own), we need a
       holding
    ** variable to keep track of which employee we're currently working on. */
    DECLARE @Employee AS int

    /* This inserts the current employee into our working table. The significance here
       is
    ** that we need the first record as something of a primer due to the recursive
       nature
    ** of the function - this is how we get it. */
    INSERT INTO @Reports
        SELECT EmployeeID, ManagerID
        FROM HumanResources.Employee
        WHERE EmployeeID = @EmployeeID
    /* Now we also need a primer for the recursive calls we're getting ready to start
       making
```

Chapter 11

```
** to this function. This would probably be better done with a cursor, but we
haven't
** gotten to that chapter yet, so.... */
SELECT @Employee = MIN(EmployeeID)
FROM HumanResources.Employee
WHERE ManagerID = @EmployeeID

/* This next part would probably be better done with a cursor but we haven't gotten
to
** that chapter yet, so we'll fake it. Notice the recursive call to our function!
*/
WHILE @Employee IS NOT NULL
BEGIN
    INSERT INTO @Reports
        SELECT *
        FROM fnGetReports(@Employee)

    SELECT @Employee = MIN(EmployeeID)
    FROM HumanResources.Employee
    WHERE EmployeeID > @Employee
        AND ManagerID = @EmployeeID
END

RETURN

END
GO
```

I've written this one to provide just minimal information about the employee and their manager—I can join back to the `Employee` table if need be to fetch additional information. I also took a little bit of liberty with the requirements on this one, and added in the selected manager to the results. This was done primarily to support the recursion scenario and also to provide something of a base result for our result set. Speaking of which, let's look at our results—Terri is `EmployeeID` #12; to do this, we'll feed that into our function:

```
SELECT * FROM fnGetReports(12)
```

This gets us not only the original one person who reported to Terri Duffy but also those who report to Roberto Tamburello (who reports to Ms. Duffy) and Ms. Duffy herself (remember, I added her in as something of a starting point).

EmployeeID	ManagerID
12	109
3	12
4	3
9	3
11	3
158	3
79	158
114	158
217	158
263	3

Getting Procedural: Stored Procedures and User-Defined Functions

```
5          263
265        263
267        3
270        3
```

```
(14 row(s) affected)
```

Now, let's go the final step here and join this back to actual data. We'll use it much as we did our original query looking for the reports of Terri Duffy:

```
DECLARE @EmployeeID int

SELECT @EmployeeID = EmployeeID
FROM HumanResources.Employee e
JOIN Person.Contact c
    ON e.ContactID = c.ContactID
WHERE LastName = 'Duffy'
AND FirstName = 'Terri'

SELECT e.EmployeeID, ec.LastName, ec.FirstName, mc.LastName AS ReportsTo
FROM HumanResources.Employee AS e
JOIN dbo.fnGetReports(@EmployeeID) AS r
    ON e.EmployeeID = r.EmployeeID
JOIN HumanResources.Employee AS m
    ON m.EmployeeID = r.ManagerID
JOIN Person.Contact AS ec
    ON e.ContactID = ec.ContactID
JOIN Person.Contact AS mc
    ON m.ContactID = mc.ContactID
```

This gets us back all 14 employees who are under Ms. Duffy:

EmployeeID	LastName	FirstName	ReportsTo
12	Duffy	Terri	Sánchez
3	Tamburello	Roberto	Duffy
4	Walters	Rob	Tamburello
9	Erickson	Gail	Tamburello
11	Goldberg	Jossef	Tamburello
158	Miller	Dylan	Tamburello
79	Margheim	Diane	Miller
114	Matthew	Gigi	Miller
217	Raheem	Michael	Miller
263	Cracium	Ovidiu	Tamburello
5	D'Hers	Thierry	Cracium
265	Galvin	Janice	Cracium
267	Sullivan	Michael	Tamburello
270	Salavarria	Sharon	Tamburello

```
(14 row(s) affected)
```

So, as you can see, we can actually have very complex code build our table results for us, but it's still a table that results and, as such, it can be used just like any other table.

Understanding Determinism

Any coverage of UDFs would be incomplete without discussing determinism. If SQL Server is going to build an index over something, it has to be able to deterministically define (define with certainty) what the item being indexed is. Why does this matter to functions? Well, because we can have functions that feed data to things that will be indexed (computed column or indexed view).

User-defined functions can be either deterministic or non-deterministic. The determinism is not defined by any kind of parameter but rather by what the function is doing. If, given a specific set of valid inputs, the function will return exactly the same value every time, then the function is said to be deterministic. An example of a built-in function that is deterministic is `SUM()`. The sum of 3, 5, and 10 is always going to be 18—*every* time the function is called with those values as inputs. The value of `GETDATE()`, however, is non-deterministic—it changes pretty much every time you call it.

To be considered deterministic, a function has to meet four criteria:

- ❑ The function must be schema-bound. This means that any objects that the function depends on will have a dependency recorded and no changes to those objects will be allowed without first dropping the dependent function.
- ❑ All other functions referred to in your function, regardless of whether they are user or system defined, must also be deterministic.
- ❑ You cannot reference tables that are defined outside the function itself (use of table variables and temporary tables is fine, as long as the table variable or temporary table was defined inside the scope of the function).
- ❑ You cannot use an extended stored procedure inside the function.

The importance of determinism shows up if you want to build an index on a view or computed column. Indexes on views or computed columns are allowed only if the result of the view or computed column can be reliably determined. This means that, if the view or computed column refers to a non-deterministic function, no index will be allowed on that view or column. This situation isn't necessarily the end of the world, but you will want to think about whether a function is deterministic or not before creating indexes against views or columns that use that function.

So, this should beget the question: “How do I figure out whether my function is deterministic or not?” Well, beyond checking the rules we've already described, you can also have SQL Server tell you whether your function is deterministic or not—this information stored in the `IsDeterministic` property of the object. To check this out, you can make use of the `OBJECTPROPERTY` function. For example, we could check out the determinism of our `DayOnly` function that we used earlier in the chapter:

```
USE Accounting  
  
SELECT OBJECTPROPERTY(OBJECT_ID('DayOnly'), 'IsDeterministic')
```

It may come as a surprise to you (or maybe not) that the response is that this function is *not* deterministic:

```
-----  
0  
(1 row(s) affected)
```

Getting Procedural: Stored Procedures and User-Defined Functions

Look back through the list of requirements for a deterministic function and see if you can figure out why this one doesn't meet the grade.

When I was working on this example, I got one of those not so nice little reminders about how it's the little things that get you. You see, I was certain this function should be deterministic, and, of course, it wasn't. After too many nights writing until the morning hours, I completely missed the obvious—SCHEMABINDING.

Fortunately, we can fix the only problem this one has. All we need to do is add the WITH SCHEMABINDING option to our function, and we'll see better results:

```
ALTER FUNCTION DayOnly(@Date datetime)
RETURNS varchar(12)
WITH SCHEMABINDING
AS
BEGIN
    RETURN CONVERT(varchar(12), @Date, 101)
END
```

Now, we just re-run our OBJECTPROPERTY query:

```
-----
1
(1 row(s) affected)
```

And voilà—a deterministic function!

Debugging

Real-live debugging tools first made an appearance in SQL Server in SQL Server 2000. Much as with SQL Server 2005, you needed your settings to be just perfect and several starts to align to get it working. After that, it was wonderful.

The debugging effort for SQL Server 2005 is highly integrated with Visual Studio and does work fairly well.

I'm not going to kid you: The debugging tools are still a pain at best (and impossible at worst) to get functional. Given the focus on security in recent years, so many parts of your server are locked down to external calls now that remote debugging (which is what you're going to want if you have more than one developer on the project) is particularly difficult to get going. All I can say is, hang with it and keep trying. It's worth the effort when you get it working.

Setting Up SQL Server for Debugging

SQL Server no longer ships with a debugger as part of the product—you must have Visual Studio in order to debug even your T-SQL-based stored procedures. As of this writing, the Express edition of Visual Studio is free for one year.

Depending on the nature of your installation, you may have to do absolutely nothing to get debugging working. If, however, you took the default path and installed your SQL Server to run using the LocalSystem account, debugging might not work at all. The upshot of this is that, if you want to use debugging, you need to configure the SQL Server service to run using an actual user account, specifically, one with admin access to the box on which SQL Server is running.

Having SQL Server run using an account with admin access is definitely something that most security experts would gag, cough, and choke at. It's a major security loophole. Why? Well, there are things that would wind up running with admin access also. Imagine any user who could create assemblies on your system also being able to delete any file on your box, move things around, or possibly worse. This is a development system only kind of thing. Also, make sure that you're using a local admin account rather than a domain admin.

Starting the Debugger

Much of using the Debugger works as it does in VB or C++—probably like most modern debuggers for that matter.

Before I get too far into this, I'm going to warn you that, while the Debugger is great in many respects, it leaves something to be desired in terms of how easy it is to find it. It's not even built into the query tool anymore, so pay attention to the steps you have to walk through to find it.

To get the Debugger going, you need to start up Visual Studio and go to the Server Explorer (located under the View menu). Right-click Data Connection and select Add Connection (if you don't have one already). Add in the connection information in the Add Connection dialog box, as shown in Figure 11-2.

Navigate to the sproc (or UDF) that you want to debug and right-click. In this case, navigate to the spTriangular stored procedure that you created earlier in the chapter, and right-click it; then choose Step Into Stored Procedure, as shown in Figure 11-3.

This opens the Run Stored Procedure dialog box, which prompts you to fill in the required information for the parameters for your stored procedure has. (See Figure 11-4.)

You need to set each parameter's value before the sproc can run. Set @ValueIn, to 3. For the @ValueOut, use the NULL option.

Getting Procedural: Stored Procedures and User-Defined Functions

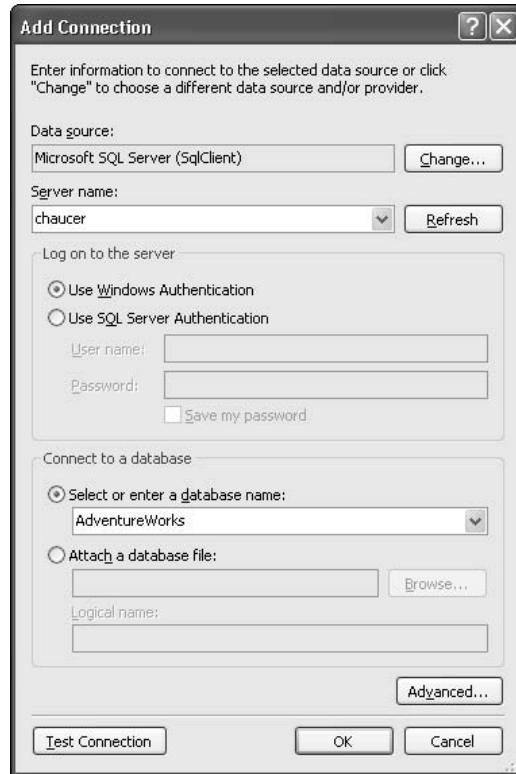


Figure 11-2

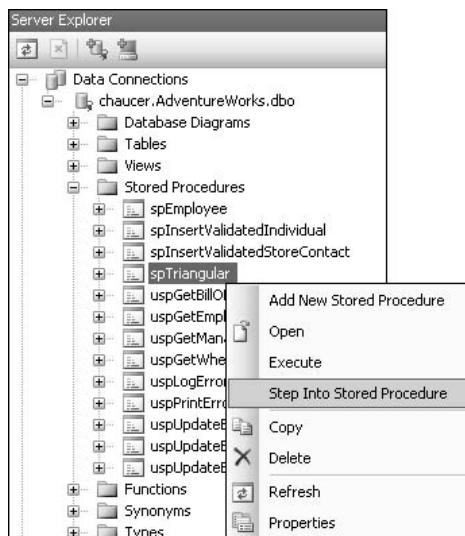


Figure 11-3

Chapter 11

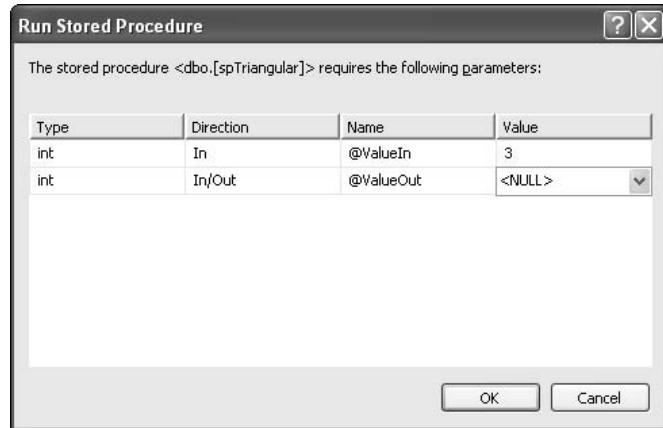


Figure 11-4

Then click OK. This steps you into your sproc, as shown in Figure 11-5:

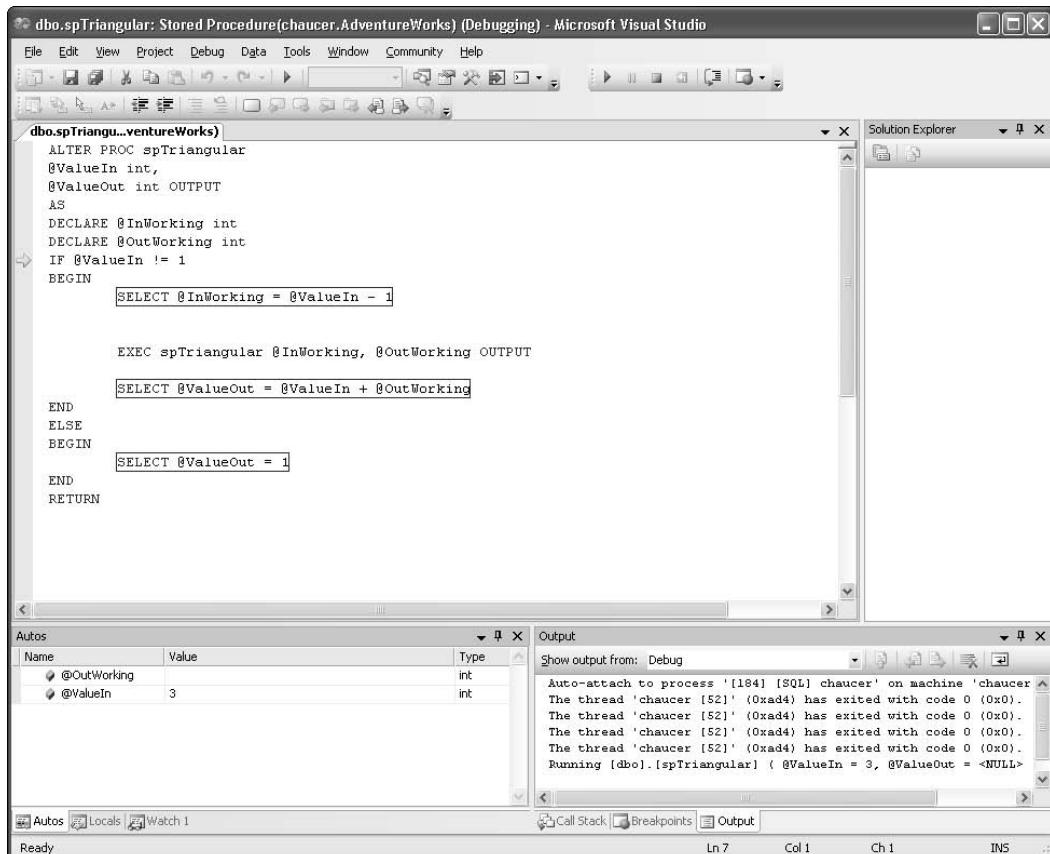


Figure 11-5

Parts of the Debugger

Several things are worth noticing when the Debugger window first opens:

- ❑ The yellow arrow on the left indicates the current row. This is the next line of code that will be executed if you do a “go” or start stepping through the code.
- ❑ The icons at the top to indicate the available options:
 - ❑ **Continue**—This option runs to the end of the sproc. After you click this, the only thing that will stop execution is a runtime error or hitting a breakpoint.
 - ❑ **Step Into**—This executes the next line of code and stops prior to running the next line of code regardless of what procedure or function that code is in. If the line of code being executed is calling a sproc or function, then **Step Into** has the effect of calling that sproc or function, adding it to the call stack, changing the locals window to represent the newly nested sproc rather than the parent, and then stopping at the first line of code in the nested sproc.
 - ❑ **Step Over**—This executes every line of code required to take you to the next statement that is at the same level in the call stack. If you aren’t calling another sproc or a UDF, this command acts just like **Step Into**. If, however, you are calling another sproc or a UDF, then a **Step Over** takes you to the statement immediately following where that sproc or UDF returned its value.
 - ❑ **Step Out**—This executes every line of code up to the next line of code at the next highest point in the call stack. That is, you will keep running until you reach the same level as the code that called your current level.
 - ❑ **Run To Cursor**—This works pretty much like the combination of a breakpoint and a Go. When this choice is made, the code starts running and keeps going until it gets to the current cursor location. The only exceptions are if there is a breakpoint prior to the cursor location (then it stops at the breakpoint instead) or the end of the sproc comes before the cursor line is executed, such as when you place the cursor on a line that has already occurred or is in part of a control-of-flow statement that doesn’t get executed.
 - ❑ **Restart**—This does exactly what it says it does. It sets the parameters to their original values, clears any variables and the call stack, and starts over.
 - ❑ **Stop Debugging**—Again, this does what it says; it stops execution immediately. The debugging window does remain open, however.
 - ❑ **Toggle Breakpoints and Remove All Breakpoints**—In addition, you can set breakpoints by clicking in the left margin of the code window. Breakpoints are points that you set to tell SQL Server to stop there when the code is running in debug mode. This is handy in big sprocs or functions where you don’t want to have to deal with every line; you just want it to run up to a point and stop every time it gets there.

There are also a few “status” windows.

The Locals Window

As I indicated back at the beginning of the book, I’m pretty much assuming that you have experience with some procedural language out there. As such, the Locals window probably isn’t all that new of a concept to you. The simple rendition is that it shows you the current value of all the variables that are in

scope. The list of variables in the Locals window may change (as may their values) as you step into nested sprocs and back out again. Remember, these are only those variables that are in scope as of the next statement to run.

Three pieces of information are provided for each variable or parameter:

- The name
- The current value
- The data type

However, perhaps the best part to the Locals window is that you can edit the values in each variable. That means it's a lot easier to change things on the fly to test certain behaviors in your sproc.

The Watch Window

This is a bit weaker than most modern debuggers but does allow you to set a variable to watch for and see changes.

The Callstack Window

The Callstack window provides a listing of all the sprocs and functions that are currently active in the process that you are running. The handy thing here is that you can see how far in you are when you are running in a nested situation, and you can change between the nesting levels to verify what current variable values are at each level.

The Output Window

The Output window is the spot where SQL Server prints any output. This includes result sets as well as the return value when your sproc has completed running.

Using the Debugger after It's Started

With the preliminaries out of the way and the Debugger window open, you're ready to start walking through your code.

The first executable line of your sproc is the `IF` statement, so that's the line that is current when the Debugger starts. None of your variables has had any values set in it yet except for the `@ValueIn` that you passed in as a parameter to the sproc. It has the value of 3 that you passed in when you filled out the Debug Procedure dialog box earlier.

Step forward one line by pressing F11 or using the Step Into icon or menu choice.

Because the value of `@ValueIn` is indeed not equal to 1, you step into the `BEGIN...END` block specified by the `IF` statement. Specifically, you move to the `SELECT` statement that initializes the `@InWorking` parameter. As you'll see later, if the value of `@ValueIn` had indeed been 1, you would have immediately dropped down to the `ELSE` statement.

Again, step forward one line by pressing F11 or using the Step Into icon or menu choice, as shown in Figure 11-6.

Getting Procedural: Stored Procedures and User-Defined Functions

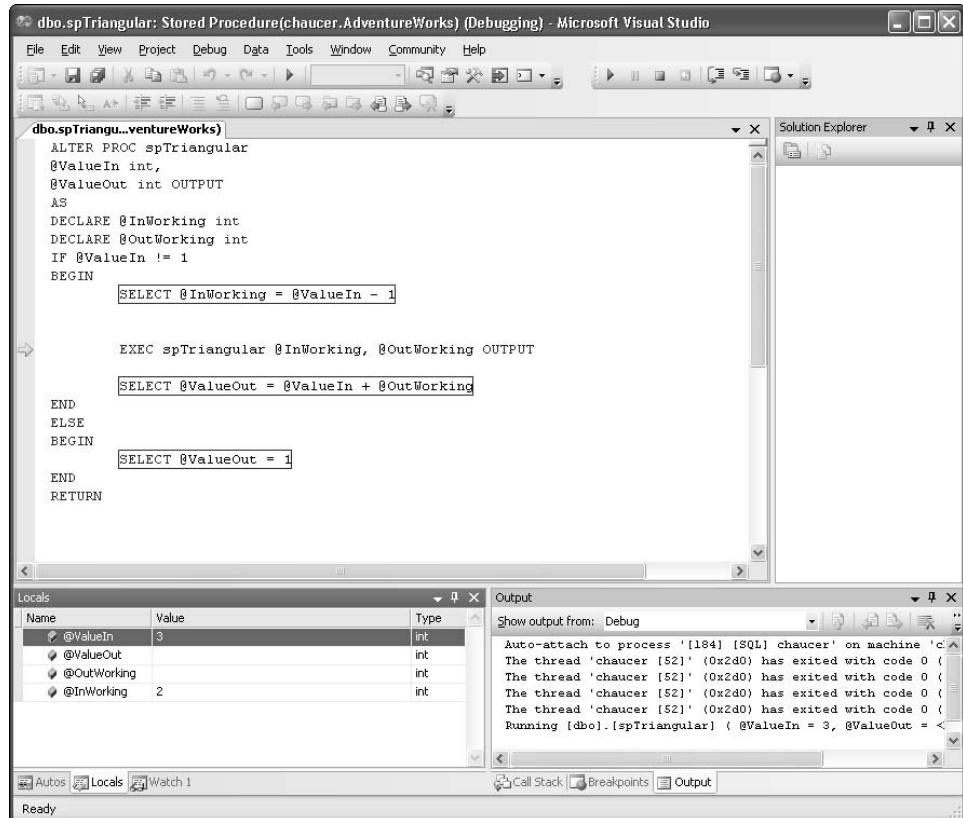


Figure 11-6

Pay particular attention to the value of `@InWorking` in the Locals window. Notice that it changed to the correct value (`@ValueIn` is currently 3, so $3 - 1$ is 2) as set by your `SELECT` statement. Also notice that the Callstack window has only the current instance of the sproc in it. Because haven't stepped down into your nested versions of the sproc yet, you see only one instance.

Now go ahead and step into the next statement. Because this is the execution of a sproc, you're going to see a number of different things change in the Debugger window, as shown in Figure 11-7:

Notice that it *appears* that the arrow that indicates the current statement jumped back up to the `IF` statement. Why? Well, this is a new instance of your sproc. You can tell this based on the Callstack window. Notice that it now has two instances of your sproc listed. The blue one at the top is the current instance. Notice also that the `@ValueIn` parameter has the value of 2. That is the value you passed in from the outer instance of the sproc.

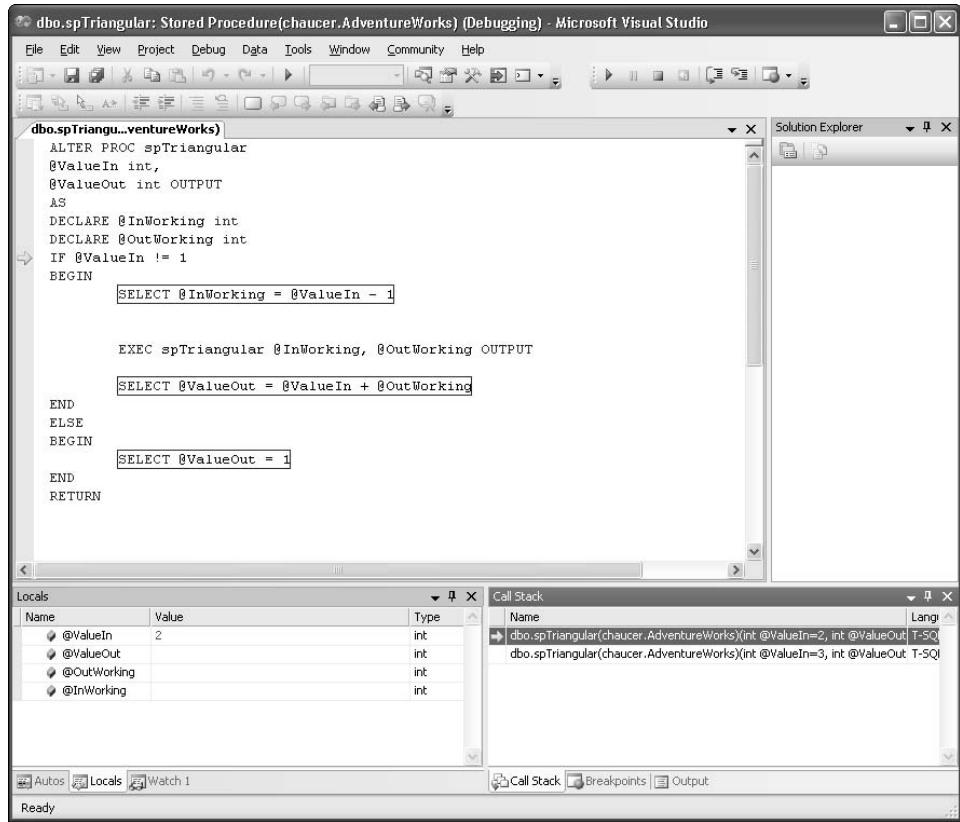


Figure 11-7

If you want to see the value of variables in the scope of the outer instance of the sproc, just double-click that instance's line in the Callstack window (the one with the green arrow) and you'll see several things change, as shown in Figure 11-8.

There are two things to notice here. First, the values of your variables have changed to those in the scope of the outer (and currently selected) instance of the sproc. Second, the icon for the current execution line is different. This new green arrow is meant to show that this is the current line in this instance of the sproc, but it isn't the current line in the overall callstack.

Getting Procedural: Stored Procedures and User-Defined Functions

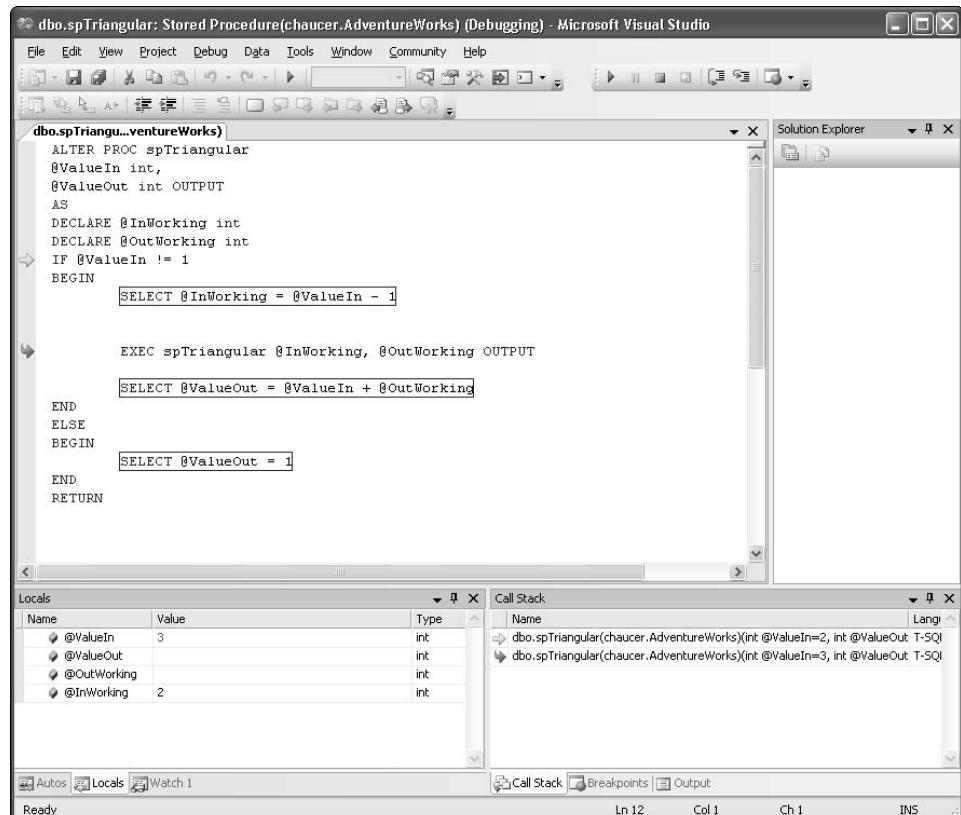


Figure 11-8

Go back to the current instance by clicking the top item in the Callstack window. Then step in three more times. This should bring you to the top line (the IF statement) in the third instance of the sproc. Notice that your callstack has become three deep and that the values of your variables and parameters in the Locals window have changed again. Last, but not least, notice that this time your @ValueIn parameter has a value of 1.

Step into the code a couple more times, and you'll see a slight change in behavior when you get to a @ValueIn of 1. This time, because the value in @ValueIn is indeed equal to 1, you move into the BEGIN...END block defined with the ELSE statement, as shown in Figure 11-9.

Chapter 11

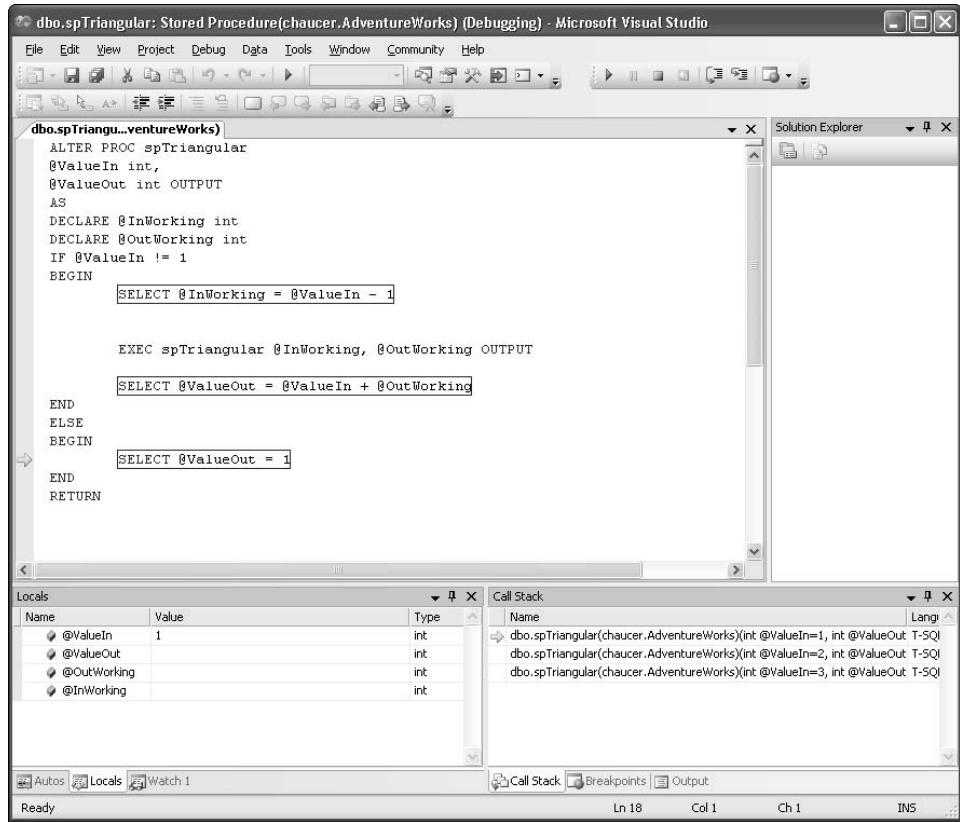


Figure 11-9

Since you reached the bottom, you're ready to start going back up the callstack. Two more steps take you back to the second level in the stack, as shown in Figure 11-10.

Notice that our callstack is back to only one level. Also, notice that our output parameter (@OutWorking) has been appropriately set.

This time, do a Step Out (Shift+F11), and you're done.

So, you should now be able to see how the Debugger can be very handy indeed.

Getting Procedural: Stored Procedures and User-Defined Functions

The screenshot shows the Microsoft Visual Studio interface with the following components:

- Title Bar:** Shows the project name "dbo.spTriangular: Stored Procedure(chaucer.AdventureWorks) (Debugging) - Microsoft Visual Studio".
- Toolbar:** Standard Visual Studio toolbar with various icons for file operations, search, and navigation.
- Code Editor:** Displays the T-SQL code for the stored procedure "spTriangular". The code includes logic to calculate triangular numbers based on the input parameter @ValueIn. It uses temporary variables @InWorking and @OutWorking, and returns the result through @ValueOut.
- Solution Explorer:** A small window on the right side of the interface.
- Locals Window:** Shows the current values of local variables:

Name	Value	Type
@ValueIn	3	int
@ValueOut		int
@OutWorking	3	int
@InWorking	2	int
- Call Stack Window:** Shows the call stack entry: "dbo.spTriangular(chaucer.AdventureWorks)(int @ValueIn=3, int @ValueOut T-SQL)".
- Bottom Status Bar:** Displays "Ln 14 Col 1 Ch 1 INS".

Figure 11-10

Summary

Stored procedures and user-defined functions form the core elements of SQL Server programmability. In this chapter, we've look extensively at what they have to offer in terms of core T-SQL programmability. Over the next two chapters, we will add in transactions and then follow that with the last of the major core programming elements of SQL Server — triggers. From there, we're ready to move on to .NET and what it has to offer in the way of extending our SQL Server world.

12

Transactions and Locks

Ahhh . . . the fundamentals. In this case, a fundamental that even lots of fairly advanced users don't quite "get it" on. I've said it in every version of this book I've done, and even in my *Beginning SQL Server 2005 Programming* title: nothing in this chapter is wildly difficult, yet transactions and locks tend to be two of the most misunderstood areas in the database world.

This is one of those chapters that, when you go back to work, make you sound like you've had your Wheaties today. Nothing in what we're going to cover in this chapter is wildly difficult, yet transactions and locks tend to be two of the most misunderstood areas in the database world. As such, this "beginning" (or at least I think it's a basic) concept is going to make you start to look like a real pro.

In this chapter, we're going to:

- Examine transactions
- Examine how the SQL Server log and "checkpoints" work
- Unlock your understanding of locks

We'll learn why these topics are so closely tied to each other, and how to minimize problems with each.

Transactions

Transactions are all about *atomicity*. Atomicity is the concept that something should act as a unit. From our database standpoint, it's about the smallest grouping of one or more statements that should be considered to be "all or nothing."

Chapter 12

Often, when dealing with data, we want to make sure that if one thing happens, another thing happens, or that neither of them do. Indeed, this can be carried out to the degree where 20 things (or more) all have to happen together or nothing happens. Let's look at a classic example.

Imagine that you are a banker. Sally comes in and wants to transfer \$1,000 from checking to savings. You are, of course, happy to oblige, so you process her request.

Behind the scenes, we have something like this happening:

```
UPDATE checking
    SET Balance = Balance - 1000
    WHERE Account = 'Sally'
UPDATE savings
    SET Balance = Balance + 1000
    WHERE Account = 'Sally'
```

This is a hypersimplification of what's going on, but it captures the main thrust of things: you need to issue two different statements—one for each account.

Now, what if the first statement executes and the second one doesn't? Sally would be out of a thousand dollars! That might, for a short time, seem okay from your perspective (heck, you just made a thousand bucks!), but not for long. By that afternoon you'd have a steady stream of customers leaving your bank—it's hard to stay in the bank business with no depositors.

What you need is a way to be certain that if the first statement executes, the second statement executes. There really isn't a way that we can be certain of that—all sorts of things can go wrong, from hardware failures to simple things such as violations of data integrity rules. Fortunately, however, there is a way to do something that serves the same overall purpose—we can essentially forget that the first statement ever happened. We can enforce at least the notion that if one thing didn't happen, then nothing did—at least within the scope of our *transaction*.

In order to capture this notion of a transaction, however, we need to be able to define very definite boundaries. A transaction has to have very definitive begin and end points. Actually, every `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statement you issue in SQL Server is part of an implicit transaction. Even if you issue only one statement, that one statement is considered to be a transaction—everything about the statement will be executed, or none of it will. Indeed, by default, that is the length of a transaction—one statement.

But what if we need to have more than one statement be all or nothing—such as our preceding bank example? In such a case, we need a way of marking the beginning and end of a transaction, as well as the success or failure of that transaction. To that end, there are several T-SQL statements that we can use to “mark” these points in a transaction. We can:

- BEGIN a transaction**—Set the starting point.
- COMMIT a transaction**—Make the transaction a permanent, irreversible part of the database.
- ROLLBACK a transaction**—Essentially saying that you want to forget that it ever happened.
- SAVE a transaction**—Establish a specific marker to allow us to do only a partial rollback.

Let's look over all of these individually before we put them together into our first transaction.

BEGIN TRAN

The beginning of the transaction is probably one of the easiest concepts to understand in the transaction process. Its sole purpose in life is to denote the point that is the beginning of a unit. If, for some reason, we are unable to or do not want to commit the transaction, this is the point to which all database activity will be rolled back. That is, everything beyond this point that is not eventually committed will effectively be forgotten as far as the database is concerned.

The syntax is:

```
BEGIN TRAN[SACTION] [<transaction name>|<@transaction variable>]
```

COMMIT TRAN

The committing of a transaction is the end of a completed transaction. At the point that you issue the COMMIT TRAN, the transaction is considered to be what is called *durable*. That is, the effect of the transaction is now permanent and will last even if you have a system failure (as long as you have a backup or the database files haven't been physically destroyed). The only way to "undo" whatever the transaction accomplished is to issue a new transaction that, functionally speaking, is a reverse of your first transaction.

The syntax for a COMMIT looks pretty similar to a BEGIN:

```
COMMIT TRAN[SACTION] [<transaction name>|<@transaction variable>]
```

ROLLBACK TRAN

Whenever I think of a ROLLBACK, I think of the old movie *The Princess Bride*. If you've ever seen the film (if you haven't, I highly recommend it), you'll know that the character Vizzini (considered a genius in the film) always said, "If anything goes wrong — go back to the beginning."

That was some mighty good advice. A ROLLBACK does just what Vizzini suggested — it goes back to the beginning. In this case, it's your transaction that goes back to the beginning. Anything that happened since the associated BEGIN statement is effectively forgotten about. The only exception to going back to the beginning is through the use of what are called *savepoints* — which I'll describe shortly.

The syntax for a ROLLBACK again looks pretty much the same, with the exception of allowance for a savepoint.

```
ROLLBACK TRAN[SACTION] [<transaction name>|<save point name>|<@transaction variable>|<@savepoint variable>]
```

SAVE TRAN

To save a transaction is essentially to create something of a bookmark. You establish a name for your bookmark (you can have more than one). After this "bookmark" is established, you can reference it in a rollback. What's nice about this is that you can roll back to the exact spot in the code that you want to — just by naming a savepoint to which you want to roll back.

Chapter 12

The syntax is simple enough:

```
SAVE TRAN[SACTION] [<save point name>| <@savepoint variable>]
```

The thing to remember about savepoints is that they are cleared on ROLLBACK—that is, even if you save five savepoints, once you perform one ROLLBACK they are all gone. You can start setting new savepoints again, and rolling back to those, but whatever savepoints you had when the ROLLBACK was issued are gone.

Savepoints were something of a major confusion area for me when I first came across them. Books Online indicates that, after rolling back to a savepoint, you must run the transaction to a logical conclusion (this is technically correct). Where the confusion came was an implication in the way that Books Online was written that seemed to indicate that you had to go to a ROLLBACK or COMMIT without using any more savepoints. This is not the case—you just can't use the savepoints that we declared prior to the ROLLBACK—savepoints after this are just fine.

Let's test this out with a bit of code to see what happens when we mix the different types of TRAN commands. Type the following code in and then we'll run through an explanation of it:

```
USE AdventureWorks -- Since we're making our own table, what DB doesn't matter

-- Create table to work with
CREATE TABLE MyTranTest
(
    OrderID      INT      PRIMARY KEY      IDENTITY
)
-- Start the transaction
BEGIN TRAN TranStart

-- Insert our first piece of data using default values.
-- Consider this record No1. It is also the 1st record that stays
-- after all the rollbacks are done.
INSERT INTO MyTranTest
    DEFAULT VALUES

-- Create a "Bookmark" to come back to later if need be
SAVE TRAN FirstPoint

-- Insert some more default data (this one will disappear
-- after the rollback).
-- Consider this record No2.
INSERT INTO MyTranTest
    DEFAULT VALUES

-- Roll back to the first savepoint. Anything up to that
-- point will still be part of the transaction. Anything
-- beyond is now toast.
ROLLBACK TRAN FirstPoint

-- Insert some more default data.
-- Consider this record No3 It is the 2nd record that stays
```

```
-- after all the rollbacks are done.

INSERT INTO MyTranTest
    DEFAULT VALUES

-- Create another point to roll back to.
SAVE TRAN SecondPoint

-- Yet more data. This one will also disappear,
-- only after the second rollback this time.
-- Consider this record No4.
INSERT INTO MyTranTest
    DEFAULT VALUES

-- Go back to second savepoint
ROLLBACK TRAN SecondPoint

-- Insert a little more data to show that things
-- are still happening.
-- Consider this record No5. It is the 3rd record that stays
-- after all the rollbacks are done.
INSERT INTO MyTranTest
    DEFAULT VALUES

-- Commit the transaction
COMMIT TRAN TranStart

-- See what records were finally committed.
SELECT TOP 3 OrderID
FROM MyTranTest
ORDER BY OrderID DESC

-- Clean up after ourselves
DROP TABLE MyTranTest
```

First, we create a table to work with for our test:

```
-- Create table to work with
CREATE TABLE MyTranTest
(
    OrderID      INT      PRIMARY KEY      IDENTITY
)
```

Since we're creating our own table to play with, what database we are using doesn't really matter for this demonstration.

Then it's time to begin the transaction. This starts our grouping of "all or nothing" statements. We then **INSERT** a row. At this juncture, we have just one row inserted:

```
-- Start the transaction
BEGIN TRAN TranStart

-- Insert our first piece of data using default values.
-- Consider this record No1. It is also the 1st record that stays
```

Chapter 12

```
-- after all the rollbacks are done.  
INSERT INTO MyTranTest  
    DEFAULT VALUES
```

Next, we establish a savepoint called `FirstPoint` and insert yet another row. At this point, we have two rows inserted, but remember, they are not committed yet, so the database doesn't consider them to really be part of the database:

```
-- Create a "Bookmark" to come back to later if need be  
SAVE TRAN FirstPoint  
  
-- Insert some more default data (this one will disappear  
-- after the rollback).  
-- Consider this record No2.  
INSERT INTO MyTranTest  
    DEFAULT VALUES
```

We then `ROLLBACK`—explicitly saying that it is *not* the beginning that we want to rollback to, but just to `FirstPoint`. With the `ROLLBACK`, everything between we `ROLLBACK` and the `FirstPoint` savepoint is undone. Since we have one `INSERT` statement between the `ROLLBACK` and the `SAVE`, that statement is rolled back. At this juncture, we are back down to just one row inserted. Any attempt to reference a savepoint would now fail since all savepoints have been reset with our `ROLLBACK`:

```
-- Roll back to the first savepoint. Anything up to that  
-- point will still be part of the transaction. Anything  
-- beyond is now toast.  
ROLLBACK TRAN FirstPoint
```

We add another row, putting us back up to a total of two rows inserted at this point. We also create a brand new savepoint. This is perfectly valid, and we can now refer to this savepoint since it is established after the `ROLLBACK`:

```
-- Insert some more default data.  
-- Consider this record No3 It is the 2nd record that stays  
-- after all the rollbacks are done.  
  
INSERT INTO MyTranTest  
    DEFAULT VALUES  
  
-- Create another point to roll back to.  
SAVE TRAN SecondPoint
```

Time for yet another row to be inserted, bringing our total number of still-valid inserts up to three:

```
-- Yet more data. This one will also disappear,  
-- only after the second rollback this time.  
-- Consider this record No4.  
INSERT INTO MyTranTest  
    DEFAULT VALUES
```

Now we perform another `ROLLBACK`—this time referencing our new savepoint (which happens to be the only one valid at this point since `FirstPoint` was reset after the first `ROLLBACK`). This one undoes

everything between it and the savepoint it refers to—in this case just one `INSERT` statement. That puts us back at two `INSERT` statements that are still valid:

```
-- Go back to second savepoint  
ROLLBACK TRAN SecondPoint
```

We then issue yet another `INSERT` statement, bringing our total number of `INSERT` statements that are still part of the transaction back up to three:

```
-- Insert a little more data to show that things  
-- are still happening.  
-- Consider this record No5. It is the 3rd record that stays  
-- after all the rollbacks are done.  
INSERT INTO MyTranTest  
    DEFAULT VALUES
```

Last (for our transaction anyway), but certainly not least, we issue the `COMMIT TRAN` statement that locks our transaction in and makes it a permanent part of the history of the database:

```
-- Commit the transaction  
COMMIT TRAN TranStart  
  
-- See what records were finally committed.  
SELECT TOP 3 OrderID  
FROM MyTranTest  
ORDER BY OrderID DESC
```

Note that if either of these `ROLLBACK` statements had not included the name of a savepoint, or had included a name that had been set with the `BEGIN` statement, then the entire transaction would have been rolled back, and the transaction would be considered to be closed.

With the transaction complete, we can issue a little statement that shows us our three rows. When you look at this, you'll be able to see what's happened in terms of rows being added and then removed from the transaction:

```
OrderID  
-----  
5  
3  
1  
  
(3 row(s) affected)
```

Sure enough, every other row was inserted.

Finally, we clean up after ourselves — this really has nothing to do with the transaction.

```
DROP TABLE MyTranTest
```

How the SQL Server Log Works

You definitely must have the concept of transactions down before you get into trying to figure out the way that SQL Server tracks what's what in your database. You see, what you *think* of as your database is only rarely a complete version of all the data. Except for rare moments when it happens that everything has been written to disk, the data in your database is made up of not only the data in the physical database file(s) but also any transactions that have been committed to the log since the last checkpoint.

In the normal operation of your database, most activities that you perform are “logged” to the *transaction log* rather than written directly to the database. A *checkpoint* is a periodic operation that forces all dirty pages for the database currently in use to be written to disk. Dirty pages are log or data pages that have been modified after they were read into the cache, but the modifications have not yet been written to disk. Without a checkpoint the log would fill up and/or use all the available disk space. The process works something like the diagram in Figure 12-1.

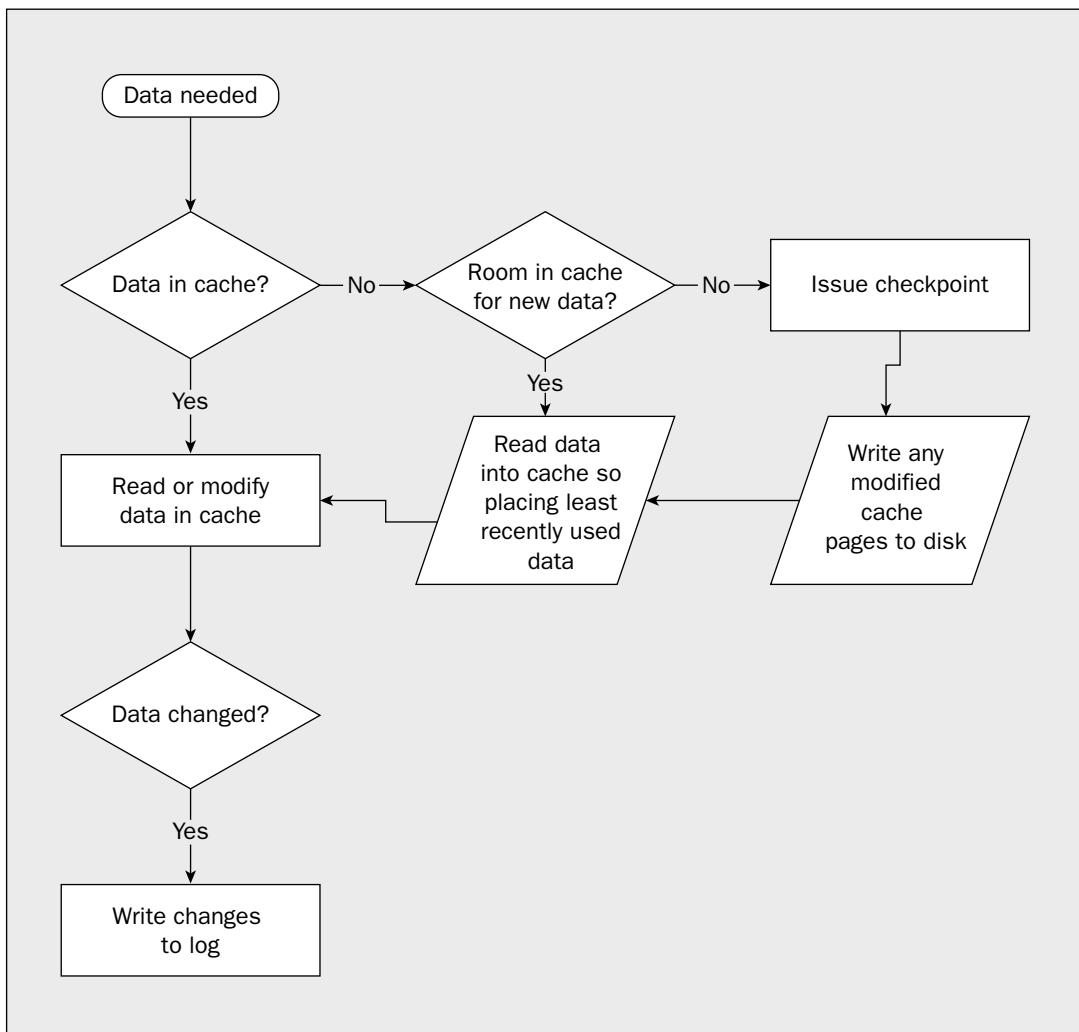


Figure 12-1

Don't mistake all this as meaning that you have to do something special to get your data out of the cache. SQL Server handles all of this for you. This information is only provided here to facilitate your understanding of how the log works, and, from there, the steps required to handle a transaction.

Whether something is in cache or not can make a big difference to performance, so understanding when things are logged and when things go in and out of the cache can be a big deal when you are seeking maximum performance.

Note that the need to read data into a cache that is already full is not the only reason that a checkpoint would be issued. Checkpoints can be issued under the following circumstances:

- By a manual statement—using the `CHECKPOINT` command.
- At normal shutdown of the server (unless the `WITH NOWAIT` option is used).
- When you change any database option (for example, single user only, dbo only, and so on).
- When the Simple Recovery option is used and the log becomes 70 percent full.
- When the amount of data in the log since the last checkpoint (often called the *active* portion of the log) exceeds the size that the server could recover in the amount of time specified in the `recovery interval` option.

Let's look at each of these more carefully.

Using the `CHECKPOINT` Command

One way—but probably the least often used way—for the database to have a checkpoint issued is for it to be done manually. You can do this anytime by just typing in the word:

`CHECKPOINT`

It's just that simple.

SQL Server does a very good job of managing itself in the area of checkpoints, so the times when issuing a manual checkpoint makes sense are fairly rare.

One place that I will do this is during the development cycle when I have the Truncate On Checkpoint option turned on for my database (you are very unlikely to want that option active on a production database). It's not at all uncommon during the development stage of your database to perform actions that are long running and fill up the log rather quickly. While I could always just issue the appropriate command to truncate the log myself, `CHECKPOINT` is a little shorter and faster and, when Truncate On Checkpoint is active, has the same effect.

`CHECKPOINT` on Recovery

Every time you start your server, it goes through period called *recovery* (covered a bit later in the chapter). Turning on the `Checkpoint on Recovery` option means just what it says: a checkpoint will be issued every time your database does a recovery—which is going to be anytime your server starts up.

Generally speaking, I recommend against the use of this option. Since a checkpoint is automatically issued at normal system shutdown, there really shouldn't be anything that needs to be the subject of a checkpoint unless your shutdown was caused by a system failure of some kind. Having this option on means that your server still needs to go through the process though, and that will slow down your startup times—not much since there usually won't be any data to commit to the main database file, but it still slows things down.

At Normal Server Shutdown

Ever wonder why SQL Server can sometimes take a very long time to shut down? Besides the deallocation of memory and other destructor routines that have to run to unload the system, SQL Server must also first issue a checkpoint before the shutdown process can begin. This means that you'll have to wait for any data that's been committed in the log to be written out to the physical database before your shutdown can continue. Checkpoints also occur when the server is stopped:

- Using the Management Studio
- Using the net stop mssqlserver NT command on the command prompt
- Using the services icon in the Windows control panel, selecting the mssqlserver service, and clicking the stop button

Unlike Checkpoint on Recovery, this is something that I like. I like the fact that all my committed transactions are in the physical database (not split between the log and database), which just strikes me as being cleaner, with less chance of data corruption.

There is a way you can get around the delay if you so choose. To use it, you must be shutting down using the SHUTDOWN command in T-SQL. To eliminate the delay associated with the checkpoint (and the checkpoint itself for that matter), you just add the WITH NO WAIT key phrase to your shutdown statement:

```
SHUTDOWN [WITH NO WAIT]
```

Note that I recommend highly *against* using this unless you have some programmatic need to shut down your server. It will cause the subsequent restart to take a longer time than usual to recover the databases on the server.

At a Change of Database Options

A checkpoint is issued anytime you issue a change to your database options regardless of how the option gets changed (such as using sp_dboption or ALTER DATABASE). The checkpoint is issued prior to making the actual change in the database.

When the Truncate on Checkpoint Option Is Active

If you have turned on the Truncate On Checkpoint database option (which is a common practice during the development phase of your database), then SQL Server will automatically issue a checkpoint any time the log becomes more than 70 percent full.

When Recovery Time Would Exceed the Recovery Interval Option Setting

As we saw briefly earlier (and will see more closely next), SQL Server performs a process called recovery every time the SQL Server is started up. SQL Server will automatically issue a checkpoint any time the estimated time to run the recovery process would exceed the amount of time set in a database option called recovery interval. By default, the recovery interval is set to zero—which means that SQL Server will decide for you (in practice, this means about 1 minute).

Failure and Recovery

A recovery happens every time that SQL Server starts up. SQL Server takes the database file and then applies (by writing them out to the physical database file) any committed changes that are in the log since the last checkpoint. Any changes in the log that do not have a corresponding commit are rolled back—that is, they are essentially forgotten about.

Let's take a look at how this works depending on how transactions have occurred in your database. Imagine five transactions that span the log, as pictured in Figure 12-2

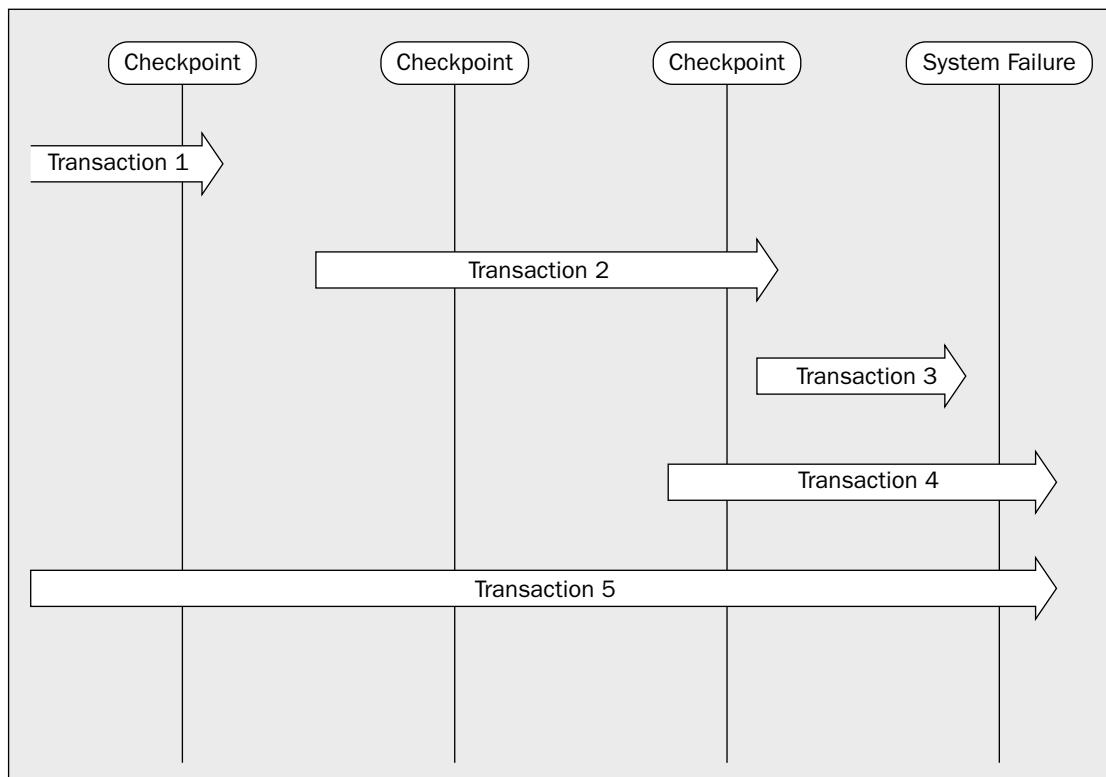


Figure 12-2

Chapter 12

Let's look at what would happen to these transactions one by one.

Transaction 1

Absolutely nothing would happen. The transaction has already been through a checkpoint and has been fully committed to the database. There is no need to do anything at recovery, because any data that is read into the data cache would already reflect the committed transaction.

Transaction 2

Even though the transaction existed at the time that a checkpoint was issued, the transaction had not been committed (the transaction was still going). Without that commitment, the transaction does not actually participate in the checkpoint. This transaction would, therefore, be "rolled forward." This is just a fancy way of saying that we would need to read all the related pages back into cache and then use the information in the log to re-run all the statements that we ran in this transaction. When that's finished, the transaction should look exactly as it did before the system failed.

Transaction 3

It may not look the part, but this transaction is exactly the same as Transaction 2 from the standpoint of what needs to be done. Again, because Transaction 3 wasn't finished at the time of the last checkpoint, it did not participate in that checkpoint just like Transaction 2 didn't. The only difference is that Transaction 3 didn't even exist at that time, but, from a recovery standpoint, that makes no difference—it's where the commit is issued that makes all the difference.

Transaction 4

This transaction wasn't completed at the time of system failure, and must, therefore, be rolled back. In effect, it never happened from a row data perspective. The user would have to re-enter any data, and any process would need to start from the beginning.

Transaction 5

This one is no different than Transaction 4. It appears to be different because the transaction has been running longer, but that makes no difference. The transaction was not committed at the time of system failure, and must therefore be rolled back.

Implicit Transactions

Primarily for compatibility with other major RDBMS systems such as Oracle or DB2, SQL Server supports (it is off by default but can be turned on if you choose) the notion of what is called an *implicit transaction*. Implicit transactions do not require a BEGIN TRAN statement—instead, they are automatically started with your first statement. They then continue until you issue a COMMIT TRAN or ROLLBACK TRAN statement. The next transaction then begins with your next statement.

Theoretically, the purpose behind this is to make sure that every statement is part of a transaction. SQL Server also wants every statement to be part of a transaction, but, by default, takes a different approach—if there is no BEGIN TRAN, then SQL Server assumes you have a transaction of just one statement, and automatically begins and ends that transaction for you. With some other systems though, you'll find the implied transaction approach. Those systems will assume that any one statement is only the beginning

of the transaction, and therefore require that you explicitly end the every transaction with a `COMMIT` or `ROLLBACK`.

By default, the `IMPLICIT_TRANSACTIONS` option is turned off (and the connection is in autocommit transaction mode). You can turn it on by issuing the command:

```
SET IMPLICIT_TRANSACTIONS ON
```

After that, any of the following statements will initiate a transaction:

```
CREATE  
ALTER TABLE  
GRANT  
REVOKE  
SELECT  
UPDATE  
DELETE  
INSERT  
TRUNCATE TABLE  
DROP  
OPEN  
FETCH
```

The transaction will continue until you `COMMIT` or `ROLLBACK`. Note that the implicit transactions option will affect only the current connection—any other users will still have the option turned off unless they have also executed the `SET` statement.

The implicit transactions option is something of a dangerous territory, and I highly recommend that you leave this option off unless you have a very specific reason to turn it on (such as compatibility with code written in another system).

Here's a common scenario: A user calls up and says, "I've been inserting data for the last half hour, and none of my changes are showing." So, you go run a `DBCC OPEN-TRAN` (see Chapter 24), and discover that there's a transaction that's been there for a while—you can take a guess at what's happened. The user has a transaction open, and his or her changes won't appear until that transaction is committed. The user may have done it using an explicit `BEGIN TRANS` statement, but he or she may also have executed some code that turned `implicit transactions` on and then didn't turn it off. A mess follows.

Locks and Concurrency

Concurrency is a major issue for any database system. It addresses the notion of two or more users each trying to interact with the same object at the same time. The nature of that interaction may be different for each user (updating, deleting, reading, inserting), and the ideal way to handle the competition for control of the object changes depending on just what all the users in question are doing and just how

important their actions are. The more users—more specifically, the more transactions—that you can run with reasonable success at the same time, the higher your concurrency is said to be.

In the Online Transaction Processing (OLTP) environment, concurrency is usually the first thing we deal with in data, and it is the focus of most of the database notions put forward in this book. (Online Analytical Processing (OLAP) is usually something of an afterthought—it shouldn't necessarily be that way, but it is.) Dealing with the issue of concurrency can be critical to the performance of your system. At the foundation of dealing with concurrency in databases is a process called *locking*.

Locks are a mechanism for preventing a process from performing an action on an object that conflicts with something already being done on that object. That is, you can't do some things to an object if someone else got there first. What you can and cannot do depends on what the other user is doing. It is also a means of describing what is being done, so the system knows if the second process action is compatible with the first process or not. For example, 1, 2, 10, 100, 1,000, or whatever number of user connections the system can handle are usually all able to share the same piece of data at the same time as long as they all only want the record on a read-only basis. Think of it as being like a crystal shop—lots of people can be in looking at things—even the same thing—as long as they don't go to move it, buy it, or otherwise change it. If more than one person does that at the same time, you're liable to wind up with broken crystal. That's why the shopkeeper usually keeps a close eye on things, and they will usually decide who gets to handle it first.

The SQL Server *lock manager* is that shopkeeper. When you come into the SQL Server “store,” the lock manager asks what is your intent—what it is you’re going to be doing. If you say “just looking,” and no one else already there is doing anything but “just looking,” then the lock manager will let you in. If you want to “buy” (update or delete) something, then the lock manager will check to see if anyone’s already there. If so, then you must wait, and everyone who comes in behind you will also wait. When you are let in to “buy,” no one else will be let in until you are done.

By doing things this way, SQL Server is able to help us avoid a mix of different problems that can be created by concurrency issues. We will examine the possible concurrency problems and how to set a *transaction isolation level* that will prevent each, but for now, let’s move on to what can and cannot be locked, and what kinds of locks are available.

What Problems Can Be Prevented by Locks

Locks can address four major problems:

- ❑ Dirty reads
- ❑ Non-repeatable reads
- ❑ Phantoms
- ❑ Lost updates

Each of these presents a separate set of problems, and can be handled by mix of solutions that usually includes proper setting of the transaction isolation level. Just to help make things useful as you look back at this chapter later, I’m going to include information on which transaction isolation level is appropriate for each of these problems. We’ll take a complete look at isolation levels shortly, but for now, let’s first make sure that we understand what each of these problems is all about.

Dirty Reads

Dirty reads occur when a transaction reads a record that is part of another transaction that isn't complete yet. If the first transaction completes normally, then it's unlikely there's a problem. But what if the transaction were rolled back? You would have information from a transaction that never happened from the database's perspective!

Let's look at it in an example series of steps:

Transaction 1 Command	Transaction 2 Command	Logical Database Value	Uncommitted Database Value	What Transaction 2 Shows
BEGIN TRAN		3		
UPDATE col = 5	BEGIN TRAN	3	5	
SELECT anything	SELECT @var = col	3	5	5
ROLLBACK	UPDATE anything SET whatever = @var	3		5

Oops—problem!!!

Transaction 2 has now made use of a value that isn't valid! If you try to go back and audit to find where this number came from, you'll wind up with no trace and an extremely large headache.

Fortunately, this scenario can't happen if you're using the SQL Server default for the transaction isolation level (called `READ COMMITTED`, which will be explained later in the section "Setting the Isolation Level").

Non-Repeatable Reads

It's really easy to get this one mixed up with a dirty read. Don't worry about that—it's only terminology. Just get the concept.

A *non-repeatable read* is caused when you read the record twice in a transaction, and a separate transaction alters the data in the interim. For this one, let's go back to our bank example. Remember that we don't want the value of the account to go below 0 dollars:

Transaction 1	Transaction 2	@Var	What Transaction 1 Thinks Is in The Table	Value in Table
BEGIN TRAN		NULL		125
SELECT @Var = value FROM table	BEGIN TRAN	125	125	125
SET value = value — 50	UPDATE value,		75	

Table continued on following page

Chapter 12

Transaction 1	Transaction 2	@Var	What Transaction 1 Thinks Is in The Table	Value in Table
IF @Var >=100	END TRAN	125	125	75
UPDATE value, SET value = value—100 (Finish, wait for lock to clear, then continue)		125	125 (waiting for lock to clear)	75
		125	75	Either: -25 (If there isn't a CHECK con- straint enforce- > 0) Or: Error 547 (If there is a CHECK)

Again, we have a problem. Transaction 1 has prescanned (which can be a good practice in some instances) to make sure that the value is valid and that the transaction can go through (there's enough money in the account). The problem is that, before the UPDATE was made, Transaction 2 beat Transaction 1 to the punch. If there isn't any CHECK constraint on the table to prevent the negative value, then it will indeed be set to -25—even though it logically appeared that we prevented this through the use of our IF statement.

We can prevent this problem in only two ways:

- Create a CHECK constraint and monitor for the 547 Error.
- Set our ISOLATION LEVEL to be REPEATABLE READ or SERIALIZABLE.

The CHECK constraint seems fairly obvious. The thing to realize here is that you are taking something of a reactive rather than a proactive approach with this method. Nonetheless, in most situations we have a potential for non-repeatable reads, so this would be my preferred choice in most circumstances.

We'll be taking a full look at isolation levels shortly, but for now, suffice to say that there's a good chance that setting it to REPEATABLE READ or SERIALIZABLE is going to cause you as many headaches (or more) as it solves. Still—it's an option.

Phantoms

No—we're not talking the “of the opera” kind here—what we're talking about are records that appear mysteriously, as if unaffected by an UPDATE or DELETE statement that you've issued. This can happen quite legitimately in the normal course of operating your system, and doesn't require any kind of elaborate scenario to illustrate. Here's a classic example of how this happens.

Let's say you are running a fast food restaurant. If you're typical of that kind of establishment, you probably have a fair number of employees working at the “minimum wage” as defined by the government. The government has just decided to raise the minimum wage from \$6.25 to \$6.75 per hour, and you want to run an update on a table called Employees to move anyone making less than \$6.75 per hour up to the new minimum wage. No problem, you say, and you issue the rather simple statement:

```

UPDATE Employees
SET HourlyRate = 6.75
WHERE HourlyRate < 6.75

ALTER TABLE Employees
    ADD ckWage CHECK (HourlyRate >= 6.75)
GO

```

That was a breeze, right? *Wrong!* Just for illustration, we're going to say that you get an error message back:

```

Msg 547, Level 16, State 1, Line 1
ALTER TABLE statement conflicted with COLUMN CHECK constraint 'ckWage'. The
conflict occurred in database 'FastFood', table 'Employees', column 'HourlyRate'.

```

So, you run a quick `SELECT` statement checking for values below \$6.75, and sure enough you find one. The question is likely to come rather quickly, "How did that get there! I just did the `UPDATE` which should have fixed that!" You did run the statement, and it ran just fine—you just got a *phantom*.

The instances of phantom reads are rare and require just the right circumstances to happen. In short, someone performed an `INSERT` statement at the very same time your `UPDATE` was running. Since it was an entirely new row, it didn't have a lock on it, and it proceeded just fine.

The only cure for this is setting your transaction isolation level to `SERIALIZABLE`, in which case any updates to the table must not fall within your `WHERE` clause, or they will be locked out.

Lost Updates

Lost updates happen when one update is successfully written to the database but is accidentally overwritten by another transaction. I can just hear you right about now, "Yikes! How could that happen?"

Lost updates can happen when two transactions read an entire record, then one writes updated information back to the record, and the other writes updated information back to the record. Let's look at an example.

Let's say that you are a credit analyst for your company. You get a call that customer X has reached his or her credit limit and would like an extension, so you pull up the customer information to take a look. You see that they have a credit limit of \$5,000, and that they appear to always pay on time.

While you're looking, Sally, another person in your credit department, pulls up customer X's record to enter a change in the address. The record she pulls up also shows the credit limit of \$5,000.

At this point, you decide to go ahead and raise customer X's credit limit to \$7,500, and press enter. The database now shows \$7,500 as the credit limit for customer X.

Sally now completes her update to the address, but she's using the same edit screen that you are—that is, she updates the entire record. Remember what her screen showed as the credit limit? \$5,000. Oops, the database now shows customer X with a credit limit of \$5,000 again. Your update has been lost!

The solution to this depends on your code somehow recognizing that another connection has updated your record between the time when you read the data and when you went to update it. How this recognition happens varies depending on what access method you're using.

Lockable Resources

There are six different *lockable resources* for SQL Server, and they form a hierarchy. The higher level the lock, the less *granularity* it has (that is, you're choosing a higher and higher number of objects to be locked in something of a cascading action just because the object that contains them has been locked). These include, in ascending order of granularity:

- ❑ **Database**—The entire database is locked. This happens usually during database schema changes.
- ❑ **Table**—The entire table is locked. This includes all the data-related objects associated with that table, including the actual data rows (every one of them) and all the keys in all the indexes associated with the table in question.
- ❑ **Extent**—The entire extent is locked. Remember that an extent is made up of eight pages, so an extent lock means that the lock has control of the extent, the eight data or index pages in that extent, and all the rows of data in those eight pages.
- ❑ **Page**—All the data or index keys on that page are locked.
- ❑ **Key**—There is a lock on a particular key or series of keys in an index. Other keys in the same index page may be unaffected.
- ❑ **Row or Row Identifier (RID)**—Although the lock is technically placed on the row identifier (an internal SQL Server construct), it essentially locks the entire row.

Lock Escalation and Lock Effects on Performance

Escalation is all about recognizing that maintaining a finer level of granularity (say a row lock instead of a page lock) makes a lot of sense when the number of items being locked is small. However, as we get more and more items locked, the overhead associated with maintaining those locks actually hinders performance. It can cause the lock to be in place longer (thus creating contention issues—the longer the lock is in place, the more likely that someone will want that particular record). When you think about this for a bit, you'll realize there's probably a balancing act to be done somewhere, and that's exactly what the lock manager uses escalation to do.

When the number of locks being maintained reaches a certain threshold, the lock is escalated to the next highest level, and the lower level locks do not have to be so tightly managed (freeing resources and helping speed over contention).

Note that the escalation is based on the number of locks rather than the number of users. The importance here is that you can single-handedly lock a table by performing a mass update—a row lock can graduate to a page lock, which then escalates to a table lock. That means that you could potentially be locking every other user out of the table. If your query makes use of multiple tables, it's actually quite possible to wind up locking everyone out of all of those tables.

While you certainly would prefer not to lock all the other users out of your object, there are times when you still need to perform updates that are going to have that effect. There is very little you can do about escalation other than to keep your queries as targeted as possible. Recognize that escalations will happen, so make sure you've thought about what the possible ramifications of your query are.

Lock Modes

Beyond considering just what resource level you’re locking, you also should consider what lock mode your query is going to acquire. Just as there are a variety of resources to lock, there are also a variety of *lock modes*.

Some modes are exclusive of each other (which means they don’t work together). Some modes do nothing more than essentially modify other modes. Whether modes can work together is based on whether they are *compatible* (we’ll take a closer look at compatibility between locks later in this chapter).

Just as we did with lockable resources, let’s take a look at lock modes one by one.

Shared Locks

This is the most basic type of lock there is. A *shared lock* is used when you only need to read the data—that is you won’t be changing anything. A shared lock wants to be your friend, as it is compatible with other shared locks. That doesn’t mean that it still won’t cause you grief—while a shared lock doesn’t mind any other kind of lock, there are other locks that don’t like shared locks.

Shared locks tell other locks that you’re out there. It’s the old, “Look at me! Ain’t I special?” thing. They don’t serve much of a purpose, yet they can’t really be ignored. However, one thing that shared locks do is prevent users from performing dirty reads.

Exclusive Locks

Exclusive locks are just what they sound like. Exclusive locks are not compatible with any other lock. They cannot be achieved if any other lock exists, nor will they allow a new lock of any form to be created on the resource while the exclusive lock is still active. This prevents two people from updating, deleting, or doing whatever at the same time.

Update Locks

Update locks are something of a hybrid between shared locks and exclusive locks. An update lock is a special kind of placeholder. Think about it—in order to do an `UPDATE`, you need to validate your `WHERE` clause (assuming there is one) to figure out just what rows you’re going to be updating. That means that you only need a shared lock, until you actually go to make the physical update. At the time of the physical update, you’ll need an exclusive lock.

Update locks indicate that you have a shared lock that’s going to become an exclusive lock after you’ve done your initial scan of the data to figure out what exactly needs to be updated. This acknowledges the fact that there are two distinct stages to an update:

- ❑ First, the stage where you are figuring out what meets the `WHERE` clause criteria (what’s going to be updated). This is the part of an update query that has an update lock.
- ❑ Second, the stage where, if you actually decide to perform the update, the lock is upgraded to an exclusive lock. Otherwise, the lock is converted to a shared lock.

What’s nice about this is that it forms a barrier against one variety of *deadlock*. A deadlock is not a type of lock in itself but rather a situation where a paradox has been formed. A deadlock would arise if one lock

Chapter 12

can't do what it needs to do in order to clear because another lock is holding that resource—the problem is that the opposite resource is itself stuck waiting for the lock to clear on the first transaction.

Without update locks, these deadlocks would crop up all the time. Two update queries would be running in shared mode. Query A completes its query and is ready for the physical update. It wants to escalate to an exclusive lock, but it can't because Query B is finishing its query. Query B then finishes the query, except that it needs to do the physical update. In order to do that, Query B must escalate to an exclusive lock, but it can't because Query A is still waiting. This creates an impasse.

Instead, an update lock prevents any other update locks from being established. The instant that the second transaction attempts to achieve an update lock, they will be put into a wait status for whatever the lock timeout is—the lock will not be granted. If the first lock clears before the lock timeout is reached, then the lock will be granted to the new requester, and that process can continue. If not, an error will be generated.

Update locks are compatible only with shared locks and intent shared locks.

Intent Locks

An *intent lock* is a true placeholder and is meant to deal with the issue of object hierarchies. Imagine a situation where you have a lock established on a row, but someone wants to establish a lock on a page, or extent, or modify a table. You wouldn't want another transaction to go around yours by going higher up the hierarchy, would you?

Without intent locks, the higher-level objects wouldn't even know that you had the lock at the lower level. Intent locks improve performance, as SQL Server needs to examine intent locks only at the table level, and not check every row or page lock on the table, to determine if a transaction can safely lock the entire table. Intent locks come in three different varieties:

- ❑ **Intent shared lock**—A shared lock has or is going to be established at some lower point in the hierarchy. For example, a page is about to have a page level shared lock established on it. This type of lock applies only to tables and pages.
- ❑ **Intent exclusive lock**—This is the same as intent shared, but with an exclusive lock about to be placed on the lower-level item.
- ❑ **Shared with intent exclusive lock**—A shared lock has or is about to be established lower down the object hierarchy, but the intent is to modify data, so it will become an intent exclusive at some point.

Schema Locks

These come in two flavors:

- ❑ **Schema modification lock (Sch-M)**—A schema change is being made to the object. No queries or other CREATE, ALTER, or DROP statements can be run against this object for the duration of the Sch-M lock.
- ❑ **Schema stability lock (Sch-S)**—This is very similar to a shared lock; this lock's sole purpose is to prevent a Sch-M since there are already locks for other queries (or CREATE, ALTER, DROP statements) active on the object. This is compatible with all other lock types.

Bulk Update Locks

A *bulk update lock* (BU) is really just a variant of a table lock with one little (but significant) difference. Bulk update locks will allow parallel loading of data—that is, the table is locked from any other “normal” (T-SQL Statements) activity, but multiple `BULK INSERT` or `bcp` operations can be performed at the same time.

Lock Compatibility

The table that follows shows the compatibility of the resource lock modes (listed in increasing lock strength). Existing locks are shown by the columns; requested locks by the rows:

	IS	S	U	IX	SIX	X
Intent Shared (IS)	YES	YES	YES	YES	YES	NO
Shared (S)	YES	YES	YES	NO	NO	NO
Update (U)	YES	YES	NO	NO	NO	NO
Intent Exclusive (IX)	YES	NO	NO	YES	NO	NO
Shared with Intent Exclusive (SIX)	YES	NO	NO	NO	NO	NO
Exclusive (X)	NO	NO	NO	NO	NO	NO

Also:

- The Sch-S is compatible with all lock modes except the Sch-M.
- The Sch-M is incompatible with all lock modes.
- The BU is compatible only with schema stability and other bulk update locks.

Specifying a Specific Lock Type — Optimizer Hints

Sometimes you want to have more control over how the locking goes either in your query, or perhaps in your entire transaction. You can do this by making use of what are called *optimizer hints*.

Optimizer hints are ways of explicitly telling SQL Server to escalate a lock to a specific level. They are included right after the name of the table (in your SQL Statement) that they are to act against, and are designated as follows:

Hint	Description
SERIALIZABLE/HOLDLOCK	Once a lock is established by a statement in a transaction, that lock is not released until the transaction is ended (via <code>ROLLBACK</code> or <code>COMMIT</code>). Inserts are also prevented if the inserted record would match the criteria in the <code>WHERE</code> clause in the query that established the lock (no phantoms). This is the highest isolation level, and guarantees absolute consistency of data.

Table continued on following page

Chapter 12

Hint	Description
READUNCOMMITTED/NOLOCK	Obtains no lock (not even a shared lock) and does not honor other locks. While a very fast option, it can generate dirty reads as well as a host of other problems.
READCOMMITTED	The Default. Honors all locks, but how it handles acquiring locks depends on the database option READ_COMMITTED_SNAPSHOT. If that setting is on, then READCOMMITTED will <i>not</i> acquire locks, and will instead use a row versioning scheme to determine whether any conflicts have occurred. In practice, this should work just fine, and READCOMMITTED should be the way for you to go for both backward compatibility and what is likely better performance.
READCOMMITTEDLOCK	This is nuance stuff here. Consider this one to be largely the same as READCOMMITTED in most situations (indeed, this one works exactly as READCOMMITTED did in prior versions of SQL Server). Honors all locks but releases any locks held as soon as the object in question is no longer needed. Performs the same as the READ COMMITTED isolation level.
REPEATABLEREAD	Once a lock is established by a statement in a transaction, that lock is not released until the transaction is ended (via ROLLBACK or COMMIT). New data can be inserted, however.
READPAST	Rather than waiting for a lock to clear, skips all locked rows. The skip is limited to row locks (still waits for page, extent, and table locks) and can only be used with a SELECT statement.
ROWLOCK	This forces the initial level of the lock to be at the row level, even if the optimizer would have otherwise selected a less granular locking strategy. It does not prevent the lock from being escalated to those less granular levels if the number of locks reaches the system's lock threshold.
PAGLOCK	Uses a page-level lock regardless of the choice that otherwise would have been made by the optimizer. The usefulness of this can go both ways—sometimes you know that a page lock is more appropriate than a row lock for resource conservation—other times you want to minimize contention where the optimizer might have chosen a table lock.
TABLOCK	Forces a full table lock rather than whatever the lock manager would have used. Can really speed up known table scan situations but creates big contention problems if other users want to modify data in the table.
TABLOCKX	Similar to TABLOCK, but creates an exclusive lock—locks all other users out of the table for the duration of the statement or transaction depending on how the TRANSACTION ISOLATION LEVEL is set.

Hint	Description
UPDLOCK	Uses an update lock instead of a shared lock. This is a highly underutilized tool in the war against deadlocks, as it still allows other users to obtain shared locks but ensures that no data modification (other update locks) are established until you end the statement or transaction (presumably after going ahead and updating the rows).
XLOCK	With its roots in TABLOCKX, this one first appeared in SQL Server 2000. The advantage here is that you can specify an exclusive lock regardless of what lock granularity you have chosen (or not chosen) to specify.

Most of these have times when they can be very useful, but, before you get too attached to using these, make sure that you also check out the concept of isolation levels later in the chapter.

The syntax for using them is fairly easy—just add it after the table name, or after the alias if you’re using one:

```
....  
FROM <table name> [AS <alias>] [[WITH](<hint>)]
```

So, to put this into a couple of examples, any of these would be legal, and all would force a table lock (rather than the more likely key or row lock) on the SalesOrderHeader table:

```
SELECT * FROM Sales.SalesOrderHeader AS ord WITH (TABLOCKX)  
  
SELECT * FROM Sales.SalesOrderHeader AS ord (TABLOCKX)  
  
SELECT * FROM Sales.SalesOrderHeader WITH (TABLOCKX)  
  
SELECT * FROM Sales.SalesOrderHeader (TABLOCKX)
```

Now look at it from a multiple-table perspective. The queries below would do the same thing as those above in terms of locking—they would force an exclusive table lock on the SalesOrderHeader table. The thing to note, though, is that they do *not* place any kind of special lock on the SalesOrderDetail table—the SQL Server lock manager still is in complete control of that table.

```
SELECT *  
FROM Sales.SalesOrderHeader AS ord WITH (TABLOCKX)  
JOIN Sales.SalesOrderDetail AS od  
    ON ord.SalesOrderID = od.SalesOrderID  
SELECT *  
FROM Sales.SalesOrderHeader AS ord (TABLOCKX)  
JOIN Sales.SalesOrderDetail AS od  
    ON ord.SalesOrderID = od.SalesOrderID  
SELECT *  
FROM Sales.SalesOrderHeader WITH (TABLOCKX)
```

```
JOIN Sales.SalesOrderDetail AS od
    ON Sales.SalesOrderHeader.SalesOrderID = od.SalesOrderID
SELECT *
FROM Sales.SalesOrderHeader (TABLOCKX)
JOIN Sales.SalesOrderDetail AS od
    ON Sales.SalesOrderHeader.SalesOrderID = od.SalesOrderID
```

We also could have done something completely different here and placed a totally separate hint on the SalesOrderDetail table—it's all up to you.

Determining Locks Using the Management Studio

Perhaps the nicest way of all to take a look at your locks is by using Management Studio. Management Studio will show you locks in two different sorts—by *process ID* or by *object*—by utilizing the Activity Monitor.

To make use of Management Studio's lock display, just navigate to the Management | Activity Monitor node of your server. Then right-click and choose the kind of information you're after. You should come up with a new window that looks something like Figure 12-3.

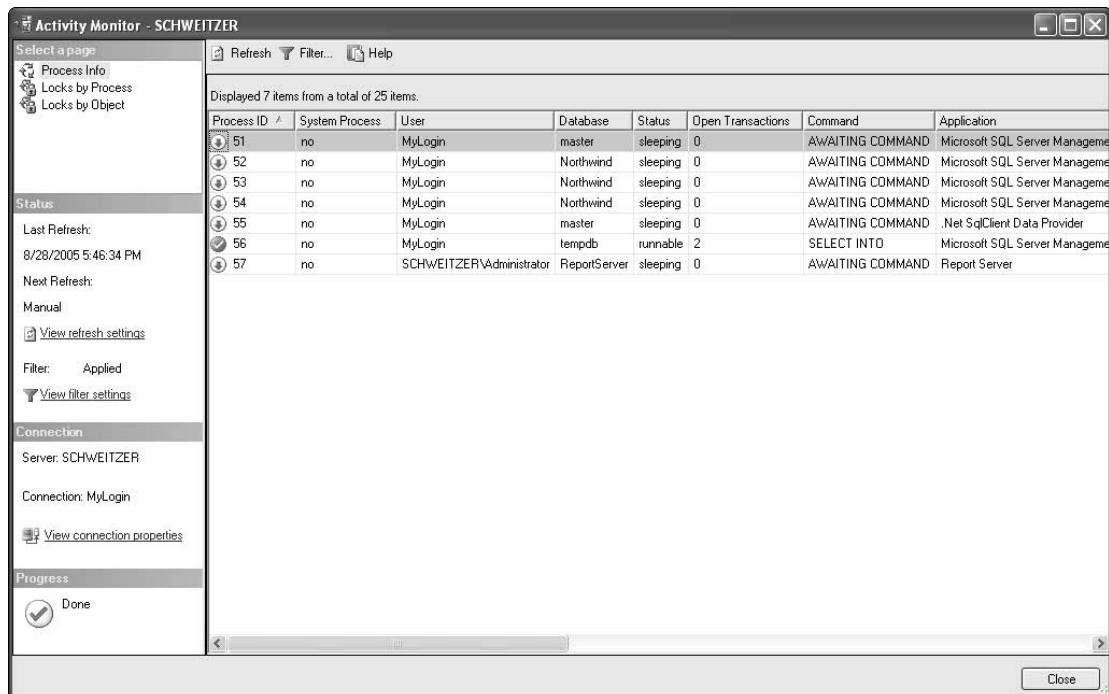


Figure 12-3

Just expand the node that you're interested in (either the *Process ID* or the *Object*), and you'll see various locks.

Perhaps the coolest feature in this shows itself when you double-click on a specific lock in the right-hand side of the window. A dialog box will come up and tell you the last statement that was run by that process ID. This can be very handy when you are troubleshooting deadlock situations.

Setting the Isolation Level

We've seen that several different kinds of problems that can be prevented by different locking strategies. We've also seen what kinds of locks are available and how they have an impact on the availability of resources. Now it's time to take a closer look at how these process management pieces work together to ensure overall data integrity — to make certain that you can get the results you expect.

The first thing to understand about the relationship between transactions and locks is that they are inextricably linked with each other. By default, any lock that is data modification-related will, once created, be held for the duration of the transaction. If you have a long transaction, this means that your locks may be preventing other processes from accessing the objects you have a lock on for a rather long time. It probably goes without saying that this can be rather problematic.

However, that's only the default. In fact, there are actually four different *isolation levels* that you can set:

- READ COMMITTED (the default)
- READ UNCOMMITTED
- REPEATABLE READ
- SERIALIZABLE

The syntax for switching between them is pretty straightforward:

```
SET TRANSACTION ISOLATION LEVEL <READ COMMITTED|READ UNCOMMITTED  
|REPEATABLE READ|SERIALIZABLE>
```

The change in isolation level will affect only the current connection — so you don't need to worry about adversely affecting other users (or them affecting you).

Let's start by looking at the default situation (READ COMMITTED) a little more closely.

READ COMMITTED

With READ COMMITTED, any shared locks you create will be automatically released as soon as the statement that created them is complete. That is, if you start a transaction, run several statements, run a SELECT statement, and then run several more statements, the locks associated with the SELECT statement are freed as soon as the SELECT statement is complete — SQL Server doesn't wait for the end of the transaction.

Action queries (UPDATE, DELETE, and INSERT) are a little different. If your transaction performs a query that modifies data, then those locks will be held for the duration of the transaction (in case you need to roll back).

By keeping this level of default, with READ COMMITTED, you can be sure that you have enough data integrity to prevent dirty reads. However, non-repeatable reads and phantoms can still occur.

READ UNCOMMITTED

`READ UNCOMMITTED` is the most dangerous of all isolation level choices but also has the highest performance in terms of speed.

Setting the isolation level to `READ UNCOMMITTED` tells SQL Server not to set any locks, and not to honor any locks. With this isolation level, it is possible to experience any of the various concurrency issues we discussed earlier in the chapter (most notably a dirty read).

Why would one ever want to risk a dirty read? When I watch the newsgroups on Usenet, I see the question come up on a regular basis. It's surprising to a fair number of people, but there are actually good reasons to have this isolation level, and they are almost always to do with reporting.

In an OLTP environment, locks are both your protector and your enemy. They prevent data integrity problems, but they also often prevent, or block, you from getting at the data you want. It is extremely commonplace to see a situation where the management wants to run reports regularly, but the data entry people are often prevented from or delayed in entering data because of locks held by the manager's reports.

By using `READ UNCOMMITTED`, you can often get around this problem—at least for reports where the numbers don't have to be exact. For example, let's say that a sales manager wants to know just how much has been done in sales so far today. Indeed, we'll say he's a micro-manager and asks this same question (in the form of re-running the report) several times a day.

If the report happened to be a long-running one, then there's a high chance that his running it would damage the productivity of other users due to locking considerations. What's nice about this report though, is that it is a truly nebulous report—the exact values are probably meaningless. The manager is really just looking for ballpark numbers.

By having an isolation level of `READ UNCOMMITTED`, we do not set any locks, so we don't block any other transactions. Our numbers will be somewhat suspect (because of the risk of dirty reads), but we don't need exact numbers anyway, and we know that the numbers are still going to be close even on the off chance that a dirty read is rolled back.

You can get the same effect as `READ UNCOMMITTED` by adding the `NOLOCK` optimizer hint in your query. The advantage to setting the isolation level is that you don't have to use a hint for every table in your query, or use it in multiple queries. The advantage to using the `NOLOCK` optimizer hint is that you don't need to remember to set the isolation level back to the default for the connection. (With `READ UNCOMMITTED` you do.)

REPEATABLE READ

The `REPEATABLE READ` escalates your isolation level somewhat, and provides an extra level of concurrency protection by preventing not only dirty reads (the default already does that) but also preventing non-repeatable reads.

That prevention of non-repeatable reads is a big upside, but holding even shared locks until the end of the transaction can block users' access to objects, and therefore hurt productivity. Personally, I prefer to

use other data integrity options (such as a CHECK constraint together with error handling) rather than this choice, but it remains an available option.

The equivalent optimizer hint for the REPEATABLE READ isolation level is REPEATABLEREAD (these are the same, only no space).

SERIALIZABLE

SERIALIZABLE is something of the fortress of isolation levels. It prevents all forms of concurrency issues except for a lost update. Even phantoms are prevented.

When you set your isolation to SERIALIZABLE, you're saying that any UPDATE, DELETE, or INSERT to the table or tables used by your transaction must not meet the WHERE clause of any statement in that transaction. Essentially, if the user was going to do something that your transaction would be interested in, then it must wait until your transaction has been completed.

The SERIALIZABLE isolation level can also be simulated by using the SERIALIZABLE or HOLDLOCK optimizer hint in your query. Again, as with the READ UNCOMMITTED and NOLOCK debate, the option of not having to set it every time versus not having to remember to change the isolation level back is the big issue.

Going with an isolation level of SERIALIZABLE would, on the surface, appear to be the way you want to do everything. Indeed, it does provide your database with the highest level of what is called consistency—that is, the update process works the same for multiple users as it would if all your users did one transaction at a time (processed things serially).

As with most things in life, however, there is a trade-off. Consistency and concurrency can, in a practical sense, be thought of as polar opposites. Making things SERIALIZABLE can prevent other users from getting to the objects they need—that equates to lower concurrency. The reverse is also true—increasing concurrency (by going to a REPEATABLE READ for example) reduces the consistency of your database.

My personal recommendation on this is to stick with the default (READ COMMITTED) unless you have a specific reason not to.

Dealing with Deadlocks (a.k.a. “A 1205”)

Okay. So now you've seen locks, and you've also seen transactions. Now that you've got both, we can move on to the rather pesky problem of dealing with *deadlocks*.

As we've already mentioned, a deadlock is not a type of lock in itself, but rather a situation where a paradox has been formed by other locks. Like it or not, you'll bump into these on a regular basis (particularly when you're just starting out), and you'll be greeted with an *error number 1205*. So, prolific is this particular problem that you'll hear many a database developer refer to them simply by the number.

Deadlocks are caused when one lock can't do what it needs to do in order to clear because a second lock is holding that resource, and vice versa. When this happens, somebody has to win the battle, so SQL Server chooses a deadlock *victim*. The deadlock victim's transaction is then rolled back and is notified that this happened through the 1205 error. The other transaction can continue normally (indeed, it will be entirely unaware that there was a problem, other than seeing an increased execution time).

How SQL Server Figures Out There's a Deadlock

Every 5 seconds SQL Server checks all the current transactions for what locks they are waiting on but haven't yet been granted. As it does this, it essentially makes a note that the request exists. It will then re-check the status of all open lock requests again, and, if one of the previous requests has still not been granted, it will recursively check all open transactions for a circular chain of lock requests. If it finds such a chain, then one or more deadlock victims will be chosen.

How Deadlock Victims Are Chosen

By default, a deadlock victim is chosen based on the "cost" of the transactions involved. The transaction that costs the least to rollback will be chosen (in other words SQL Server has to do the least number of things to undo it). You can, to some degree override this by using the DEADLOCK_PRIORITY SET option available in SQL Server; this is, however, generally both ill-advised and out of the scope of this book.

Avoiding Deadlocks

Deadlocks can't be avoided 100 percent of the time in complex systems, but you can almost always totally eliminate them from a practical standpoint—that is, make them so rare that they have little relevance to your system.

To cut down or eliminate deadlocks, follow these simple (okay, usually simple) rules:

- ❑ Use your objects in the same order.
- ❑ Keep your transactions as short as possible and in one batch.
- ❑ Use the lowest transaction isolation level necessary.
- ❑ Do not allow open-ended interruptions (user interactions, batch separations) within the same transaction.
- ❑ In controlled environments, use bound connections.

Nearly every time I run across deadlocking problems, at least one (usually more) of these rules has been violated. Let's look at each one individually.

Using Objects in the Same Order

This is the most common problem area within the few rules that I consider to be basic. What's great about using this rule is that it almost never costs you anything to speak of—it's more a way of thinking. You decide early in your design process how you want to access your database objects—including order—and it becomes a habit in every query, procedure, or trigger that you write for that project.

Think about it for a minute—if our problem is that our two connections each have what the other wants, then it implies that we're dealing with the problem too late in the game. Let's look at a simple example.

Consider that we have two tables: `Suppliers` and `Products`. Now say that we have two processes that make use of both of these tables. Process 1 accepts inventory entries, updates `Products` with the new amount of product on hand, and then updates `Suppliers` with the total amount of product that we've purchased. Process 2 records sales; it updates the total amount of product sold in the `Suppliers` table and then decreases the inventory quantity in `Products`.

If we run these two processes at the same time, we're begging for trouble. Process 1 will grab an exclusive lock on the `Products`. Process 2 grabs an exclusive lock on the `Suppliers` table. Process 1 then attempts to grab a lock on the `Suppliers` table, but it will be forced to wait for Process 2 to clear its existing lock. In the meantime, Process 2 tries to create a lock on the `Products` table, but it will have to wait for Process 1 to clear its existing lock. We now have a paradox—both processes are waiting on each other. SQL Server will have to pick a deadlock victim.

Now let's rearrange that scenario, with Process 2 changed to first decrease the inventory quantity in `Products` and then update the total amount of product sold in the `Suppliers` table. This is a functional equivalent to the first way we organized the processes, and it will cost us nothing to perform it this new way. The impact though, will be stunning—no more deadlocks (at least not between these two processes)! Let's walk through what will now happen.

When we run these two processes at the same time, Process 1 will grab an exclusive lock on the `Products` table (so far, it's the same). Process 2 then also tries to grab a lock on the `Products` table but will be forced to wait for Process 1 to finish (notice that we haven't done anything with `Suppliers` yet). Process 1 finishes with the `Products` table but doesn't release the lock because the transaction isn't complete yet. Process 2 is still waiting for the lock on `Products` to clear. Process 1 now moves on to grab a lock on the `Suppliers` table. Process 2 continues to wait for the lock to clear on `Products`. Process 1 finishes and commits or rolls back the transaction as required but frees all locks in either case. Process 2 now is able to obtain its lock on the `Products` table and moves through the rest of its transaction without further incident.

Just swapping the order in which these two queries are run has eliminated a potential deadlock problem. Keep things in the same order wherever possible and you, too, will experience far less in the way of deadlocks.

Keeping Transactions As Short As Possible

This is another of the basics. Again, it should become just an instinct—something you don't really think about, something you just do.

This is one that never has to cost you anything really. Put what you need to put in the transaction, and keep everything else out—it's just that simple. Why this works isn't rocket science—the longer the transaction is open, and the more it touches (within the transaction), the higher the likelihood that you're going to run into some other process that wants one or more of the objects that you're using (reducing concurrency). If you keep your transaction short, you minimize the number of objects that can potentially cause a deadlock, plus you cut down on the time that you have your lock on them. It's as simple as that.

Keeping transactions in one batch minimizes network roundtrips during a transaction, reducing possible delays in completing the transaction and releasing locks.

Using the Lowest Transaction Isolation Level Possible

This one is considerably less basic, and requires some serious thought. As such, it isn't surprising just how often it isn't thought of at all. Consider it Rob's axiom—that which requires thought is likely not to be thought of. Be different—think about it.

We have several different transaction isolation levels available. The default is `READ COMMITTED`. Using a lower isolation level holds shared locks for a shorter duration than a higher isolation level, thereby reducing locking contention.

Allowing No Open-Ended Transactions

This is probably the most common sense out of all the recommendations here—but it's one that's often violated because of past practices.

One of the ways we used to prevent lost updates (mainframe days here folks!) was just to grab the lock and hold it until we were done with it. I can't tell you how problematic this was (can you say *yuck!*).

Imagine this scenario (it's a real-life example): Someone in your service department likes to use update (exclusive locks) screens instead of display (shared locks) screens to look at data. He goes on to look at a work order. Now his buddy calls and asks if he's ready for lunch. "Sure!" comes the reply, and the service clerk heads off to a rather long lunch (1–2 hours). Everyone who is interested in this record is now locked out of it for the duration of this clerk's lunch.

Wait—it gets worse. In the days of the mainframe, you used to see the concept of queuing far more often (it actually can be quite efficient). Now someone submits a print job (which is queued) for this work order. It sits in the queue waiting for the record lock to clear. Since it's a queue environment, every print job your company has for work orders now piles up behind that first print job (which is going to wait for that person's lunch before clearing).

This is a rather extreme example—but I'm hoping that it clearly illustrates the point. Don't ever create locks that will still be open when you begin some form of open-ended process. Usually we're talking user interaction (like our lunch lover), but it could be any process that has an open-ended wait to it.

Using Bound Connections

Hmm. I had to debate even including this one, because it's something of a can of worms. Once you open it, you're never going to get them all back in. I'll just say that this is one is used extremely rarely and is not for the faint of heart.

It's not that it doesn't have its uses; it's just that things can become rather convoluted rather quickly, so you need to manage things well. It's my personal opinion that there is usually a better solution.

That brings on the question of what is a bound connection. *Bound connections* are connections that have been associated and are allowed to essentially share the same set of locks. What that means is that the two transactions can operate in tandem without any fear of deadlocking each other or being blocked by one another. The flip side of that is that it means that you essentially are on your own in terms of dealing with most concurrency issues—locks aren't keeping you safe anymore.

Given my distaste for these for 99.9 percent of situations, we're going to forget that these exist now that we've seen that they are an option. If you're going to insist on using them, just remember that you're going to be dealing with an extremely complex relationship between connections, and you need to manage the activities in those connections rather closely if you are going to maintain data integrity within the system.

Summary

Transactions and locks are both cornerstone items to how SQL Server works and, therefore, to maximizing your development of solutions in SQL Server.

By using transactions, you can make sure that everything you need to have happen as a unit happens, or none of it does. SQL Server's use of locks ensures that we avoid the pitfalls of concurrency to the maximum extent possible (you'll never avoid them entirely, but it's amazing how close you can come with a little—OK a lot—of planning). By using the two together, you are able to pass what the database industry calls the *ACID* test. If a transaction is ACID, then it has:

- Atomicity**—The transaction is all or nothing.
- Consistency**—All constraints and other data integrity rules have been adhered to, and all related objects (data pages, index pages) have been updated completely.
- Isolation**—Each transaction is completely isolated from any other transaction. The actions of one transaction cannot be interfered with by the actions of a separate transaction.
- Durability**—After a transaction is completed, its effects are permanently in place in the system. The data is “safe,” in the sense that things such as a power outage or other non-disk system failure will not lead to data that is only half-written.

In short, by using transactions and locks, you can minimize deadlocks, ensure data integrity, and improve the overall efficiency of your system.

In our next chapter, we'll be looking at triggers. Indeed, we'll see that, for many of the likely uses of triggers, the concepts of transactions and rollbacks will be at the very center of the trigger.

13

Triggers

Hi, ho Trigger! And away!

Okay, so it's a little cliché even for me. What's more, it glorifies what is, although really wonderful, also really dastardly. I am often asked, "Should I use triggers?" The answer is, as with most things in SQL, "It depends." There's little that's black and white in the wonderful world of SQL Server—triggers are definitely a very plain shade of gray.

Know what you're doing before you go the triggers route—it's important for the health and performance of your database. The good news is that's what we're here to learn.

As with most of the core subjects we've covered in this book (save for a few that were just too important to rush), we're going to be moving along quickly in the assumption that you already know the basics. In this chapter, we'll try to look at triggers in all of their colors—from black all the way to white and a whole lot in between. The main issues we'll be dealing with include:

- What is a trigger (the *very* quick and dirty version)?
- Using triggers for more flexible referential integrity
- Using triggers to create flexible data integrity rules
- Using `INSTEAD OF` triggers to create more flexible updateable views
- Other common uses for triggers
- Controlling the firing order of triggers
- Performance considerations

By the time we're done, you should have an idea of just how complex the decision about when and where not to use triggers is. You'll also have an inkling of just how powerful and flexible they can be.

Chapter 13

Most of all, if I've done my job well, you won't be a trigger extremist (which *so* many SQL Server people I meet are) with the distorted notion that triggers are evil and should never be used. Neither will you side with the other end of the spectrum, those who think that triggers are the solution to all the world's problems. The right answer in this respect is that triggers can do a lot for you, but they can also cause a lot of problems. The trick is to use them when they are the right things to use, and not to use them when they aren't.

Some common uses of triggers include:

- ❑ Enforcement of referential integrity: Although I recommend using declarative referential integrity (DRI) whenever possible, there are many things that DRI won't do (for example, referential integrity across databases or even servers, many complex types of relationships, and so on). The use of triggers for RI is becoming very special case, but it's still out there.
- ❑ Creating audit trails, which means writing out records that keep track of not just the most current data but also the actual change history for each record.
- ❑ Functionality similar to a CHECK constraint, but which works across tables, databases, or even servers.
- ❑ Substituting your own statements in the place of a user's action statement (usually used to enable inserts in complex views).

In addition, you have the new but likely much more rare case (as I said, they are new, so only time will tell for sure) DDL trigger—which is about monitoring changes in the structure of your table.

And these are just a few. So, with no further ado, let's look at exactly what a trigger is.

What Is a Trigger?

A trigger is a special kind of stored procedure that responds to specific events. There are two kinds of triggers: Data Definition Language (DDL) triggers and Data Manipulation Language (DML) triggers.

DDL triggers fire in response to someone changing the structure of your database in some way (CREATE, ALTER, DROP, and similar statements). These are new with SQL Server 2005 and are critical to some installations (particularly high-security installations) but are pretty narrow in use. In general, you will need to look into using these only where you need extreme auditing of changes/history of your database structure. We will save these until last.

DML triggers are pieces of code that you attach to a particular table or view. Unlike sprocs, where you needed to explicitly invoke the code, the code in triggers is automatically run whenever the event(s) you attached the trigger to occur in the table. Indeed, you *can't* explicitly invoke triggers—the only way to do this is by performing the required action in the table that they are assigned to.

Beyond not being able to explicitly invoke a trigger, you'll find two other things that exist for sprocs but are missing from triggers: parameters and return codes.

While triggers take no parameters, they do have a mechanism for figuring out what records they are supposed to act on (we'll investigate this further later in the chapter). And, while you can use the RETURN keyword, you cannot return a specific return code (because you didn't explicitly call the trigger, what would you return a return code to?).

What events can you attach triggers to? — the three “action” query types you use in SQL. So, you wind up with triggers based in inserts, updates, and/or deletes (you can mix and match what events you want the trigger to fire based on).

It's worth noting that there are times when a trigger will not fire — even though it seems that the action you are performing falls into one of the preceding categories. At issue is whether the operation you are doing is in a logged activity or not. For example, a DELETE statement is a normal, logged activity that would fire any delete trigger, but a TRUNCATE TABLE, which has the effect of deleting rows, just deallocates the space used by the table — there is no individual deletion of rows logged, and no trigger is fired.

The syntax for creating triggers looks an awful lot like all of our other CREATE syntax, except that it has to be attached to a table somewhat similar to an index — a trigger can't stand on its own.

Let's take a look:

```
CREATE TRIGGER <trigger name>
    ON [<schema name>.]<table or view name>
    [WITH ENCRYPTION | EXECUTE AS <CALLER | SELF | <user> >]
    {{ {FOR|AFTER} <[DELETE] [,] [INSERT] [,] [UPDATE]>} | INSTEAD OF}
    [WITH APPEND]
    [NOT FOR REPLICATION]
AS
    < <sql statements> | EXTERNAL NAME <assembly method specifier> >
```

As you can see, the all too familiar CREATE <object type> <object name> is still there as well as the execution stuff we've seen in many other objects — we've just added the ON clause to indicate the table to which this trigger is going to be attached, as well as when and under what conditions it fires.

ON

This part just names what object you are creating the trigger against. Keep in mind that if the type of the trigger is an AFTER trigger (if it uses FOR or AFTER to declare the trigger), then the target of the ON clause must be a table — AFTER triggers are not supported for views.

WITH ENCRYPTION

This works just as it does for views and sprocs. If you add this option, you can be certain that no one will be able to view your code (not even you!). This is particularly useful if you are going to be building software for commercial distribution, or if you are concerned about security and don't want your users to be able to see what data you're modifying or accessing. Obviously, you should keep a copy of the code required to create the trigger somewhere else, in case you want to re-create it sometime later.

As with views and sprocs, the thing to remember when using the WITH ENCRYPTION option is that you must reapply it every time you ALTER your trigger. If you make use of an ALTER TRIGGER statement and do not include the WITH ENCRYPTION option, then the trigger will no longer be encrypted.

The **FOR|AFTER** versus the **INSTEAD OF** Clause

In addition to deciding what kind of queries will fire your trigger (`INSERT`, `UPDATE`, and/or `DELETE`), you also have some choice as to the timing of when the trigger fires. While the `FOR` (alternatively, you can use the keyword `AFTER` instead if you choose) trigger is the one that has been around a long time and that people generally think of, you also have the ability to run what is called an `INSTEAD OF` trigger. Choosing between these two will affect whether you enter your trigger before the data has been modified or after. In either case, you will be in your trigger before any changes are truly committed to the database.

Confusing? Probably. Let's try it a different way with a diagram that shows where each choice fires (see Figure 13-1).

The thing to note here is that, regardless of which choice you make, SQL Server will put together two working tables—one holding a copy of the records that were inserted (and, incidentally, called `INSERTED`) and one holding a copy of any records that were deleted (called `DELETED`). We'll look into the details of the uses of these working tables a little later. For now realize that with `INSTEAD OF` triggers the creation of these working tables will happen *before* any constraints are checked, and with `FOR` triggers, these tables will be created *after* constraints are checked.

The key to `INSTEAD OF` triggers is that you can actually run your own code in the place of whatever the user requested. This means we can clean up ambiguous insert problems in views (remember the problem back in Chapter 9 with inserting when there was a `JOIN` in the view?). It also means that we can take action to clean up constraint violations before the constraint is even checked.

Triggers using the `FOR` and `AFTER` declaration behave identically to each other. The big difference between them and `INSTEAD OF` triggers is that they build their working tables *after* any constraints have been checked.

The `AFTER` (or, alternatively, you can use `FOR`) clause indicates under what type of action(s) you want this trigger to fire. You can have the trigger fire whenever there is an `INSERT`, `UPDATE`, or `DELETE`, or any mix of the three. So, for example, your `FOR` clause could look something like:

AFTER INSERT, DELETE

... or:

AFTER UPDATE, INSERT

... or:

AFTER DELETE

As was stated in the section about the `ON` clause, triggers declared using the `AFTER` or `FOR` clause can only be attached to tables—no views are allowed (see `INSTEAD OF` triggers for those).

It's worth noting that, unlike prior editions of this book, I actually do advise a specific choice between `AFTER` and `FOR`. While both are equally usable, and there is no indication that either will be deprecated, the `AFTER` clause is the "standard" way of doing things, so it is more likely to be supported by other database vendors.

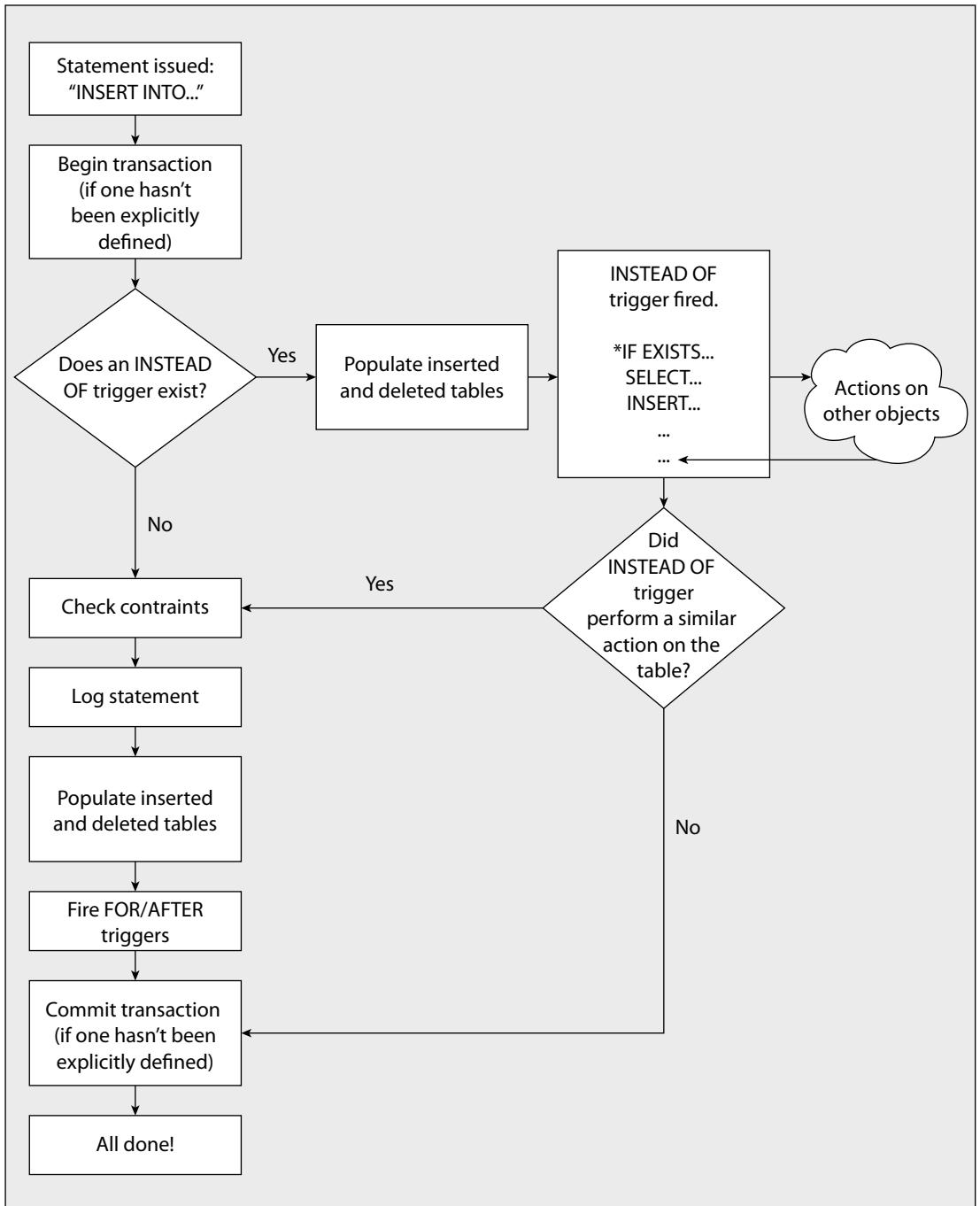


Figure 13-1

INSERT Trigger

The code for any trigger that you mark as being `FOR INSERT` will be executed any time that someone inserts a new row into your table. For each row that is inserted, SQL Server will create a copy of that new row and insert it in a special table that exists only within the scope of your trigger. That table is called `INSERTED`, and we'll see much more of it over the course of this chapter. The big thing to understand is that the `INSERTED` table only lives as long as your trigger does. Think of it as not existing before your trigger starts or after your trigger completes.

DELETE Trigger

This works much the same as an `INSERT` trigger does, save that the `INSERTED` table will be empty (after all, you deleted rather than inserted, so there are no records for the `INSERTED` table). Instead, a copy of each record that was deleted is inserted into another table called `DELETED`. That table, like the `INSERTED` table, is limited in scope to just the life of your trigger.

UPDATE Trigger

More of the same, save for a twist. The code in a trigger declared as being `FOR UPDATE` will be fired whenever an existing record in your table is changed. The twist is that there's no such table as `UPDATED`. Instead, SQL Server treats each row as if the existing record had been deleted, and a totally new record was inserted. As you can probably guess from that, a trigger declared as `FOR UPDATE` contains not one but two special tables called `INSERTED` and `DELETED`. The two tables have exactly the same number of rows, of course.

WITH APPEND

`WITH APPEND` is something of an oddball and, in all honesty, you're pretty unlikely to use it; nonetheless, we'll cover it here for that "just-in-case" scenario. `WITH APPEND` applies only when you are running in 6.5 compatibility mode (which can be set using `sp_dbcmptlevel`).

SQL Server 6.5 and prior did not allow multiple triggers of the same type on any single table. For example, if you had already declared a trigger called `trgCheck` to enforce data integrity on updates and inserts, then you couldn't create a separate trigger for cascading updates. Once one update (or insert, or delete) trigger was created, that was it — you couldn't create another trigger for the same type of action.

This was a real pain. It meant that you had to combine logically different activities into one trigger. Trying to get what amounted to two entirely different procedures to play nicely together could, at times, be quite a challenge. In addition, it made reading the code something of an arduous task.

Along came SQL Server 7.0 and the rules changed substantially. No longer do we have to worry about how many triggers we have for one type of action query — you can have several if you like. When running our database in 6.5 compatibility mode, though, we run into a problem — our database is still working on the notion that there can only be one trigger of a given type on a given table.

`WITH APPEND` gets around this problem by explicitly telling SQL Server that we want to add this new trigger even though we already have a trigger of that type on the table — both will be fired when the appropriate trigger action (`INSERT`, `UPDATE`, `DELETE`) occurs. It's a way of having a bit of both worlds.

NOT FOR REPLICATION

Adding this option slightly alters the rules for when the trigger is fired. With this option in place, the trigger will not be fired whenever a replication-related task modifies your table. Usually a trigger is fired (to do the housekeeping/cascading/etc.) when the original table is modified and there is no point in doing it again.

AS

Exactly as it was with sprocs, this is the meat of the matter. The AS keyword tells SQL Server that your code is about to start. From this point forward, we're into the scripted portion of your trigger.

Using Triggers for Data Integrity Rules

Although they shouldn't be your first option, triggers can also perform the same functionality as a CHECK constraint or even a DEFAULT. The answer to the question "Should I use triggers or CHECK constraints?" is the rather definitive: "It depends." If a CHECK can do the job, then it's probably the preferable choice. There are times, however, when a CHECK constraint just won't do the job, or when something inherent in the CHECK process makes it less desirable than a trigger. Examples of where you would want to use a trigger over a CHECK include:

- Your business rule needs to reference data in a separate table.
- Your business rule needs to check the *delta* (difference between before and after) of an update.
- You require a customized error message.

A summary table of when to use what type of data integrity mechanism is provided at the end of Chapter 5.

This really just scratches the surface of things. Since triggers are highly flexible, deciding when to use them really just comes down to whenever you need something special done.

Dealing with Requirements Sourced from Other Tables

CHECK constraints are great—fast and efficient—but they don't do everything you'd like them to. Perhaps the biggest shortcoming shows up when you need to verify data across tables.

To illustrate this, let's take a look at the Products and SalesOrderDetail tables in AdventureWorks as well as the related SpecialOfferProduct table. The relationship looks like Figure 13-2.

So, under normal DRI, you can be certain that no order line item can be entered into the SalesOrder Detail table unless there is a matching ProductID in the Products table (via the chain through the SpecialOfferProduct table). We are, however, looking for something more than just the "norm" here.

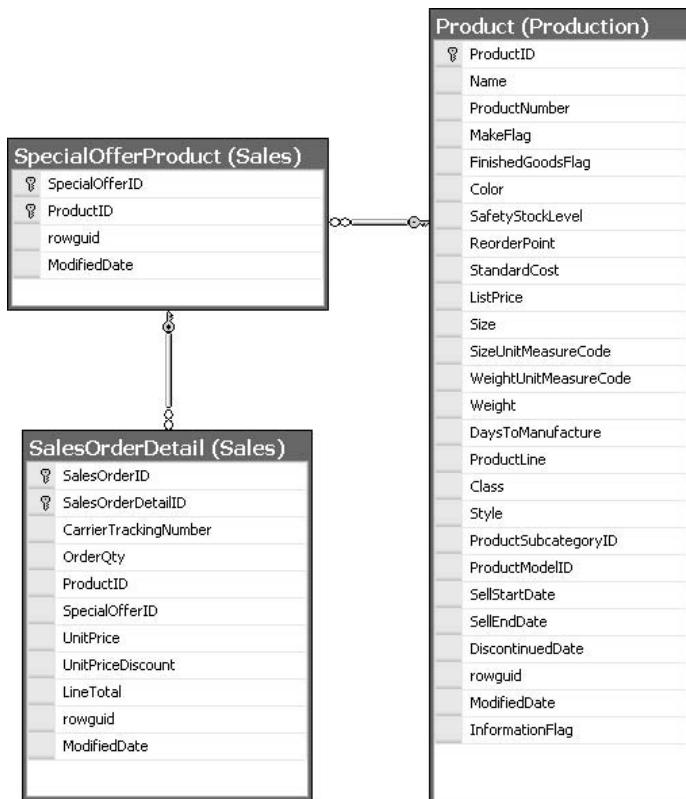


Figure 13-2

Our Inventory department has been complaining that our Customer Support people keep placing orders for products that are discontinued. They would like to have such orders rejected before they get into the system.

We can't deal with this using a CHECK constraint because the place where we know about the discontinued status (the `Products` table) is in a separate table from where we are placing the restriction (the `SalesOrderDetail` table). Don't sweat it though—you can tell the Inventory department, "No problem!" You just need to use a trigger:

```
USE AdventureWorks
GO

CREATE TRIGGER OrderDetailNotDiscontinued
    ON Sales.SalesOrderDetail
    AFTER INSERT, UPDATE
AS
    IF EXISTS
    (
        SELECT 'True'
        FROM Inserted i
```

```

        JOIN Production.Product p
          ON i.ProductID = p.ProductID
         WHERE p.DiscontinuedDate IS NOT NULL
      )
BEGIN
  RAISERROR('Order Item is discontinued. Transaction Failed.',16,1)
  ROLLBACK TRAN
END

```

Let's go ahead and test out our handiwork. First, we need at least one record that will fail when it hits our trigger. That means we need a discontinued item in the `Products` table — the problem is, there is no such record currently.

```

SELECT ProductID, Name FROM Production.Product WHERE DiscontinuedDate IS NOT NULL
ProductID    Name
-----
(0 row(s) affected)

```

So, we'll pick one and change it ourselves for test purposes:

```

UPDATE Production.Product
SET DiscontinuedDate = GETDATE()
WHERE ProductID = 680

```

With that done, we're ready to see if our trigger works, so let's go ahead and add a line item that violates this constraint. I'm going to make use of a `SalesOrderHeader` that already exists, so we don't have to get over elaborate building up a full order:

```

INSERT Sales.SalesOrderDetail
  (SalesOrderID, OrderQty, ProductID, SpecialOfferID, UnitPrice, UnitPriceDiscount)
VALUES
  (43660, 5, 680, 1, 1431, 0)

```

This gets the rejection that we expect:

```

Msg 50000, Level 16, State 1, Procedure OrderDetailNotDiscontinued, Line 14
Order Item is discontinued. Transaction Failed.
Msg 3609, Level 16, State 1, Line 1
The transaction ended in the trigger. The batch has been aborted.

```

Remember that we could, if desired, also create a custom error message to raise, instead of the ad hoc message that we used with the `RAISERROR` command.

Using Triggers to Check the Delta of an Update

Sometimes, you're not interested as much in what the value was or is as you are in how much it changed. While there isn't any one column or table that gives you that information, you can calculate it by making use of both the `Inserted` and `Deleted` tables in your trigger.

Chapter 13

A quick example of this might be to write records for security reasons. Let's say, for example, that you wanted to track every adjustment to inventory regardless of what initiated it for auditing purposes (for example, inventory adjustments might be made directory against inventory tables rather than via an order item).

To implement something like this, we would need an audit table and to make use of both the `Inserted` and `Deleted` tables:

```
CREATE TABLE Production.InventoryAudit
(
    TransactionID      int          IDENTITY PRIMARY KEY,
    ProductID         int          NOT NULL
        REFERENCES Production.Product(ProductID),
    NetAdjustment     smallint     NOT NULL,
    ModifiedDate      datetime    DEFAULT(CURRENT_TIMESTAMP)
)
GO

CREATE TRIGGER ProductAudit
    ON Production.ProductInventory
    FOR INSERT, UPDATE, DELETE
AS
    INSERT INTO Production.InventoryAudit
    (ProductID, NetAdjustment)
        SELECT COALESCE(i.ProductID, d.ProductID),
               ISNULL(i.Quantity, 0) - ISNULL(d.Quantity, 0) AS NetAdjustment
        FROM Inserted i
        FULL JOIN Deleted d
            ON i.ProductID = d.ProductID
            AND i.LocationID = d.LocationID
        WHERE ISNULL(i.Quantity, 0) - ISNULL(d.Quantity, 0) != 0
```

Before we test this, let's analyze what we're doing here. I've started by adding an audit table to receive information about changes to our base table. From there, I've created a trigger that will fire on any change to the table and will write the next change out to our new audit table.

Now, let's check this out by running a test script:

```
PRINT 'Now making the change'
UPDATE Production.ProductInventory
SET Quantity = Quantity + 7
WHERE ProductID = 1
    AND LocationID = 50

PRINT 'The values before the change are:'
SELECT ProductID, LocationID, Quantity
FROM Production.ProductInventory
WHERE ProductID = 1
    AND LocationID = 50

PRINT 'Now making the change'
UPDATE Production.ProductInventory
```

```

SET Quantity = Quantity - 7
WHERE ProductID = 1
    AND LocationID = 50

PRINT 'The values after the change are:'
SELECT ProductID, LocationID, Quantity
FROM Production.ProductInventory
WHERE ProductID = 1
    AND LocationID = 50

SELECT * FROM Production.InventoryAudit

```

And we can use the before and after output to verify that our audit records were properly written:

```

The values before the change are:
ProductID  LocationID  Quantity
-----  -----  -----
1          50          346

(1 row(s) affected)

Now making the change

(1 row(s) affected)

(1 row(s) affected)
The values after the change are:
ProductID  LocationID  Quantity
-----  -----  -----
1          50          339

(1 row(s) affected)

TransactionID  ProductID  NetAdjustment  ModifiedDate
-----  -----  -----  -----
1            1           -7           2006-04-12 09:54:29.187
2            1           -7           2006-04-12 09:54:29.200

(1 row(s) affected)

```

Using Triggers for Custom Error Messages

We've already touched on this in some of our other examples, but remember that triggers can be handy for when you want control over the error message or number that gets passed out to your user or client application.

With a CHECK constraint for example, you're just going to get the standard 547 error along with its rather nondescript explanation. As often as not, this is less than helpful in terms of the user really figuring out what went wrong—indeed, your client application often doesn't have enough information to make an intelligent and helpful response on behalf of the user.

In short, sometimes you create triggers when there is already something that would give you the data integrity that you want but won't give you enough information to handle it.

Other Common Uses for Triggers

In addition to the straight data integrity uses, triggers have a number of other uses. Indeed, the possibilities are fairly limitless, but here are a few common examples:

- ❑ Updating summary information
- ❑ Feeding de-normalized tables for reporting
- ❑ Setting condition flags

Updating Summary Information

Sometimes we like to keep aggregate information around to help with reporting or to speed performance when checking conditions.

Take, for instance, the example of a customer's credit limit versus their current balance. The limit is a fairly static thing and is easily stored with the rest of the customer information. The current balance is another matter. We can always figure out the current balance by running a query to total all of the unpaid balances for any orders the customer has, but think about that for a moment. Let's say that you work for Sears, and you do literally millions of transactions every year. Now think about how your table is going to have many millions of records for your query to sort through and that you're going to be competing with many other transactions in order to run your query. Things would perform an awful lot better if we could just go to a single place to get that total—but how to maintain it?

We certainly could just make sure that we always use a stored procedure for adding and paying order records, and then have the sproc update the customer's current balance. But that would mean that we would have to be sure that every sproc that has a potential effect on the customer's balance would have the update code. If just one sproc leaves it out, then we have a major problem, and figuring out which sproc is the offending one is a hassle at best, and problematic at worst. By using a trigger, however, the updating of the customer balance becomes pretty easy.

We could maintain virtually any aggregation we want to keep track of. Keep in mind, however, that every trigger that you add increases the amount of work that has to be done to complete your transactions. That means that you are placing an additional burden on your system and increasing the chances that you will run into deadlock problems.

Feeding Data into De-normalized Tables for Reporting

I'm going to start right off by saying this isn't the way you should do things in most circumstances. Usually, this kind of data transfer should be handled as part of a batch process run at night or during non-peak hours for your system—depending on the nature of what you are moving, replication may also be an excellent answer. We will be discussing replication in detail in Chapter 20.

That being said, sometimes you need the data in your reporting tables to be right up-to-the-minute. The only real way to take care of this is to either modify all your sprocs and other access points into your system to update the reporting tables at the same time as they update the Online Transaction Processing (OLTP) tables (YUCK!), or to use triggers to propagate any updates to records.

What's nice about using this method to propagate data is that you are always certain to be up-to-the-minute on what's happening in the OLTP tables. That being said, it defeats a large part of the purpose of keeping separate reporting tables. While keeping the data in a de-normalized format can greatly improve query performance, one of its main goals, in most installations, is to clear reporting needs out of the main OLTP database and minimize concurrency issues. If all your OLTP updates still have to update information in your reporting tables, then all you've done is to move which database the actual deadlock or other concurrency issue is happening in. From the OLTP standpoint, you've added work without gaining any benefits.

The thing you have to weigh out here is whether you're going to gain enough performance in your reporting to make it worth the damage you're going to do to performance on your OLTP system.

Setting Condition Flags

This situation is typically used much as aggregation is—to maintain a flag as changes are made rather than having to look for a certain condition across a complete table. Lookup flags are one of the little things that, while they usually break the rules of normalization (you're not supposed to store data that can be derived elsewhere), can really boost system performance substantially.

For the example on this topic, let's assume that we maintain a variety of information on the products that we sell. Material Safety Data Sheets (MSDS), information on suppliers—imagine there can be an unlimited number of different documents that all provide some sort of information on our products. Now, further imagine that we have something more than the mere 504 products that are in the AdventureWorks database (it's not at all uncommon for businesses to have 50,000 or more different line items in their catalog). The number of possible informational records could get extremely high.

We want to be able to put a flag on our Customer Support screens that tell the order taker whether there is any additional information available for this product. If we were living by the rules of a normalized database, we would have to look in the `ProductDocument` table to see if it had any records that matched up with our `ProductID`.

Rather than do those lookups, we can just place a bit field in our `Products` table that is a yes/no indicator on whether other information is available. We would then put a trigger on the `ProductInformation` table that updates the bit flag in the `Products` table. If a record is inserted into `ProductInformation`, then we set the bit flag to `TRUE` for the corresponding product. When a `ProductInformation` record is deleted, we look to see whether it was the last one, and, if so, set the bit flag in the `Products` table back to `FALSE`.

We'll go for an ultra-quick example. First, we need to set up a bit by creating the bit flag field and `ProductDocument` table:

```
ALTER TABLE Production.Product
    ADD InformationFlag    bit    NOT NULL
    CONSTRAINT InformationFlagDefault
        DEFAULT 0 WITH VALUES
```

Then we need to fix the data in the table to allow for documentation we already have:

```
UPDATE p
SET p.InformationFlag = 1
FROM Production.Product p
```

Chapter 13

```
WHERE EXISTS
(
    SELECT 1
    FROM Production.ProductDocument pd
    WHERE pd.ProductID = p.ProductID
)
```

Then we're ready to add our trigger:

```
CREATE TRIGGER DocumentBelongsToProduct
    ON Production.ProductDocument
    FOR INSERT, DELETE
AS
    DECLARE @Count      int

    SELECT @Count = COUNT(*) FROM Inserted

    IF @Count > 0
        BEGIN
            UPDATE p
                SET p.InformationFlag = 1
                FROM Inserted i
                JOIN Production.Product p
                    ON i.ProductID = p.ProductID
        END

    IF @@ERROR != 0
        ROLLBACK TRAN

    SELECT @Count = COUNT(*) FROM Deleted
    IF @Count > 0
        BEGIN
            UPDATE p
                SET p.InformationFlag = 0
                FROM Inserted i
                RIGHT JOIN Production.Product p
                    ON i.ProductID = p.ProductID
                WHERE i.ProductID IS NULL
        END

    IF @@ERROR != 0
        ROLLBACK TRAN
```

And we're ready to test:

```
SELECT ProductID, InformationFlag
FROM Production.Product p
WHERE p.ProductID = 1

INSERT INTO Production.ProductDocument
    (ProductID, DocumentID)
VALUES
    (1, 1)
```

```
SELECT ProductID, InformationFlag
FROM Production.Product p
WHERE p.ProductID = 1
```

This yields us the proper update:

```
ProductID    InformationFlag
----- -----
1           0
(1 row(s) affected)

(1 row(s) affected)

(1 row(s) affected)
ProductID    InformationFlag
----- -----
1           1
(1 row(s) affected)
```

And the delete:

```
DELETE Production.ProductDocument
WHERE ProductID = 1
    AND DocumentID = 1

SELECT ProductID, InformationFlag
FROM Production.Product p
WHERE p.ProductID = 1
```

Again, this gets up the proper update:

```
ProductID    InformationFlag
----- -----
1           0
(1 row(s) affected)
```

Now we can find out whether there's product documentation right in the very same query with which we grab the base information on the product. We won't incur the overhead of the query to the ProductDocument table unless there really is something out there for us to retrieve.

Other Trigger Issues

You have most of it now, but if you're thinking you are finished with triggers, then think again. As I indicated early in the chapter, triggers create an awful lot to think about. The sections that follow attempt to point out some of the biggest issues you need to consider, plus provide some information on additional trigger features and possibilities.

Triggers Can Be Nested

A nested trigger is one that did not fire directly as a result of a statement that you issued but rather because of a statement that was issued by another trigger.

This can actually set off quite a chain of events—with one trigger causing another trigger to fire which, in turn, causes yet another trigger to fire, and so on. Just how deep the triggers can fire depends on:

- ❑ Whether nested triggers are turned on for your system (this is a system-wide, not database-level option; it is set using Management Studio or `sp_configure`, and defaults to on).
- ❑ Whether there is a limit of nesting to 32 levels deep.
- ❑ Whether a trigger has already been fired. A trigger can, by default, only be fired once per trigger transaction. Once fired, it will ignore any other calls as a result of activity that is part of the same trigger action. Once you move on to an entirely new statement (even within the same overall transaction), the process can start all over again.

In most circumstances, you actually want your triggers to nest (thus the default), but you need to think about what's going to happen if you get into a circle of triggers firing triggers. If it comes back around to the same table twice, then the trigger will not fire the second time, and something you think is important may not happen; for example, a data integrity violation may get through. It's also worth noting that, if you do a `ROLLBACK` anywhere in the nesting chain, then entire chain is rolled back. In other words, the entire nested trigger chain behaves as a transaction.

Triggers Can Be Recursive

What is a recursive trigger? A trigger is said to be recursive when something the trigger does eventually causes that same trigger to be fired. It may be directly (by an action query done to the table on which the trigger is set), or indirectly (through the nesting process).

Recursive triggers are rare. Indeed, by default, recursive triggers are turned off. This is, however, a way of dealing with the situation just described, where you are nesting triggers and you want the update to happen the second time around. Recursion, unlike nesting, is a database-level option and can be set using the `sp_dboption` system sproc.

The danger in recursive triggers is that you'll get into some form of unintended loop. As such, you'll need to make sure that you get some form of recursion check in place to stop the process if necessary.

Debugging Triggers

Debugging triggers is a hassle at best. Since you have something of a level of indirection (you write a statement that causes the trigger to fire, rather than explicitly firing it yourself), it always seems like you have to second guess what's going on.

You can utilize the same Visual Studio debugger we utilized last chapter—you just need to get tricky to do it. The trick? The trick is to create a stored procedure that will cause your trigger to fire, and then step into that stored procedure. You can then step your way right into the trigger.

When debugging with Visual Studio is a trial, use `PRINT` and `SELECT` statements to output your values in the triggers. Beyond telling you what your variables are doing along the way, they can also tip you off to recursion and, in some cases nesting, problems.

Nesting issues can be one of the biggest gotchas of trigger design. You will find it not at all uncommon to see situations where you execute a command and wind up with unexpected results because you didn't realize how many other triggers were, in turn, going to be fired. What's more, if the nested triggers perform updates to the initiating table, the trigger will not fire a second time — this creates data integrity problems in tables where you are certain that your trigger is correct in preventing them. It probably has the right code for the first firing, but it doesn't even run the second time around in a nested situation.

You can also make use of `SELECT @@NESTLEVEL` to show just how deep into a nesting situation you've got.

Keep in mind though, that `PRINT` and result set generating `SELECT` statements don't really have anywhere to send their data other than the screen (in Management Studio) or as an informational message (data access models). This is usually far more confusing than anything else is. As such, I highly recommend removing these statements once you've finished debugging, and before you go to production release.

Triggers Don't Get in the Way of Architecture Changes

This is a classic good news/bad news story.

Using triggers is positively great in terms of making it easy to make architecture changes. Indeed, I often use triggers for referential integrity early in the development cycle (when I'm more likely to be making lots of changes to the design of the database) and then change to DRI late in the cycle when I'm close to production.

When you want to drop a table and re-create it using DRI, you must first drop all of the constraints before dropping the table. This can create quite a maze in terms of dropping multiple constraints, making your changes, and then adding back the constraints again. It can be quite a wild ride trying to make sure that everything drops that is supposed to so that your changed scripts will run. Then it's just as wild a ride to make sure that you've got everything back on that needs to be. Triggers take care of all this because they don't care that anything has changed until they actually run.

There's the rub though — when they run. You see, it means that you may change architecture and break several triggers without even realizing that you've done it. It won't be until the first time that those triggers try to address the object(s) in question that you find the error of your ways. By that time, you may find difficulty in piecing together exactly what you did and why.

Both sides have their hassles — just keep the hassles in mind no matter which method you're employing.

Triggers Can Be Turned Off without Being Removed

Sometimes, just like with `CHECK` constraints, you want to turn off the integrity feature, so you can do something that will violate the constraint but still has a valid reason for happening (importation of data is probably the most common of these).

Another common reason for doing this is when you are performing some sort of bulk insert (importation again), but you are already 100 percent certain the data is valid. In this case, you may want to turn off the triggers to eliminate their overhead and speed up the insert process.

You can turn a trigger off and on by using an `ALTER TABLE` statement. The syntax looks like this:

```
ALTER TABLE <table name>
    <ENABLE|DISABLE> TRIGGER <ALL|<trigger name>>
```

As you might expect, my biggest words of caution in this area are, “Don’t forget to re-enable your triggers!”

One last thing. If you’re turning them off to do some form of mass importation of data, I highly recommend that you kick out all your users and go either to single-user mode, dbo-only mode, or both. This will make sure that no one sneaks in behind you while you have the triggers turned off.

Be sure to consider the ability to disable triggers when addressing security concerns. If you are counting on triggers to perform audits for you, but you are allowing the disabling of triggers (granted, they would have to have some degree of security already, but you still need to fully consider the possibilities), then you have a loop-hole in your auditing.

Trigger Firing Order

In long ago releases of SQL Server (7.0 and prior), we had no control over firing order. Indeed, you may recall me discussing how there was only one of any particular kind of trigger (`INSERT`, `UPDATE`, `DELETE`) prior to 7.0, so firing order was something of a moot point. Later releases of SQL Server provide a limited amount of control over which triggers go in what order. For any given table (not views, since firing order can only be specified for `AFTER` triggers and views accept only `INSTEAD OF` triggers), you can elect to have one (and only one) trigger fired first. Likewise, you may elect to have one (and only one) trigger fired last. All other triggers are considered to have no preference on firing order—that is, you have no guarantee in what order a trigger with a firing order of “none” will fire in other than that they will fire after the `FIRST` trigger (if there is one) is complete and before the `LAST` trigger (again, if there is one) begins (see Figure 13-3).

The creation of a trigger that is to be first or last works just the same as any other trigger. You state the firing order preference after the trigger has already been created by using a special system stored procedure, `sp_settriggerorder`.

The syntax of `sp_settriggerorder` looks like this:

```
sp_settriggerorder[@triggername =] '<trigger name>',
    [@order =] '{FIRST|LAST|NONE}',
    [@stmttype =] '{INSERT|UPDATE|DELETE}'
```

There can be only one trigger that is considered to be “first” for any particular action (`INSERT`, `UPDATE`, or `DELETE`). Likewise, there can be only one “last” trigger for any particular action. Any number of triggers can be considered to be “none”—that is, the number of triggers that don’t have a particular firing order is unlimited.

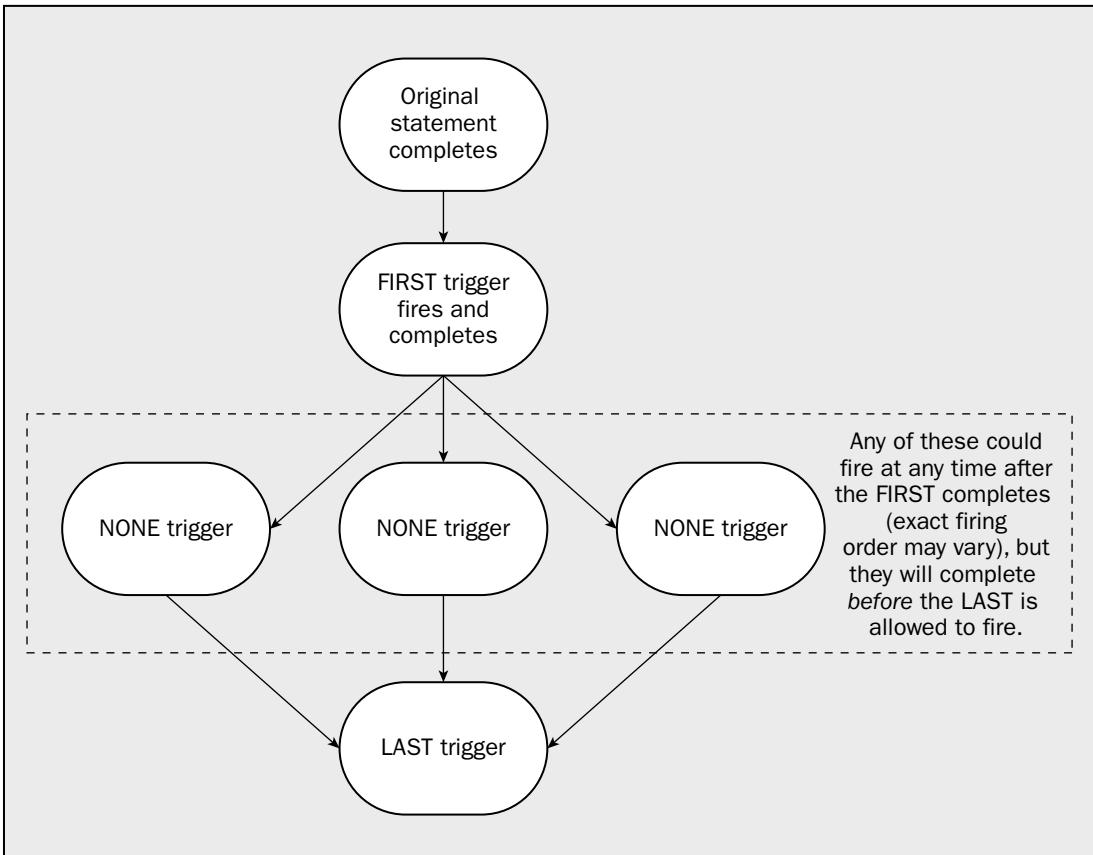


Figure 13-3

So, the question should be, "Why do I care what order they fire in?" Well, often you won't care at all. At other times, it can be important logic-wise or just a good performance idea. Let's consider what I mean in a bit more detail.

Controlling Firing Order for Logic Reasons

Why would you *need* to have one trigger fire before another? The most common reason would be that the first trigger lays some sort of foundation for, or otherwise validates, what will come afterwards. Under SQL Server 6.5 and earlier, we didn't have to think about this kind of thing much—we were only allowed one trigger of any particular type (`UPDATE`, `DELETE`, or `INSERT`) for a given table. This meant that having one thing happen before another wasn't really a problem. Because you combined all logic into one trigger, you just put the first thing that needed to happen first in the code and the last part last (no real rocket science there at all).

Version 7.0 came along and made things both better and worse than they were before. You were no longer forced to jam all of your logic into one trigger. This was really cool because it meant that you could physically separate parts of your trigger code that were logically different, which, in turn, both

made the code much easier to manage and allowed one part of the code to be disabled (remember that NO CHECK thing we did a few sections ago?) while other parts of the code continued to function. The downside was that if you went ahead and separated out your code that way, you lost the logical stepping order that the code had when it was in one trigger.

By gaining at least a rudimentary level of control over firing order, we now have something of the best of both worlds—we can logically separate our triggers but still maintain necessary order of precedence on what piece of code runs first or last.

Controlling Firing Order for Performance Reasons

On the performance front, a FIRST trigger is the only one that really has any big thing going for it. If you have multiple triggers, but only one of them is likely to generate a rollback (for example, it may be enforcing a complex data integrity rule that a constraint can't handle), you would want to consider making such a trigger a FIRST trigger. This makes certain that your most likely cause of a rollback is already complete before you invest any more activity in your transaction. The more you do before the rollback is detected, the more that will have to be rolled back. Get the highest possibility of that rollback happening determined before performing additional activity.

INSTEAD OF Triggers

While it can work against tables, the primary purpose of an INSTEAD OF trigger is usually to allow updates to views in places where it was previously not possible.

Essentially, an INSTEAD OF trigger is a block of code we can use as something of an interceptor for anything that anyone tries to do to our table or view. We can either elect to go ahead and do whatever the user requests or, if we choose, we can go so far as doing something that is entirely different.

As with FOR/AFTER triggers, INSTEAD OF triggers come in three different flavors—INSERT, UPDATE, and DELETE. Unlike FOR/AFTER triggers, however, you can only have one trigger per table or view for each of the different flavors (one each for INSERT, UPDATE, DELETE).

If we're going to explore these, we need to get some appropriate sample tables out there. To that end, let's take the follow four tables (you can change the script to use an existing database if you wish).

```
CREATE TABLE dbo.Customers
(
    CustomerID varchar(5) NOT NULL PRIMARY KEY ,
    Name varchar(40) NOT NULL
)

CREATE TABLE dbo.Orders
(
    OrderID int IDENTITY NOT NULL PRIMARY KEY,
    CustomerID varchar(5) NOT NULL
        REFERENCES Customers(CustomerID),
    OrderDate datetime NOT NULL
)

CREATE TABLE dbo.Products
```

```

(
    ProductID int IDENTITY NOT NULL PRIMARY KEY,
    Name varchar(40) NOT NULL,
    UnitPrice money NOT NULL
)

CREATE TABLE dbo.OrderItems
(
    OrderID int NOT NULL
        REFERENCES dbo.Orders(OrderID),
    ProductID int NOT NULL
        REFERENCES dbo.Products(ProductID),
    UnitPrice money NOT NULL,
    Quantity int NOT NULL
        CONSTRAINT PKOrderItem PRIMARY KEY CLUSTERED
            (OrderID, ProductID)
)

-- INSERT sample records
INSERT dbo.Customers
    VALUES ('ABCDE', 'Bob''s Pretty Good Garage')

INSERT dbo.Orders
    VALUES ('ABCDE', CURRENT_TIMESTAMP)

INSERT dbo.Products
    VALUES ('Widget', 5.55)

INSERT dbo.Products
    VALUES ('Thingamajig', 8.88)

INSERT dbo.OrderItems
    VALUES (1, 1, 5.55, 3)

```

We will use these tables for all three of the upcoming examples of `INSTEAD OF` triggers.

INSTEAD OF INSERT Triggers

The `INSTEAD OF INSERT` trigger allows us to examine the data that is about to go into our table or view, and decide what we want to do with it prior to the insert physically occurring. The typical use of this will usually be on a view — where getting to manipulate the data before the actual physical insert is attempted can mean the difference between the insert succeeding or failing.

Let's look at an example by creating an updateable view — specifically, one that will accept `INSERTS` where, before `INSTEAD OF INSERT` triggers, we wouldn't have been able to do it.

In this case, we'll create a view that demonstrates the update problem and then look at how to fix it. Let's take the case of showing some order line items, but with more full information about the customer and products (be sure you're using the database you created the sample tables in):

```

CREATE VIEW CustomerOrders_vw
WITH SCHEMABINDING
AS

```

Chapter 13

```
SELECT      o.SalesOrderID,
            o.OrderDate,
            od.ProductID,
            p.Name,
            od.OrderQty,
            od.UnitPrice,
            od.LineTotal
        FROM Sales.SalesOrderHeader AS o
    JOIN     Sales.SalesOrderDetail AS od
        ON o.SalesOrderID = od.SalesOrderID
    JOIN     Production.Product AS p
        ON od.ProductID = p.ProductID
```

The view is not fully updateable in its current state — how would SQL Server know which data went to which table? Sure, one could make a case for a straight update statement working, but we don't have the primary key for every table here. Even worse — what if we wanted to do an insert (which, as it happens, we do)?

The answer is something that SQL Server can't give you by itself — you need to provide more instructions on what you want to do in such complex situations. That's where `INSTEAD OF` triggers really shine.

Let's take a look at our example order:

```
SELECT *
    FROM CustomerOrders_vw
   WHERE OrderID = 1
```

This gets us back the one row we used to prime our sample:

```
Bob's Pretty Good Garage...1...2006-04-13 05:14:22.780...1...Widget...3...5.55
Now, just to prove it doesn't work, let's try to INSERT a new order item:
```

```
INSERT INTO CustomerOrders_vw
(
    OrderID,
    OrderDate,
    ProductID,
    Quantity,
    UnitPrice
)
VALUES
(
    1,
    '1998-04-06',
    2,
    10,
    6.00
)
```

As expected, it doesn't work:

```
Server: Msg 4405, Level 16, State 2, Line 1
View or function 'CustomerOrders_vw' is not updatable because the modification
affects multiple base tables.
```

It's time for us to take care of this with an `INSTEAD OF` trigger. What we need to do here is decide ahead of time what scenarios we want to handle (in this case, just the insert of new `OrderItem` records) and what we want to do about it.

We're going to treat any `INSERT` as an attempt to add a new order item. We're going to assume for this example that the customer already exists (if we wanted to get complex, we could break things up further) and that we have an `OrderID` available. Our trigger might look something like:

```
CREATE TRIGGER trCustomerOrderInsert ON CustomerOrders_vw
INSTEAD OF INSERT
AS
BEGIN
    -- Check to see whether the INSERT actually tried to feed us any rows.
    -- (A WHERE clause might have filtered everything out)
    IF (SELECT COUNT(*) FROM Inserted) > 0
    BEGIN
        INSERT INTO dbo.OrderItems
        SELECT i.OrderID,
               i.ProductID,
               i.UnitPrice,
               i.Quantity
        FROM Inserted AS i
        JOIN Orders AS o
        ON i.OrderID = o.OrderID
        -- If we have records in Inserted, but no records could join to
        -- the orders table, then there must not be a matching order
        IF @@ROWCOUNT = 0
            RAISERROR('No matching Orders. Cannot perform insert',10,1)
    END
END
```

So, let's try that insert again:

```
INSERT INTO CustomerOrders_vw
(
    OrderID,
    OrderDate,
    ProductID,
    Quantity,
    UnitPrice
)
VALUES
(
    (
        1,
        '1998-04-06',
        2,
        10,
        6.00
    )
)
```

We've explicitly addressed what table we're going to insert into, and so SQL Server is happy. We could easily extend this to address non-nullable columns that don't participate in the view if we needed to (the customer can't provide values to those columns because they are not in the view the customer is using).

INSTEAD OF UPDATE Triggers

We've now seen how `INSERT` statements against views can lead to ambiguous situations and also how to fix them with an `INSTEAD OF INSERT` trigger—but what about updates?

Even on the update side of things our statements can become ambiguous—if we update the `ProductName` in `CustomerOrders_vw`, does that mean we want to change the actual name on the product or does it mean that we want to change what product this line item is selling? The answer, of course, is that it depends on the situation. For one system, changing the `ProductName` might be the correct answer—for another system, changing the product sold might be the thing.

Much like `INSTEAD OF INSERT` triggers, `INSTEAD OF UPDATE` triggers give us the chance to trap what is coming in and address it explicitly. In our `ProductName` example, we could have chosen to do it either way. By default, SQL Server would update the name in the products table. We could, however, use an `INSTEAD OF UPDATE` trigger to trap it and explicitly look up the `ProductName` to find the `ProductID` if that is what the user intended. From there, we could generate an error if the provided `ProductID` did not make the one that went with the name.

INSTEAD OF DELETE Triggers

Okay, the last of our `INSTEAD OF` triggers and, most likely, the one that you'll run into the least often. As with the other two `INSTEAD OF` trigger types, these are used almost exclusively to allow views to delete data in one or more underlying tables.

So, continuing with our `CustomerOrders_vw` example, we'll add some delete functionality. This time, however, we're going to raise the complexity bar a bit. We want to delete all the rows for a given order, but if deleting those rows means that the order has no detail items left, then we also want to delete the order header.

We know from our last section (assuming you've been playing along) that we have two rows in Order 1 (the one we seeded when we built the table and the one we inserted in the `INSTEAD OF INSERT` example) but, before we start trying to delete things, let's build our trigger:

```
CREATE TRIGGER trCustomerOrderDelete ON CustomerOrders_vw
INSTEAD OF DELETE
AS
BEGIN

    -- Check to see whether the DELETE actually tried to feed us any rows
    -- (A WHERE clause might have filtered everything out)
    IF (SELECT COUNT(*) FROM Deleted) > 0
    BEGIN
        DELETE od
        FROM dbo.OrderItems AS oi
        JOIN Deleted AS d
            ON d.OrderID = oi.OrderID
            AND d.ProductID = oi.ProductID
        DELETE Orders
        FROM Orders AS o
    END
END
```

```

        JOIN Deleted AS d
          ON o.OrderID = d.OrderID
      LEFT JOIN OrderItem AS oi
        ON oi.OrderID = d.OrderID
       AND oi.ProductID = d.OrderID
      WHERE oi.OrderID IS NULL
    END

END

```

And now we're ready to test. We'll start off by deleting just a single row from our `CustomerOrders_vw` view:

```

DELETE CustomerOrders_vw
WHERE OrderID = 1
  AND ProductID = 2

```

We're ready to run our select again:

```

SELECT ProductID, UnitPrice, Quantity
FROM CustomerOrders_vw
WHERE OrderID = 1

```

Sure enough, the row that we first inserted in our `INSTEAD OF INSERT` section is now gone:

ProductID	UnitPrice	Quantity
1	5.55	3

(1 row(s) affected)

So, our deleting of individual detail lines is working just fine. Now let's get a bit more cavalier and delete the entire order:

```

DELETE CustomerOrders_vw
WHERE OrderID = 1

```

To really check that this worked okay, we need to go all the way to our `Orders` table:

```

SELECT * FROM Orders WHERE OrderID = 1

```

Sure enough—the order has been removed.

While we don't have to think about individual columns with `INSTEAD OF DELETE` triggers (you delete by row, not by column), we do need to be aware of what referential integrity actions exist on any table (not view) that we are defining an `INSTEAD OF DELETE` trigger for. Just like `INSTEAD OF UPDATE` triggers, `INSTEAD OF DELETE` triggers are not allowed on tables that have referential integrity actions.

IF UPDATE() and COLUMNS_UPDATED()

In an UPDATE trigger, we can often limit the amount of code that actually executes within the trigger by checking to see whether the column(s) we are interested in are the ones that have been changed. To do this, we make use of the `UPDATE()` or `COLUMNS_UPDATED()` functions. Let's look at each.

The `UPDATE()` Function

The `UPDATE()` function has relevance only within the scope of a trigger. Its sole purpose in life is to provide a Boolean response (true/false) to whether a particular column has been updated or not. You can use this function to decide whether a particular block of code needs to run or not—for example, if that code is only relevant when a particular column is updated.

Let's run a quick example of this by modifying one of our earlier triggers:

```
ALTER TRIGGER Production.ProductAudit
    ON Production.ProductInventory
    FOR INSERT, UPDATE, DELETE
AS
IF UPDATE(Quantity)
BEGIN
    INSERT INTO Production.InventoryAudit
    (ProductID, NetAdjustment)
        SELECT COALESCE(i.ProductID, d.ProductID),
               ISNULL(i.Quantity, 0) - ISNULL(d.Quantity, 0) AS NetAdjustment
        FROM Inserted i
    FULL JOIN Deleted d
        ON i.ProductID = d.ProductID
        AND i.LocationID = d.LocationID
    WHERE ISNULL(i.Quantity, 0) - ISNULL(d.Quantity, 0) != 0
END
```

With this change, we will now limit the rest of the code to run only when the `Quantity` column (the one we care about) has been changed. The user can change the value of any other column, and we don't care. This means that we'll be executing fewer lines of code and, therefore, this trigger will perform slightly better than our previous version.

The `COLUMNS_UPDATED()` Function

This one works somewhat differently from `UPDATE()` but has the same general purpose. What `COLUMNS_UPDATED()` gives us is the ability to check multiple columns at one time. In order to do this, the function uses a bit mask that relates individual bits in one or more bytes of varbinary data to individual columns in the table. It ends up looking something like Figure 13-4.

In this case, our single byte of data is telling us that the second, third, and sixth columns were updated—the rest were not.

In the event that there are more than eight columns, SQL Server just adds another byte on the right-hand side and keeps counting (see Figure 13-5).

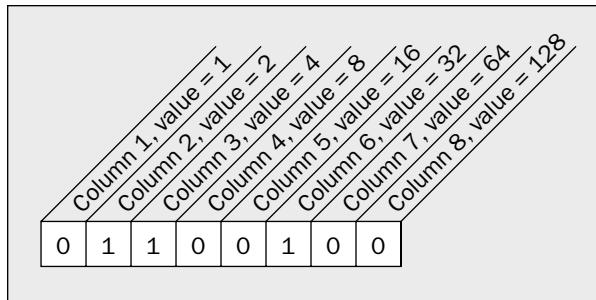


Figure 13-4

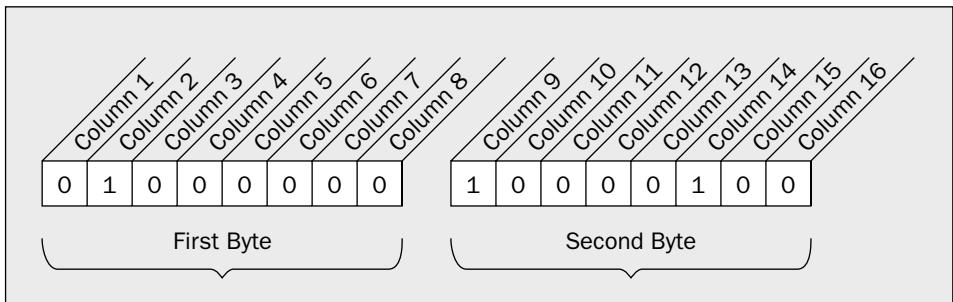


Figure 13-5

This time the second, ninth, and fourteenth columns were updated.

I can hear you out there: “Gee, that’s nice—but how do I make any use of this?” Well, to answer that, we have to get into the world of Boolean algebra.

Making use of this information means that you need to add up the binary value of all the bytes, considering the leftmost digit to be the least significant. So, if you want your comparison to take into account 2, 5, and 7, then you need to add the binary value of each bit: $2 + 16 + 64$. Then you need to compare the sum of the binary values of your columns to the bit mask by using bitwise operators:

- | Represents bitwise OR
- & Represents bitwise AND
- ^ Represents bitwise Exclusive OR

As I read back over what I’ve just written, I realize that it is correct, but about as clear as mud, so let’s look a little closer at what I mean with a couple of examples.

Imagine that we updated a table that contained five columns. If we updated the first, third, and fifth columns, the bit mask used by `COLUMNS_UPDATED` would contain 10101000, from $1 + 4 + 16 = 21$. We could use:

- ❑ `COLUMNS_UPDATED() > 0` to find out if any column was updated
- ❑ `COLUMNS_UPDATED() ^ 21 = 0` to find out if *all* of the columns specified (in this case 1, 3, and 5) were updated and nothing else was
- ❑ `COLUMNS_UPDATED() & 21 = 21` to find out if all of the columns specified were updated, but the state of other columns doesn't matter
- ❑ `COLUMNS_UPDATED() | 21 != 21` to find out if any column *other* than those we're interested in was updated

Understand that this is tough stuff—Boolean math is not exactly the easiest of concepts to grasp for most people, so check things carefully and TEST, TEST, TEST!

Performance Considerations

I've seen what appear almost like holy wars happen over the pros and cons, evil and good, and light and dark of triggers. The worst of it tends to come from purists—people who love the theory, and that's all they want to deal with, or people that have figured out how flexible triggers are and want to use them for seemingly everything.

My two bits worth on this is, as I stated early in the chapter, use them when they are the right things to use. If that sounds sort of noncommittal and ambiguous—good! Programming is rarely black and white, and databases are almost never that way. I will, however, point out some facts for you to think about.

Triggers Are Reactive Rather Than Proactive

What I mean here is that triggers happen after the fact. By the time that your trigger fires, the entire query has run and your transaction has been logged (but not committed and only to the point of the statement that fired your trigger). This means that, if the trigger needs to roll things back, it has to undo what is potentially a ton of work that's already been done. *Slow!* Keep this knowledge in balance though. How big an impact this adds up to depends strongly on how big your query is.

"So what?" you say. Well, compare this to the notion of constraints, which are proactive—that is, they happen before your statement is really executed. That means that they prevent things that will eventually fail from happening before the majority of the work has been done. This will usually mean that they will run at least slightly faster—much faster on more complex queries. Note that this extra speed really only shows itself to any significant extent when a rollback occurs.

What's the end analysis here? Well, if you're dealing with very few rollbacks, and/or the complexity and runtime of the statements affected are low, then there probably isn't much of a difference between triggers and constraints. There's some, but probably not much. If however, the number of rollbacks is unpredictable or if you know it's going to be high, you'll want to stick with constraints if you can (and frankly, I suggest sticking with constraints unless you have a very specific reason not to).

Triggers Don't Have Concurrency Issues with the Process That Fires Them

You may have noticed throughout this chapter that we often make use of the `ROLLBACK` statement, even though we don't issue a `BEGIN TRAN`. That's because a trigger is always implicitly part of the same transaction as the statement that caused the trigger to fire.

If the firing statement was not part of an explicit transaction (one where there was a `BEGIN TRAN`), then it would still be part of its own one-statement transaction. In either case, a `ROLLBACK TRAN` issued inside the trigger will still roll back the entire transaction.

Another upshot of this part-of-the-same-transaction business is that triggers inherit the locks already open on the transaction they are part of. This means that we don't have to do anything special to make sure that we don't bump into the locks created by the other statements in the transaction. We have free access within the scope of the transaction, and we see the database based on the modifications already placed by previous statements within the transaction.

Keep It Short and Sweet

I feel like I'm stating the obvious here, but it's for a good reason.

I can't tell you how often I see bloated, stupid code in sprocs and triggers. I don't know whether it's that people get in a hurry, or if they just think that the medium they are using is fast anyway, so it won't matter.

Remember that a trigger is part of the same transaction as the statement in which it is called. This means the statement is not complete until your trigger is complete. Think about it—if you write long-running code in your trigger, this means that every piece of code that you create that causes that trigger to fire will, in turn, be long running. This can really cause heartache in terms of trying to figure out why your code is taking so long to run. You write what appears to be a very efficient sproc, but it performs terribly. You may spend weeks and yet never figure out that your sproc is fine—it just fires a trigger that isn't.

Don't Forget Triggers When Choosing Indexes

Another common mistake. You look through all your sprocs and views figuring out what the best mix of indexes is—and totally forget that you have significant code running in your triggers.

This is the same notion as the "Short and Sweet" section—long-running queries make for long running statements which, in turn, lead to long running everything. Don't forget your triggers when you optimize!

Try Not to Roll Back within Triggers

This one's hard since rollbacks are so often a major part of what you want to accomplish with your triggers.

Just remember that AFTER triggers (which are far and away the most common type of trigger) happen after most of the work is already done—that means a rollback is expensive. This is where DRI picks up

almost all of its performance advantage. If you are using many ROLLBACK TRAN statements in your triggers, then make sure that you pre-process looking for errors before you execute the statement that fires the trigger. That is, because SQL Server can't be proactive in this situation, be proactive for it. Test for errors beforehand rather than waiting for the rollback.

Dropping Triggers

Dropping triggers is as easy as it has been for almost everything else this far:

```
DROP TRIGGER <trigger name>
```

And it's gone.

Debugging Triggers

If you try to navigate to the debugger for triggers the way that you navigate to the debugger for sprocs (see Chapter 11 for that) or functions, then you're in for a rude awakening—you won't find it. Because trigger debugging is such a pain, and I'm not very good at taking "no" for an answer, I decided to make the debugger work for me—it isn't pretty, but it works.

Basically, what we're going to do is create a wrapper procedure to fire off the trigger we want to debug. Essentially, it's a sproc whose sole purpose in life is to give us a way to fire off a statement that will let us step into our trigger with the debugger.

For example purposes, I'm going to take a piece of the last bit of test code that we used in this chapter and just place it into a sproc, so I can watch the debugger run through it line by line:

```
ALTER PROC spTestTriggerDebugging
AS
BEGIN
    -- This one should work
    UPDATE Products
    SET UnitsInStock = UnitsInStock - 1
    WHERE ProductID = 6;

    -- This one shouldn't
    UPDATE Products
    SET UnitsInStock = UnitsInStock - 12
    WHERE ProductID = 26;
END
```

Now I just navigate to this sproc in the Server Explorer, right-click on it, and select Step Into (see Figure 13-6).

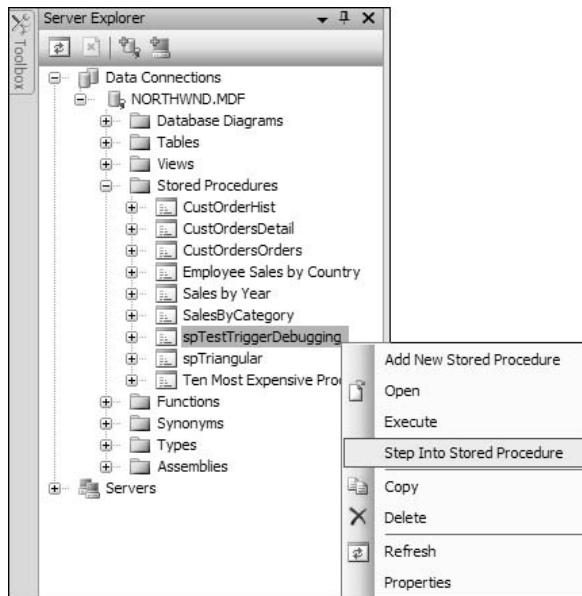


Figure 13-6

Click `Execute` at the dialog (we don't need to enter any parameters). At first, you'll just be in the debugger at the beginning of the sproc, but “step into” the lines that cause your trigger to fire, and you'll get a nice surprise (see Figure 13-7)!

From here, there's no real rocket science in using the debugger—it works pretty much as it did when we looked at it with sprocs. The trigger even becomes an active part of your call stack.

Keep this in mind when you run into those sticky trigger debugging situations. It's a pain that it has to be done this way, but it's better than nothing.

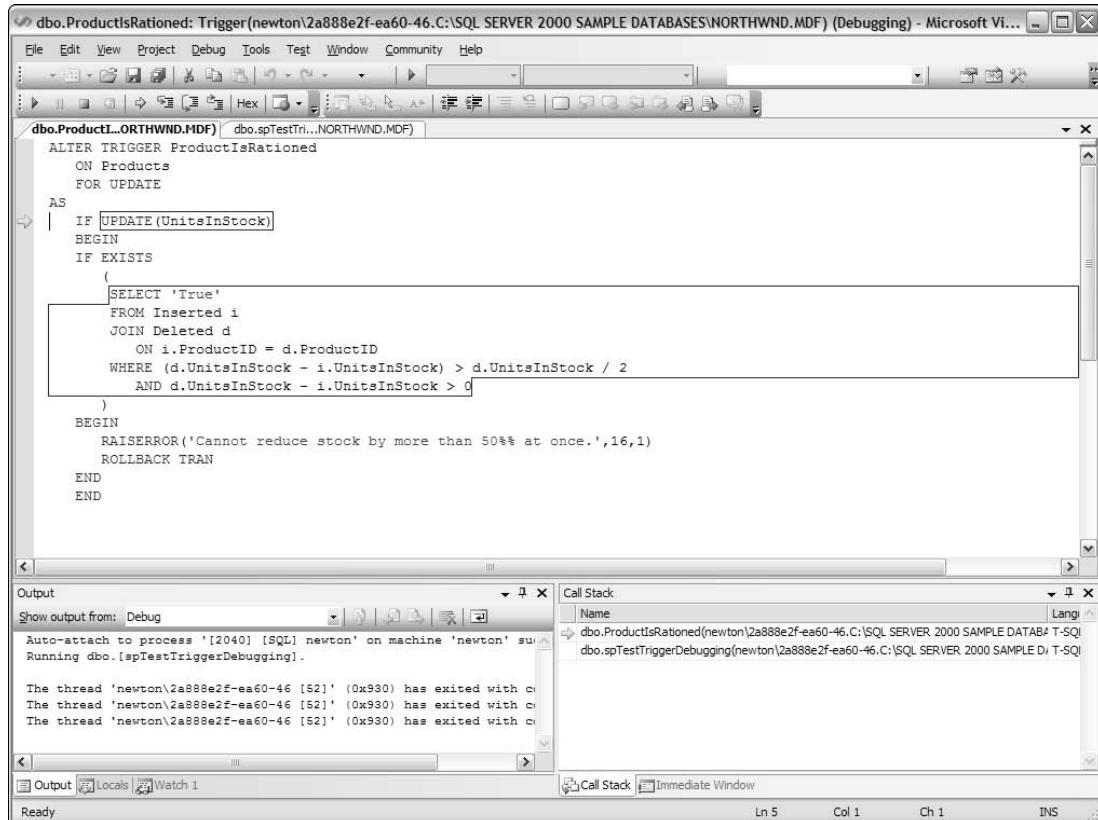


Figure 13-7

Summary

Triggers are an extremely powerful tool that can add tremendous flexibility to both your data integrity and the overall operation of your system. That being said, they are not something to take lightly. Triggers can greatly enhance the performance of your system if you use them for proper summarization of data, but they can also be the bane of your existence. They can be very difficult to debug (even now that we have the debugger), and a poorly written trigger affects not only the trigger itself but any statement that causes that trigger to fire.

14

Nothing But NET!

In terms of where to locate a subject in my books, this subject has to be the hardest one to place I've ever had to deal with. At issue is how fundamental the addition of .NET assemblies is to SQL Server programming in the SQL Server 2005 era versus how much it is shared between the various programming topics and how much of it happens in its own world. Oh well, obviously I made a choice, and that choice was to hold the introduction of major .NET elements until we had all the major SQL Server-specific programming elements covered. So, that done, here we go.

.NET and all things related to it were just on the horizon when SQL Server 2000 came out. It's been a long wait since early 2000 (yes, even before SQL Server 2000 was out) when I first heard that T-SQL was finally going to accept code from non T-SQL languages. The story just got better and better as we heard that complex user-defined data types would be supported, and T-SQL itself would become a .NET language with associated error handling. And so it is here, and the days of the old claustrophobic TSQL are gone, and we have a wide world of possibilities available to us.

In this chapter, we're going to take a look at some of the major elements that .NET has brought to SQL Server 2005. We'll see such things utilizing .NET as:

- Creating basic assemblies — including non T-SQL based stored procedures, functions, and triggers
- Defining aggregate functions (something T-SQL user defined functions can't do)
- Complex data types
- External calls (and with it, some security considerations)

.NET is something of a wide-ranging topic that will delve into many different areas we've already touched on in this book and take them even farther, so, with that said, let's get going!

Note that several of the examples in this chapter utilize the existing Microsoft Sample set. You must install the sample scripts during SQL Server installation or download the SQL Server .NET development SDK to access these samples. In addition, there is a significant reliance on Visual Studio .NET (2005 is used in the examples).

Assemblies 101

All the new .NET functionality is surrounded by this new (to SQL Server anyway) term *assembly*. So, a reasonable question might be: "What exactly is an assembly?" An assembly is a DLL that has been created using managed code (what .NET language does not matter). The assembly may have been built using Visual Studio .NET or some other development environment, but the .NET Framework SDK also provides a command-line compiler for those of you who do not have Visual Studio available.

Not all custom attributes or .NET Framework APIs are legal for assemblies used in SQL Server. You can consult Books Online for a full list, but, in general, anything that supports windowing is not allowed, nor is anything marked UNSAFE, unless your assembly is to be granted access at an UNSAFE level.

Compiling an Assembly

Use of .NET assemblies requires that you enable the Common Language Runtime (CLR) in SQL Server —which is disabled by default. You can enable the CLR by executing the following in the Management Studio:

```
sp_configure 'clr enabled', 1  
GO
```

```
RECONFIGURE
```

There really isn't that much to this beyond compiling a normal DLL. The real key points to compiling a DLL that is going to be utilized as a SQL Server .NET assembly are:

- ❑ You cannot reference any other assemblies that include functions related to windowing (dialogs, etc.).
- ❑ How the assembly is marked (safe, external access, unsafe) will make a large difference to whether or not the assembly is allowed to execute any functions.

From there, most things are not all that different from any other DLL you might create to make a set of classes available. You can either compile the project using Visual Studio (if you have it), or you can use the compiler that is included in the .NET SDK.

Let's go ahead and work through a relatively simple example for an assembly we'll use as a stored procedure example a little later in the chapter.

Create a new SQL Server project in Visual Studio using C# (you can translate this to VB if you wish) called `ExampleProc`. You'll find the SQL Server project type under the "Database" project group. When it comes up, cancel out of any database instance dialogs you get, and choose Class Library project in Visual Studio.

The actual project type you start with does not really matter other than what references it starts with. While this example suggests starting with a SQL Server project, you could just as easily start with a simple class project.

Now add a new stored procedure by right-clicking the project and selecting Add>Stored Procedure... as shown in Figure 14-1:

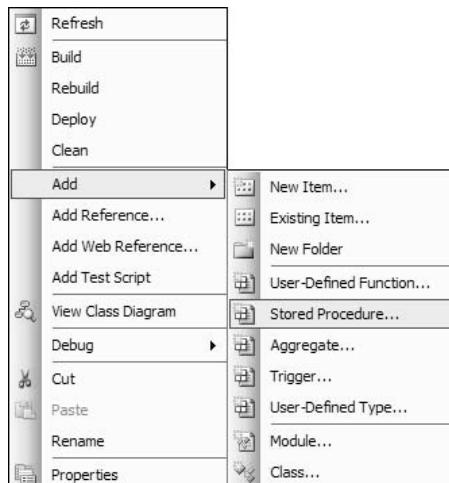


Figure 14-1

In this new stored proc, we need set a few references—some of them Visual Studio will have already done for you:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
```

Chapter 14

And then we're ready to get down to writing some real code. Code you want to put in a .NET assembly is implemented through a public class. I've chosen to call my class `StoredProcedures`, but, really, I could have called it most anything. I'm then ready to add my method declaration:

```
public partial class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void ExampleSP(out int outval)
    {
```

The method has, like the class, been declared as `public`. I can also have `private` classes if I so choose (they would, of course, need to be supporting methods, as they would not be exposed externally). The `void` indicates that I do not intend to supply a return value (when we run it in SQL Server, it will always return the default of zero). I could, however, declare it as returning type `int` and supply a value as appropriate in my code (most likely zero for no error, and non-zero if there was an error).

Notice also the `Microsoft.SqlServer.Server.SqlProcedure` directive. This is completely optional and is utilized by Visual Studio's deployment tool to know that the following method is a stored procedure. I left it in there primarily to show you that it is there (we're going to manually deploy the proc rather than use the deployment functionality of Visual Studio).

From there, we're ready to grab a reference to a connection. Pay particular attention to this one, as it is different from what you will see in typical .NET database connections. Everything about it is the same as a typical connection, except for the connect string — with that, we are using a special syntax that indicates that we want to utilize the same connection that called the stored procedure. The beauty of this is that we can assume a login context and do not need to explicitly provide a server or even a username.

```
// This causes the connection to use the existing connection context
// that the stored procedure is operating in. We could also create a
// completely new connection to fetch data from external sources.
using (SqlConnection cn = new SqlConnection("context connection=true"))
{
    cn.Open();
```

So, now we have a connection. Technically, we didn't really open a new connection (remember, we're utilizing the one that was already opened to call this stored procedure). Instead, we're really creating a new reference to the existing connection.

We're now ready to create a command object. There really isn't anything mysterious about this at all — it is created using a fairly typical syntax (indeed, you can create the command object in any of the typical ways you would if you were doing this from a typical .NET client). I've chosen to define the `CommandText` and `connection` properties as part of my object declaration.

```
// set up a simple command that is going to return two columns.
SqlCommand cmd = new SqlCommand("SELECT @@SERVERNAME, @@SPID", cn);
```

And then we're ready to execute the command. This is one of the spots I'm utilizing something assembly specific, as I am not only executing the command, but also I'm explicitly saying to go ahead and send it to the client.

Unlike a T-SQL based stored procedure, queries you execute are not defaulted as going to the client. Instead, the assumption is that you want to work with the result set locally. The result is that you need to explicitly issue a command to send anything out to the client.

The object that does this is the `.Pipe` object within the `SqlContext`.

```
// The following actually sends the row for the select.  
// It could have been multiple rows, and that would be fine too.  
SqlContext.Pipe.ExecuteAndSend(cmd);
```

Last, but not least, I'm populating my output variable. In this procedure, I haven't really done anything special with it, but I'm tossing something into it just so we can see that it really does work.

```
// Set the output value to something. It could have been anything  
// including some form of computed value, but we're just showing  
// that we can output some value for now.  
outval = 12345;  
}  
}  
};
```

Now simply build your project, and you have your first assembly ready to be uploaded to your SQL Server. Later, we'll take a look at how to define the assembly for use as a stored procedure.

Uploading Your Assembly to SQL Server

That's right—I used the term *upload*. When you "create" an assembly in SQL Server, you're both creating a copy of the DLL within SQL Server as well as something of a handle that defines the assembly and the permissions associated with it.

```
CREATE ASSEMBLY <assembly name>  
[ AUTHORIZATION <owner name> ]  
FROM { <client assembly specifier> | <assembly bits> [ ,...n ] }  
[ WITH PERMISSION_SET = { SAFE | EXTERNAL_ACCESS | UNSAFE } ]  
[ ; ]
```

The CREATE portion of things adheres to the standard `CREATE <object type> <object name>` notion that we've seen throughout SQL. From there, we have a few different things to digest:

Option	Description
AUTHORIZATION	The authorization is the name of the user this assembly will be considered to belong to. If this parameter is not supplied, then the current user is assumed to be the owner. You can use this to alias to a user with appropriate network access to execute any file actions defined by the assembly.
FROM	The fully qualified path to the physical DLL file. This can be a local file or a UNC path. You can, if you so choose, provide the actual byte sequence to build the file right on the line in the place of a file (I have to admit I've never tried that one).
WITH PERMISSION_SET	You have three options for this. SAFE is the default and implies that the object is not utilizing anything that requires access outside of the SQL Server process (no file access, no external database access). EXTERNAL_ACCESS indicates that your assembly requires access outside of the SQL Server process (to files in the operating system or UNC path, or perhaps an external ODBC/OLEDB connection). UNSAFE implies that your assembly gives your assembly free access to the SQL Server memory space without regard to the CLR managed code facilities. This means your assembly has the potential to destabilize your SQL Server through improper access.

So, with all this in mind, we're ready to upload our assembly:

```
USE AdventureWorks

CREATE ASSEMBLY ExampleProc
    FROM '<solution path>\ExampleProc\bin\Debug\ExampleProc.dll'
```

Assuming that you have the path to your compiled DLL correct, you shouldn't see any messages except for the typical "Command(s) completed successfully" message, and, with that, you are ready to create the SQL Server stored procedure that will reference this assembly.

Creating Your Assembly-Based Stored Procedure

All right then; all the tough stuff is done (if you're looking for how to actually create the assembly that is the code for the stored proc, take a look back two sections). We have a compiled assembly, and we have uploaded it to SQL Server—it's time to put it to use.

To do this, we use the same `CREATE PROCEDURE` command we learned back in Chapter 11. The difference is that, in the place of T-SQL code, we reference our assembly. For review, the syntax looks like this:

```

CREATE PROCEDURE|PROC <sproc name>
    [<parameter name> [<schema>.]<data type> [VARYING] [= <default value>] [OUT
[PUT]] [,,
    <parameter name> [<schema>.]<data type> [VARYING] [= <default value>]
[OUT[PUT]] [,,
        ...
        ...
        ]]
[WITH
    RECOMPILE| ENCRYPTION | [EXECUTE AS { CALLER|SELF|OWNER|<'user name'>} ]
[FOR REPLICATION]
AS
<code> | EXTERNAL NAME <assembly name>.<assembly class>

```

Some of this we can ignore when doing assemblies. The key things are:

- We use the EXTERNAL NAME option instead of the <code> section we used in our main chapter on stored procedures. The EXTERNAL NAME is done in a format of
`<assembly name>.<class name>.<method name>`
- We still need to define all parameters (in an order that matches the order our assembly method).

Now let's apply that to the assembly we created in the previous section:

```

CREATE PROC spCLRExample
(
    @outval int = NULL OUTPUT
)
AS EXTERNAL NAME ExampleProc.StoredProcedures.ExampleSP

```

It is not until this point that we actually have a stored procedure that utilizes our assembly. Notice that the stored procedure name is completely different from the name of the method that implements the stored procedure.

Now go ahead and make a test call to our new stored procedure:

```

DECLARE @OutVal int
EXEC spCLRExample @OutVal OUTPUT

SELECT @OutVal

```

We're declaring a holding variable to receive the results from our output variable. We then execute the procedure and select the value for our holding variable. When you check the results, however, you'll find not one result set—but two:

 NEWTON 52 -----

(1 row(s) affected)

```
12345
```

```
(1 row(s) affected)
```

The first of these is the result set we sent down the `SqlContext.Pipe`. When we executed the `cmd` object, the results were directed down the pipe, and so the client received them. The second result set represents the `SELECT` of our `@OutVal` variable.

This is, of course, a pretty simplistic example, but realize the possibilities here. The connection could have been, assuming we were set to `EXTERNAL_ACCESS`, to any datasource. We could access files and even Web services. We can add in complex libraries to perform things like regular expressions (careful on performance considerations there).

We will look at adding some of these kinds of things in as we explore more types of assembly-based SQL programming.

Creating Scalar User-Defined Functions from Assemblies

Scalar functions are not much different from stored procedures. Indeed, for the most part, they have the very same differences that the T-SQL versions. Much as with stored procedures, we utilize the same core `CREATE` syntax that we used in the T-SQL user-defined functions (UDFs) we created back in Chapter 11.

```
CREATE FUNCTION [<schema name>.]<function name>
    ( [ <@parameter name> [AS] [<schema name>.]<scalar data type> [ = <default
value>]
    [ ,...n ] ] )
RETURNS {<scalar type>|TABLE [(<Table Definition>)]}
    [ WITH [ENCRYPTION] | [SCHEMABINDING] |
        [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ] | [EXECUTE AS {
CALLER|SELF|OWNER|<'user name'>} ]
    ]
[AS] { EXTERNAL NAME <external method> |
BEGIN
    [<function statements>]
    {RETURN <type as defined in RETURNS clause>|RETURN (<SELECT statement>)}
END }[;]
```

There are one or two new things once you get inside of the .NET code. Of particular note is that there are some properties that you can set for your function. Among those, probably the most significant is that you must indicate if the function is deterministic (the default is nondeterministic). We'll see an example of this in use shortly.

For the example this time, start a new SQL Server project in Visual Studio, but instead of adding a stored procedure as we did in our original assembly example, add a user-defined function.

SQL Server starts you out with a simple template:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

public partial class UserDefinedFunctions
{
    [Microsoft.SqlServer.Server.SqlFunction]
    public static SqlString ExampleUDF()
    {
        // Put your code here
        return new SqlString("Hello");
    }
}
```

This is actually a workable template “as is.” You could compile it and add it to SQL Server as an assembly, and it would work right out of the box (though just getting back the string “Hello” is probably not all that useful).

We’ll replace that, but this time we’re going to write something that is still amazingly simple. In the end, we’ll see that, while simple, it is much more powerful than our stored procedure example.

In previous books, I have often lamented the issues with trying to validate e-mail fields in tables. E-mail, when you think about it, is a strongly typed notion, but one that SQL Server has only been able to perform minimal validation of. What we need are regular expressions.

We could approach this issue by writing a validation function and implementing it as a user-defined data type. This approach would have some validity, but has a problem—the rules for validating e-mails change on occasion (such as when new country codes or added, or when the .biz and .info top domains were added several years ago). Instead, we’re going to implement simple regex functionality and then utilize a call to that function in a constraint.

We can do this with relatively minimal changes to the function template that SQL Server gave us. First, we can get rid of a few library declarations, since we won’t be really working with SQL Server data to speak of, and add back two of our own. We wind up with just three using declarations:

```
using System;
using System.Text.RegularExpressions;
using Microsoft.SqlServer.Server;
```

We’re then ready to implement out function with very few changes:

```
[SqlFunction(IsDeterministic = true, IsPrecise = true)]
public static bool RegEx.IsMatch(string pattern, string matchString)
{
    Regex reg = new Regex(pattern.TrimEnd(null));
    return reg.Match(matchString.TrimEnd(null)).Success;
}
```

Chapter 14

Oh sure, we completely replaced the old function, but not by much. Indeed, we only have two more lines of code—and that's including the determinism declaration!

I'm not going to review it much here, but take a look back at Chapter 11 if you need to be reminded of how determinism works. The key thing is that, given the same inputs, the function must always yield the same outputs.

Go ahead and compile this, and we're ready to upload the assembly:

```
USE AdventureWorks

CREATE ASSEMBLY ExampleUDF
FROM '<solution path>\ExampleUDF\bin\Debug\ExampleUDF.dll'
```

And then create the function reference:

```
CREATE FUNCTION fCLRExample
(
    @Pattern nvarchar(max),
    @MatchString nvarchar(max)
)
RETURNS BIT
AS EXTERNAL NAME ExampleUDF.UserDefinedFunctions.RegExIsMatch
```

Notice the use of the `nvarchar` type instead of `varchar`. The string data type is a Unicode data type, and our function data type declaration needs to match.

This done, we're ready to test things out a bit:

```
SELECT ContactID, FirstName, LastName, EmailAddress, Phone
FROM Person.Contact
WHERE dbo.fCLRExample('[a-zA-Z0-9_\-]+@[a-zA-Z0-9_\-]+\.(com|org|edu|mil|net)', 
                      EmailAddress) = 1
```

If you have the default data, then this will actually return every row in the table since they all are `adventure-works.com` addresses. So, let's try a simple test to show what works versus what doesn't:

```
DECLARE @GoodTestMail varchar(100),
        @BadTestMail varchar(100)

SET @GoodTestMail = 'robv@professionalsql.com'
SET @BadTestMail = 'misc. text'

SELECT dbo.fCLRExample('[a-zA-Z0-9_\-]+@[a-zA-Z0-9_\-]+\.(com|org|edu|nz|au)', 
                      @GoodTestMail) AS ShouldBe1
SELECT dbo.fCLRExample('[a-zA-Z0-9_\-]+@[a-zA-Z0-9_\-]+\.(com|org|edu|nz|au)', 
                      @BadTestMail) AS ShouldBe0
```

For the sake of brevity, I have not built the full e-mail regex string here. It would need to include all of the valid country code top domains such as au, ca, uk, and us. There are a couple hundred of these, so it wouldn't fit all that well. That said, the basic construct is just fine, you can tweak it to meet your particular needs.

This gets us back what we would expect:

```
ShouldBe1
-----
1

(1 row(s) affected)

ShouldBe0
-----
0

(1 row(s) affected)
```

But let's not stop there. We have this nice function, let's apply it a little further by actually applying it as a constraint to the table.

```
ALTER TABLE Person.Contact
ADD CONSTRAINT ExampleFunction
CHECK (dbo.fCLRExample(' [a-zA-Z0-9_\-]+@[a-zA-Z0-9_\-]+\.\+(\com|\org|\edu|\nz|\au|',
EmailAddress) = 1)
```

Now we try to update a row to insert some bad data into our column, and it will be rejected:

```
UPDATE Person.Contact
SET EmailAddress = 'blah blah'
WHERE ContactID = 1
```

And SQL Server tells you the equivalent of "no way":

```
Msg 547, Level 16, State 0, Line 1
The UPDATE statement conflicted with the CHECK constraint "ExampleFunction". The
conflict occurred in database "AdventureWorks", table "Person.Contact", column
'EmailAddress'.
The statement has been terminated.
```

Creating Table-Valued Functions

Functions are going to continue to be a focus of this chapter for a bit. Why? Well, functions have a few more twists to them than some of the other assembly uses.

In this section, we're going to focus in on table valued functions. They are among the more complex things we need to deal with in this chapter, but, as they are in the T-SQL version, they were also among the more powerful. The uses range far and wide. They can be as simple as special treatment of a column in something you could have otherwise done in a typical T-SQL function or can be as complex as a merge of data from several disparate and external datasources.

Chapter 14

Go ahead and start another Visual Studio project called `ExampleTVF`, using the SQL Server project template—also add a new user-defined function. We’re going to be demonstrating accessing the file system this time, so add the following references:

```
using System;
using System.IO;
using System.Collections;
using Microsoft.SqlServer.Server;
```

Before we get too much into the code, let’s look ahead a bit at some of the things a table-valued function—or TVF—requires:

The entry function must implement the `IEnumerable` interface. This is a special, widely used, interface in .NET that essentially allows for the iteration over some form of row (be it in an array, collection, table, or whatever). As part of this concept, we must also define the `FillRowMethodName` property. The function specified in this special property will be implicitly called by SQL Server every time SQL Server has the need to move between rows. You will find that a good many developers call whatever function they implement this in `FillRow`—for me it will vary depending on the situation and whether I feel it warrants something more descriptive of what it’s doing or not.

So, with those items described, let’s look at the opening of our function. Our function is going to be providing a directory listing, but one based on information that must be retrieved from individual files. This means that we have to enumerate the directory to retrieve each file’s information. Just to add to the flavor of things a bit, we will also support the notion of subdirectories—which means we have to understand the notion of directories within directories.

We’ll start with our top-level function call. This accepts the search filter criteria, including the directory we are considering the root directory for our list, the filename criteria for the search, and a Boolean indicator of whether to include subdirectories or not.

```
public partial class UserDefinedFunctions
{
    [SqlFunction(FillRowMethodName = "FillRow")]
    public static IEnumerable DirectoryList(string sRootDir, string sWildCard, bool bIncludeSubDirs)
    {
        // retrieve an array of directory entries. Where this an object of our own
        // making, // it would need to be one that supports IEnumerable, but since ArrayList
        // already // does that, we have nothing special to do here.
        ArrayList aFileArray = new ArrayList();
        DirectorySearch(sRootDir, sWildCard, bIncludeSubDirs, aFileArray);

        return aFileArray;
    }
}
```

This has done little other than establish an array that will hold our file list and call to internal functions to populate it. Next, we need to implement the function that is enumerating the directory list to get the files in each directory:

```
private static void DirectorySearch(string directory, string sWildCard, bool bIncludeSubDirs, ArrayList aFileArray)
{
    GetFiles(directory, sWildCard, aFileArray);

    if (bIncludeSubDirs)
    {
        foreach (string d in Directory.GetDirectories(directory))
        {
            DirectorySearch(d, sWildCard, bIncludeSubDirs, aFileArray);
        }
    }
}
```

For each directory we file, we make a simple call of the `GetFiles` method (it is implemented in `System.IO`) and enumerate the results for the current directory. As we enumerate, we populate our array with the `FullName` and a `LastWriteTime` properties from the file:

```
private static void GetFiles(string d, string sWildCard, ArrayList aFileArray)
{
    foreach (string f in Directory.GetFiles(d, sWildCard))
    {
        FileInfo fi = new FileInfo(f);

        object[] column = new object[2];
        column[0] = fi.FullName;
        column[1] = fi.LastWriteTime;

        aFileArray.Add(column);
    }
}
```

From there, we're ready to bring it all home by actually implementing our `FillRow` function, which does nothing more than serve as a conduit between our array and the outside world — managing the feed of data to one row at a time.

```
private static void FillRow(Object obj, out string filename, out DateTime date)
{
    object[] row = (object[])obj;

    filename = (string)row[0];
    date = (DateTime)row[1];
}
```

Chapter 14

With all that done, we should be ready to compile and upload our assembly. We use the same CREATE ASSEMBLY command we've used all chapter long, but there is a small change — we must declare the assembly as having the EXTERNAL_ACCESS permission set. One of two conditions that must be met in order to do this:

- ❑ The assembly is signed with a certificate (more on these in Chapter 22) that corresponds to a user with proper EXTERNAL_ACCESS rights.
- ❑ The database owner has EXTERNAL_ACCESS rights *and* the database has been marked as being TRUSTWORTHY in the database properties.

We're going to take the unsigned option, so we need to set the database to be marked as trustworthy:

```
ALTER DATABASE AdventureWorks  
SET TRUSTWORTHY ON
```

And we're now ready to finish uploading our assembly with proper access:

```
USE AdventureWorks  
  
CREATE ASSEMBLY fExampleTVF  
FROM '<solution path>\ExampleTVF\bin\Debug\ExampleTVF.dll'  
WITH PERMISSION_SET = EXTERNAL_ACCESS
```

The actual creation of the function reference that utilizes our assembly is not bad but is slightly trickier than the one for the simple scalar function. We must define the table that is to be returned in addition to the input parameters:

```
CREATE FUNCTION fTVFExample  
(  
    @RootDir nvarchar(max),  
    @WildCard nvarchar(max),  
    @IncludeSubDirs bit  
)  
RETURNS TABLE  
(  
    FileName nvarchar(max),  
    LastWriteTime nvarchar(max)  
)  
AS EXTERNAL NAME ExampleTVF.UserDefinedFunctions. DirectoryList
```

And, with that, we're ready to test:

```
SELECT FileName, LastWriteTime  
FROM dbo.fTVFExample('C:\', '*.sys', 0)
```

What you get back when you run this will vary a bit depending on what components and examples you have installed, but, in general, it should look something like:

```

FileName                                LastWriteTime
-----
C:\CONFIG.SYS                            2006-04-01 00:21:43.470
C:\IO.SYS                               2006-04-01 00:21:43.470
C:\MSDOS.SYS                            2006-04-01 00:21:43.470
C:\pagefile.sys                          2006-05-02 20:08:56.500

(4 row(s) affected)

```

We've now shown not only how we can do table valued functions but also how we can access external data — powerful stuff!

Creating Aggregate Functions

Now this one is going to be the one thing in this chapter that's really new. When we look at user-defined data types a little later, we'll see something with a bigger shift than some of the other constructs we've looked at here, but aggregate functions are something that you can't do any other way — the T-SQL version of a UDF does not allow for aggregation.

So, what am I talking about here? Well, examples are `SUM`, `AVG`, `MIN`, and `MAX`. These all look over a set of data and then return a value that is based on some analysis of the whole. It may be based on your entire result set or on some criteria defined in the `GROUP BY` clause.

Performing the analysis required to support your aggregate gets rather tricky. Unlike other functions, where everything can be contained in a single call to your procedure, aggregates require mixing activities your function does (the actual aggregation part) with activities SQL Server is doing at essentially the same time (organizing the groups for the `GROUP BY` for example). The result is something of staged calls to your assembly class. Your class can be called at any of four times and can support methods for each of these calls:

- `Init` — This supports the initialization of your function. Since you're aggregating, there's a good chance that you are setting some sort of accumulator or other holding value — this is the method where you would initialize variables that support the accumulation or holding value.
- `Accumulate` — This is called by SQL Server once for every row that is to be aggregated. How you choose to utilize the function is up to you, but presumably it will implement whatever accumulation logic you need to support your aggregate.
- `Merge` — SQL Server is a multithreaded application, and it may very well utilize multiple threads that can each be calling into your function. As such, you utilize this function to deal with merging the results from different threads into one final result. Depending on the type of aggregate you're doing, this can make things rather tricky. You can, however, make use of private members in your class to keep track of how many threads were running and reconcile the differences. It's worth noticing that this function receives a copy of your class as an argument (consider it to be what amounts to recursive when you are in this method) rather than whatever other type of value you've been accumulating — this is so you get the proper results that were calculated by the other thread.
- `Terminate` — This is essentially the opposite of `Init`. This is the call that actually returns the end result.

Chapter 14

Now, let's see what this looks like in practice.

To start things off, create a new project in Visual Studio (I'm calling mine `ExampleAggregate`), and then add a new aggregate to the project (right-click on the project in the solution and choose `Add → Aggregate`). SQL Server builds you a stock template that includes all four of the methods we just discussed:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedAggregate(Format.Native)]
public struct ExampleAggregate
{
    public void Init()
    {
        // Put your code here
    }

    public void Accumulate(SqlString Value)
    {
        // Put your code here
    }

    public void Merge(ExampleAggregate Group)
    {
        // Put your code here
    }

    public SqlString Terminate()
    {
        // Put your code here
        return new SqlString("");
    }

    // This is a place-holder member field
    private int var1;
}
```

This is the genuine foundation — complete with templates for all four method calls.

What we're going to be doing for an example in this section is to build an implementation of a `PRODUCT` function, which is essentially the same concept as `SUM` but multiplies instead of adding. Like the `SUM` function, we will ignore `NULL` values (unless they are all `NULL`, and then we will return `NULL`), but we will warn the user about the `NULL` being found and ignored should we encounter one.

We need to start with some simple changes. First, I'm going to change the class name to `Product` instead of `ExampleAggregate`, which we called the project. In addition, I need to declare some member variables to hold our accumulator and some flags.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedAggregate(Format.Native)]
public struct Product
{
    private SqlDouble dAccumulator;
    private bool fContainsNull;
    private bool fAllNull;
```

The `fContainsNull` variable will be used to tell us if we need to warn the user about any values being ignored. The `fAllNull` is used to tell if every value received was null—in which case we want to return null as our result.

We then need to initialize our member variables as part of the `Init` function:

```
public void Init()
{
    // Initialize our flags and accumulator
    dAccumulator = 1;
    fContainsNull = false;
    fAllNull = true;

}
```

We are then ready to build the main accumulator function:

```
public void Accumulate(SqlDouble Value)
{
    // This is the meat of things. This one is where we actually apply
    // whatever logic is appropriate for our accumulation. In our example,
    // we simply multiply whatever value is already in the accumulator by
    // the new input value. If the input value is null, then we set the
    // flag that indicates that we've seen null values and then ignore
    // the value we just received and maintain the existing accumulation.

    if (Value.IsNull)
    {
        fContainsNull = true;
    }
    else
    {
        fAllNull = false;
        dAccumulator *= Value;
    }
}
```

Chapter 14

The comments pretty much tell the tale here. We need to watch to make sure that none of our flag conditions have changed. Other than that, we simply need to continue accumulating by multiplying the current value (assuming it's not null) by the existing accumulator value.

With the accumulator fully implemented, we can move on to dealing with the merge scenario.

```
public void Merge(Product Group)
{
    // For this particular example, the logic of merging isn't that hard.
    // We simply multiply what we already have by the results of any other
    // instances of our Product class.

    if (Group.dAccumulator.IsNull)
    {
        if (Group.fContainsNull)
            fContainsNull = true;
        if (!Group.fAllNull)
            fAllNull = false;
        dAccumulator *= dAccumulator;
    }
}
```

For this particular function, the implementation of a merge is essentially just applying the same checks that we did in the `Accumulate` function.

Finally, we're ready to implement our `Terminate` function to close out our aggregation when it's done:

```
public SqlDouble Terminate()
{
    // And this is where we wrap it all up and output our results
    if (fAllNull)
    {
        return SqlDouble.Null;
    }
    else
    {
        SqlContext.Pipe.Send("WARNING: Aggregate values exist and were
ignored");
        return dAccumulator;
    }
}
```

With all that done, we should be ready to compile our procedure and upload it:

```
CREATE ASSEMBLY ExampleAggregate
FROM '<solution path>\ExampleAggregate\bin\Debug\ExampleAggregate.dll'
```

And create the aggregate. Note that while an aggregate is a type of function, we use a different syntax to create it. The basic syntax looks like this:

```
CREATE AGGREGATE [ <schema name> . ] <aggregate name>
    (@param_name <input sqltype> )
RETURNS <return sqltype>
EXTERNAL NAME <assembly name> [ .<class name> ]
```

So, to create the aggregate from our assembly, we would do something like:

```
CREATE AGGREGATE dbo.Product(@input float)
RETURNS float
EXTERNAL NAME ExampleAggregate.Product
```

And, with that, we're ready to try it out. To test it, we'll create a small sample table that includes some data that can be multiplied along with a grouping column, so we can test out how our aggregate works with a GROUP BY scenario.

```
CREATE TABLE TestAggregate
(
    PK          int      NOT NULL      PRIMARY KEY,
    GroupKey    int      NOT NULL,
    Value       float    NOT NULL
)
```

Now we just need some test data:

```
INSERT INTO TestAggregate(PK, GroupKey, Value)
VALUES (1, 1, 2)
INSERT INTO TestAggregate(PK, GroupKey, Value)
VALUES (2, 1, 6)
INSERT INTO TestAggregate(PK, GroupKey, Value)
VALUES (3, 1, 1.5)
INSERT INTO TestAggregate(PK, GroupKey, Value)
VALUES (4, 2, 2)
INSERT INTO TestAggregate(PK, GroupKey, Value)
VALUES (5, 2, 6)
```

And we're ready to give our aggregate a try. What we're going to be doing is returning the PRODUCT of all the rows within each group (our sample data has two groups, so this should work out to two rows).

```
SELECT GroupKey, dbo.Product(Value) AS Product
FROM TestAggregate
GROUP BY GroupKey
```

Run this and we get back two rows (just as we expected):

GroupKey	Product
1	18
2	12

(2 row(s) affected)

Do the match on our sample data, and you'll see we got back just what we wanted.

If you're thinking about it, you should be asking yourself "OK, this is great, but how often am I really going to use this?" For most of you, the answer will be "never." There are, however, those times where what's included just isn't ever going to do the job. Aggregates are one of those places where special cases come rarely, but when they come, they really need exactly what they need and nothing else. In short, I wouldn't crowd your brain cells by memorizing every little thing about this section, but do take the time to learn what's involved and get a concept for what it can and can't do so you know what's available should you need it.

Creating Triggers from Assemblies

Much like the other assembly types we've worked with so far in this chapter, triggers have a lot in common with the rest, but also their own little smattering of special things.

The differences will probably come to mind quickly if you think about it for any length of time:

- ❑ How do we deal with the contextual nature of triggers? That is, how do we know to handle things differently if it's an `INSERT` trigger situation versus a `DELETE` or `UPDATE` trigger?
- ❑ How do we access the inserted and deleted tables?

You may recall from earlier examples, how we can obtain the “context” of the current connection—it is by utilizing this context that we are able to gain access to different objects that we are interested in. For example, the `SqlContext` object that we've obtained a connection from in prior examples also contains a `SqlTriggerContext` object—we can use that to get properties such as whether we are dealing with an insert, update, or delete scenario (the first question we had). The fact that we have access to the current connections also implies that we are able to access the inserted and deleted tables simply by querying them. Let's get right to putting this to use in an example.

Start by creating a new SQL Server project in Visual Studio (I've called mine `ExampleTrigger` this time). Once your project is up, right-click the project in the solution explorer and select `Add>Trigger...`

Visual Studio is nice enough to provide you with what is, for the most part, a working template. Indeed, it would run right as provided except for one issue:

```
using System;
using System.Data;
using System.Data.SqlClient;
using Microsoft.SqlServer.Server;

public partial class Triggers
{
    // Enter existing table or view for the target and uncomment the attribute line
    // [Microsoft.SqlServer.Server.SqlTrigger (Name="ExampleTrigger",
    Target="Table1", Event="FOR UPDATE")]
```

```

public static void ExampleTrigger()
{
    // Replace with your own code
    SqlContext.Pipe.Send("Trigger FIRED");
}
}

```

I've highlighted the key code line for you. At issue is that we must provide more information to SQL Server than we do in our other object types. Specifically, we must identify what table and events we're going to be executing our trigger against. We're actually going to create a special demonstration table for this before the trigger is actually put into action, so we can just use the table name `TriggerTable` for now.

```
[Microsoft.SqlServer.Server.SqlTrigger (Name="ExampleTrigger",
Target="TriggerTable", Event="FOR INSERT, UPDATE, DELETE")]
```

Notice that I've also altered what events will fire our trigger to include all event types.

Now we'll update the meat of things just a bit, so we can show off different actions we might take in our trigger and, perhaps more importantly, how we can check the context of things and make our actions specific to what has happened to our table. We'll start by getting our class going:

```

public static void ExampleTrigger()
{
    // Get a handle to our current connection
    SqlConnection cn = new SqlConnection("context connection=true");
    cn.Open();

    SqlTriggerContext ctxt = SqlContext.TriggerContext;
    SqlCommand cmd = new SqlCommand();
    cmd.Connection = cn;
}

```

So far, this isn't much different from what we've used in our other .NET examples. Perhaps the only significant difference from things we've seen already is the `SqlTriggerContext` object—we will use this later on to determine what action caused the trigger to fire.

We're ready to start code that is conditional on the action the trigger is firing for (based on the `TriggerAction` property of the `TriggerContext` of the `SqlContext`). For this, I'm going to use a simple switch command (though there are those that will call me a programming charlatan for using a switch statement—to them I say "deal with it!"). I'm also going to pipe out various things to the client to report what we're doing.

In practice, you generally do not want to be outputting information from a trigger—figure that they should usually run silently as far as the client is concerned. I've gone ahead and output several items in this example just to make it readily apparent what the trigger is doing under what scenario.

Chapter 14

```
switch (ctxt.TriggerAction)
{
    case TriggerAction.Insert:
        cmd.CommandText = "SELECT COUNT(*) AS NumRows FROM INSERTED";
        SqlContext.Pipe.Send("Insert Trigger Fired");
        SqlContext.Pipe.ExecuteAndSend(cmd);
        break;

    case TriggerAction.Update:
        // This time, we'll use datareaders to show how we can
        // access the data from the inserted/deleted tables

        SqlContext.Pipe.Send("Update Trigger Fired");
        SqlContext.Pipe.Send("inserted rows...");
        cmd.CommandText = "SELECT * FROM INSERTED";
        SqlContext.Pipe.Send(cmd.ExecuteReader());
        break;

    case TriggerAction.Delete:
        // And now we'll go back to what we did with the inserted rows...
        cmd.CommandText = "SELECT COUNT(*) AS NumRows FROM DELETED";
        SqlContext.Pipe.Send("Delete Trigger Fired");
        SqlContext.Pipe.ExecuteAndSend(cmd);
        break;
}

SqlContext.Pipe.Send("Trigger Complete");
}
```

And, with that, we're ready to compile and upload it. The assembly upload works just as most of them have so far (we're back to not needing anything other than the default PERMISSION_SET).

```
CREATE ASSEMBLY ExampleTrigger
FROM '<solution path>\ExampleTrigger\bin\Debug\ExampleTrigger.dll'
```

Before we get to creating the reference to the trigger, however, we need a table. For this example, we'll just create something very simple:

```
CREATE TABLE TestTrigger
(
    PK      int          NOT NULL PRIMARY KEY,
    Value   varchar(max) NOT NULL
)
```

With the assembly uploaded and the table created, we're ready to create our trigger reference.

Much like stored procedures and functions, a .NET trigger creation is made from the same statement as T-SQL-based triggers. We eliminate the T-SQL side of things and replace it with the EXTERNAL NAME declaration.

```
CREATE TRIGGER trgExampleTrigger
ON TestTrigger
FOR INSERT, UPDATE, DELETE
AS EXTERNAL NAME ExampleTrigger.Triggers.ExampleTrigger
```

And with that, our trigger should be in place on our table and ready to be fired whenever one of its trigger actions is fired (which happens to be for every trigger actions), so let's test it.

We'll start by getting a few rows inserted into our table. And, wouldn't you just know it? That will allow us to test the insert part of our trigger.

```
INSERT INTO TestTrigger
(PK, Value)
VALUES
(1, 'first row')

INSERT INTO TestTrigger
(PK, Value)
VALUES
(2, 'second row')
```

Run this, and we not only get our rows in but we also get a little bit of feedback that is coming out of our trigger:

```
Insert Trigger Fired
NumRows
-----
1

(1 row(s) affected)

Trigger Complete

(1 row(s) affected)
Insert Trigger Fired
NumRows
-----
1

(1 row(s) affected)

Trigger Complete

(1 row(s) affected)
```

As you can see, we're getting output from our trigger. Notice that we're getting the "(1 row(s) affected)" both from the query running inside the trigger and from the one that actually inserted the data. We could have taken any action that could have been done a T-SQL trigger (though many are more efficient if you stay in the T-SQL world). The key is that we could do so much more if we had the need. We could, for example, make an external call or perform a calculation that isn't doable in the T-SQL world.

Chapter 14

There is an old saying: “Caution is the better part of valor.” This could have been written with triggers in mind. I can’t possibly express enough about the “be careful” when it comes to what you’re doing in triggers. Just because you can make an external call doesn’t make it a smart thing to do. Assess the need—is it really that important that the call be made right then? Realize that these things can be slow, and whatever transaction that trigger is participating in will not complete until the trigger completes—this means you may be severely damaging performance.

Okay, so with all that done, let’s try an update:

```
UPDATE TestTrigger  
SET Value = 'Updated second row'  
WHERE PK = 2
```

And let’s see what we get back:

```
Update Trigger Fired  
inserted rows...  
PK          Value  
-----  
2           Updated second row  
  
(1 row(s) affected)  
  
Trigger Complete  
  
(1 row(s) affected)
```

The result set we’re getting back is the one our trigger is outputting. That’s followed by some of our other output as well as the base “(1 row(s) affected)” that we would normally expect from our single row update. Just as with the insert statement, we were able to see what had happened and could have adapted accordingly.

And so, that leaves us with just the delete statement. This time, we’ll delete all the rows, and we’ll see how the count of our deleted table does indeed reflect both of the deleted rows.

```
DELETE TestTrigger
```

And again check the results:

```
Delete Trigger Fired  
NumRows  
-----  
2  
  
(1 row(s) affected)  
  
Trigger Complete  
  
(2 row(s) affected)
```

Now, these results may be just a little confusing, so let’s look at what we have.

We start with the notification that our trigger fired — that comes from our trigger (remember, we send that message down the pipe ourselves). Then comes the result set from our `SELECT COUNT(*)`. Notice the “`(1 row(s) affected)`” — that’s from our result set rather than the `UPDATE` that started it all. We then get to the end of execution of our trigger (again, we dropped that message in the pipe), and, finally, the “`(2 row(s) affected)`” that was from the original `UPDATE` statement.

And there we have it. We’ve done something to address every action scenario, and we could, have course, done a lot more within each. We could also do something to address a `BEFORE` trigger if we needed to.

Custom Data Types

Sometimes you have the need to store data that you want to be strongly typed, but that SQL Server doesn’t fit within SQL Server’s simple data type list. Indeed, you may need to invoke a complex set of rules in order to determine whether the data properly meets the type requirement or not.

Requests for support of complex data types have been around a very long time. Indeed, I can recall being at the Sphinx Beta 2.0 — known to most as Beta 2 for SQL Server 7.0 — event in 1998, and having that come up as something like the second most requested item in a request session I was at. Well, it took a lot of years, but it’s finally here.

By utilizing a .NET assembly, we can achieve a virtually limitless number of possibilities in our data types. The type can have complex rules or even contain multiple properties.

Before we get to the syntax for adding assemblies, let’s get an assembly constructed.

The sample used here will be the `ComplexNumber.sln` solution included in the SQL Server samples. You will need to locate the base directory for the solution — the location of which will vary depending on your particular installation.

We need to start by creating the signature keys for this project. To do this, I recommend starting with your solution directory being current and then calling `sn.exe` using a fully qualified path (or, if your .NET framework directory is already in your `PATH`, then it’s that much easier!). For me, it looks like this:

```
C:\Program Files\Microsoft.NET\SDK\v2.0 64bit\LateBreaking\SQLCLR\UserDefinedDat
aType>"C:\Program Files (x86)\Microsoft Visual Studio 8\SDK\v2.0\Bin\sn" -k temp
.snk
```

And with that, you’re ready to build your DLL.

Let’s go ahead and upload the actual assembly (alter this to match the paths on your particular system):

```
CREATE ASSEMBLY ComplexNumber
    FROM <solution path>\ComplexNumber\bin\debug\ComplexNumber.dll'
    WITH PERMISSION_SET = SAFE;
```

Chapter 14

And with the assembly loaded, we're ready to begin.

Creating Your Data Type from Your Assembly

So, you have an assembly that implements your complex data type and have uploaded it to SQL Server using the `CREATE ASSEMBLY` command. You're ready to instruct SQL Server to use it. This works pretty much as other assemblies have. The syntax looks like this:

```
CREATE TYPE [ <schema name>. ] <type name>
{
    FROM <base type>
    [ ( <precision> [ , <scale> ] ) ]
    [ NULL | NOT NULL ]
    | EXTERNAL NAME <assembly name> [ .<class name> ]
} [ ; ]
```

I've put the full type definition there, but really it's just the highlighted line that is different from the older style user-defined data type definition. You'll notice immediately that it looks like our previous assembly-related constructs, and, indeed, the use is the same.

So, utilizing our complex type created in the last section, it would look like this:

```
CREATE TYPE ComplexNumber
    EXTERNAL NAME [ComplexNumber].[Microsoft.Samples.SqlServer.ComplexNumber];
```

Accessing Your Complex Data Type

Microsoft has provided a file called `test.sql` for testing the assembly we just defined as our complex data type, but I find it falls just slightly short of where we want to be in our learning here. What I want to emphasize is how the various functions of the supporting class for our data type are still available. In addition, each individual property of the variable is fully addressable. So, let's run a modified version of the provided script:

```
USE AdventureWorks
GO

-- create a variable of the type, create a value of the type and invoke
-- a behavior over it

DECLARE @c ComplexNumber;

SET @c = CONVERT(ComplexNumber, '(1, 2i)');

SELECT @c.ToString() AS FullValueAsString;

SELECT @c.Real AS JustRealProperty
GO
```

Now run it, and check out the results:

```
FullValueAsString
-----
(1,2i)

(1 row(s) affected)

JustRealProperty
-----
1

(1 row(s) affected)
```

In the first result that was returned, the `ToString` function was called as defined as a method of our class. The string is formatted just as our method desires. If we had wanted to reverse the order of the numbers or some silly thing like that, we would only have needed to change the `ToString` function in the class, recompile it, and re-import it our database.

In our second result, we address just one property of our complex data type. The simple dot “.” delimiter told SQL Server that we were looking for a property—just as it would in C# or VB.NET.

Dropping Data Types

As you might expect, the syntax for dropping a user defined data type works just like other drop statements:

```
DROP TYPE [ <schema name>. ] <type name> [ ; ]
```

And it's gone—maybe.

Okay, so why a “maybe” this time? Well, if there is most any object out there that references this data type, then the `DROP` will be disallowed and will fail. So, if you have a table that has a column of this type, then an attempt to drop it would fail. Likewise, if you have a schema bound view, stored procedure, trigger, or function defined that utilizes this type, then a drop would also fail.

Note that this form of restriction appears in other places in SQL Server—such as dropping a table when it is the target of a foreign key reference—but those restrictions tend to be less all encompassing than this one is (virtually any use of it in your database at all will block the drop), so I haven't felt as much need to point it out (they were more self-explanatory).

Summary

Well, if you aren't thinking to yourself something along the lines of “Wow, that's powerful,” then I can only guess you somehow skipped straight to the summary without reading the rest of the chapter. That's what this chapter is all about—giving you the power to do very complex things (or, in a few cases, simple things that still weren't possible before). What this chapter was also about, in a very subtle way, was a mechanism that can break your system very quickly. Yeah, yeah, yeah—I know I'm resorting to scare tactics, but I make no apologies for it. When using assemblies, you need to be careful. Think about what you're doing, and analyze each of the steps that your assembly is going to be taking even more

Chapter 14

thoroughly than you already do. Consider latency you're going to be adding if you create long-running processes. Consider external dependencies you are creating if you make external calls—how reliable are those external processes? You need to know, as your system is now only as reliable as the external systems you're calling.

Now, having hopefully scared you into caution about assemblies, let me ease up a bit and say don't avoid them solely out of fear. Assemblies were added for a reason, and they give us a power we both need and can use (not to mention that they are just plain cool). As always, think about what you need, and don't make your solution any more complex than it needs to be. Keep in mind, however, that what seems at first to be the more complex solution may actually be simpler in the end. I've seen stored procedures that solved the seemingly unsolvable T-SQL problem. Keeping your system away from assemblies would seem to make it simpler, but what's better: a 300-line, complex T-SQL stored proc or an assembly that is concise and takes only 25 lines including declarations?

Choose wisely.

15

SQL Cursors

Throughout this book thus far, we've been dealing with data in sets. This tends to go against the way that the more procedure-driven languages go about things. Indeed, when the data gets to the client end, they almost always have to take our set and then deal with it row by row. What they are dealing with is a *cursor*. Indeed, even in traditional SQL Server tools, we can wind up in something of a cursor mode if we utilize a non-SQL-oriented language in our scripts using the new CLR-based language support.

In this chapter, we will be looking at:

- What a cursor is
- The lifespan of a cursor
- Cursor types (sensitivity and scrollability)
- Uses for cursors

We'll discover that there's a lot to think about when creating cursors.

Perhaps the biggest thing to think about when creating cursors is, "Is there a way I can get out of doing this?" If you ask yourself that question every time you're about to create a cursor, then you will be on the road to a better performing system. That being said, we shall see that there are times when nothing else will do.

What Is a Cursor?

Cursors are a way of taking a set of data and being able to interact with a single record at a time. It doesn't happen nearly as often as one tends to think, but there are indeed times where you just can't obtain the results you want to by modifying or even selecting the data in an entire set. The set is generated by something all of the rows have in common (as defined by a SELECT statement), but then you need to deal with those rows on a one-by-one basis.

Chapter 15

The result set that you place in a cursor has several distinct features that set it apart from a normal SELECT statement:

- You declare the cursor separately from actually executing it.
- The cursor and, therefore, its result set are named at declaration—you then refer to it by name.
- The result set in a cursor, once opened, stays open until you close it.
- Cursors have a special set of commands used to navigate the recordset.

While SQL Server has its own engine to deal with cursors, there are actually a few different object libraries that can also create cursors in SQL Server:

- SQL Native Client (used by ADO.NET)
- OLE DB (used by ADO)
- ODBC (used by RDO, DAO, and in some cases, OLE DB/ADO)
- DB-Lib (used by VB-SQL)

These are the libraries that client applications will typically use to access individual records. Each provides its own syntax for navigating the recordset and otherwise managing the cursor. Each, however, shares in the same set of basic concepts, so, once you have got one object model down for cursors, you're most of the way there for all of them.

Every data access API out there (ADO.NET, ADO, ODBC, OLE DB, etc.) returns data to a client application or component in a cursor—it's simply the only way that non-SQL programming languages can currently deal with things. This is the source of a big difference between this kind of cursor and SQL Server cursors. With SQL Server cursors, you usually have a choice to perform things as a set operation, which is what SQL Server was designed to do. With the API-based cursors, all you have is cursors, so you don't have the same cursor versus no cursor debate that you have in your server-side activities.

The client-side part of your data handling is going to be done using cursors—that's a given, so don't worry about it. Instead, worry about making the server side of your data access as efficient as possible—that means not using cursors on the server side if you can possibly help it.

The Lifespan of a Cursor

Cursors have lots of little pieces to them, but I think that it's best if we get right into looking first at the most basic form of cursor and then build up from there.

Before we get into the actual syntax though, we need to understand that using a cursor requires more than one statement—indeed, it takes several. The main parts include:

- The declaration
- Opening

- Utilizing/navigating
- Closing
- Deallocating

That being said, the basic syntax for declaring a cursor looks like this:

```
DECLARE <cursor name> CURSOR  
FOR <select statement>
```

Keep in mind that this is the super-simple rendition — create a cursor using defaults wherever possible. We'll look at more advanced cursors a little later in the chapter.

The cursor name is just like any other variable name, and, other than not requiring the "@" prefix, they must obey the rules for SQL Server naming. The SELECT statement can be any valid SELECT statement that returns a result set. Note that some result sets will not, however, be updatable. (For example, if you use a GROUP BY, then what part of the group is updated? The same holds true for calculated field for much the same reason.)

We'll go ahead and start building a reasonably simple example. For now, we're not really going to use it for much, but we'll see later that it will be the beginning of a rather handy tool for administering your indexes:

```
DECLARE @SchemaName varchar(255)  
DECLARE @TableName varchar(255)  
DECLARE @IndexName varchar(255)  
DECLARE @Fragmentation float  
DECLARE TableCursor CURSOR FOR  
    SELECT SCHEMA_NAME(CAST(OBJECTPROPERTYEX(i.object_id, 'SchemaId') AS int)),  
        OBJECT_NAME(i.object_id),  
        i.name,  
        ps.avg_fragmentation_in_percent  
    FROM sys.dm_db_index_physical_stats (DB_ID(), NULL, NULL, NULL, NULL) AS ps  
    JOIN sys.indexes AS i  
        ON ps.object_id = i.object_id  
        AND ps.index_id = i.index_id  
    WHERE avg_fragmentation_in_percent > 30
```

Note that this is just the beginning of what you will be building. One of the first things you should notice about cursors is that they require a lot more code than the usual SELECT statement.

We've just declared a cursor called TableCursor that is based on a SELECT statement that will select all of the tables in our database. We also declare a holding variable that will contain the values of our current row while we are working with the cursor.

Just declaring the cursor isn't enough though — we need to actually open it:

```
OPEN TableCursor
```

Chapter 15

This actually executes the query that was the subject of the FOR clause, but we still don't have anything in place we can work with it. For that, we need to do a couple of things:

- ❑ Grab—or FETCH—our first record
- ❑ Loop through, as necessary, FETCHing the remaining records

We issue our first FETCH—this is the command that says to retrieve a particular record. We must also say into which variables we want to place the values:

```
FETCH NEXT FROM TableCursor INTO @TableName, @IndexName, @Fragmentation
```

Now that we have a first record, we're ready to move onto performing actions against the cursor set:

```
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT @SchemaName + '.' + @TableName + '.' + @IndexName + ' is '
    + CAST(@Fragmentation AS varchar) + '% Fragmented'
    FETCH NEXT FROM TableCursor INTO @SchemaName, @TableName, @IndexName,
        @Fragmentation
END
```

You may remember @@FETCH_STATUS from our brief discussion of globals earlier in the book. Every time we fetch a row, @@FETCH_STATUS is updated to tell us how our fetch went. The possible values are:

- ❑ **0 Fetch succeeded**—Everything's fine.
- ❑ **-1 Fetch failed**—Record missing (you're not at the end, but a record has been deleted since you opened the cursor). We'll look at this closer later in the chapter.
- ❑ **-2 Fetch failed**—This time it's because you're beyond the last (or before the first) record in the cursor. We'll also see more of this later in the chapter.

Once we exit this loop, we are, for our purposes here, done with the cursor, so we'll close it:

```
CLOSE TableCursor
```

Closing the cursor, does not, however, free up the memory associated with that cursor. It does free up the locks associated with it. In order to be sure that you've totally freed up the resources used by the cursor, you must deallocate it:

```
DEALLOCATE TableCursor
```

So, let's bring it all together just for clarity:

```
DECLARE @SchemaName varchar(255)
DECLARE @TableName varchar(255)
DECLARE @IndexName varchar(255)
DECLARE @Fragmentation float
DECLARE TableCursor CURSOR FOR
```

```

SELECT SCHEMA_NAME(CAST(OBJECTPROPERTYEX(i.object_id, 'SchemaId') AS int)),
       OBJECT_NAME(i.object_id),
       i.name,
       ps.avg_fragmentation_in_percent
  FROM sys.dm_db_index_physical_stats (DB_ID(), NULL, NULL, NULL, NULL) AS ps
 JOIN sys.indexes AS i
    ON ps.object_id = i.object_id
   AND ps.index_id = i.index_id
 WHERE avg_fragmentation_in_percent > 30
OPEN TableCursor
FETCH NEXT FROM TableCursor INTO @SchemaName, @TableName, @IndexName,
                           @Fragmentation

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT @SchemaName + '.' + @TableName + '.' + @IndexName + ' is ' +
    CAST(@Fragmentation AS varchar) + '% Fragmented'
    FETCH NEXT FROM TableCursor INTO @SchemaName, @TableName, @IndexName,
                           @Fragmentation
END
CLOSE TableCursor
DEALLOCATE TableCursor

```

We now have something that runs, but as we've created it at the moment, it's really nothing more than if we had just run the `SELECT` statement by itself (technically, this isn't true since we can't "PRINT" a `SELECT` statement, but you could do what amounts to the same thing). What's different is that, if we so chose, we could have done nearly anything to the individual rows. Let's go ahead and illustrate this by completing our little utility.

In days of old, there was no single statement that will rebuild all the indexes in an entire database (fortunately, we now have an option in `DBCC INDEXDEFrag` to do an entire database). Keeping your indexes defragmented was, however, a core part of administering your system. The cursor example we're using here is something of a descendent of what was the common way of getting this kind of index defragmentation done. In this newer version, however, we're making use of specific fragmentation information, and we're making it possible to allow for the use of `ALTER INDEX` (which allows for more options in how exactly to do our defragmentation) instead of `DBCC INDEXDEFrag`.

Okay, so we have a few different methods for rebuilding or reorganizing indexes without entirely dropping and recreating them. `ALTER INDEX` is the most flexible in terms of letting you select different underlying methods of defragmenting (online or offline, complete rebuild or just a reorganization of what's there, etc.), so we're going to leverage this way of doing things. The simple version of the syntax for `ALTER INDEX` looks like this:

```

ALTER INDEX <index name> | ALL
  ON <object>
  { [REBUILD] | [REORGANIZE] }

```

Again, this is the hyper-simple version of `ALTER INDEX`. There are a ton of other little switches and options for it that are described in Chapter 8.

Chapter 15

The problem with trying to use this statement to rebuild all the indexes on all of your tables is that it is designed to work on one table at a time. You can use the ALL option instead of the index name if you want to build all the indexes for a table, but you can't leave off the table name to build all the indexes for all the tables. Indeed, even if we had used a tool like DBCC INDEXDEFRAG—which can do an entire database, but just doesn't have as many options—it would still be an all-or-nothing thing. That is, we can't tell it to do just the tables above a certain level of fragmentation, or to exclude particular tables that we may *want* to have fragmentation in.

Remember that there are occasionally times when fragmentation is a good thing. In particular, it can be helpful on tables where we are doing a large number of random inserts as it reduces the number of page splits.

Our cursor can get us around this by just dynamically building the DBCC command:

```
DECLARE @SchemaName varchar(255)
DECLARE @TableName varchar(255)
DECLARE @IndexName varchar(255)
DECLARE @Fragmentation float
DECLARE @Command varchar(255)
DECLARE TableCursor CURSOR FOR
    SELECT SCHEMA_NAME(CAST(OBJECTPROPERTYEX(i.object_id, 'SchemaId') AS int)),
           OBJECT_NAME(i.object_id),
           i.name,
           ps.avg_fragmentation_in_percent
      FROM sys.dm_db_index_physical_stats (DB_ID(), NULL, NULL, NULL, NULL) AS ps
     JOIN sys.indexes AS i
       ON ps.object_id = i.object_id
      AND ps.index_id = i.index_id
     WHERE avg_fragmentation_in_percent > 30
OPEN TableCursor
FETCH NEXT FROM TableCursor INTO @SchemaName, @TableName, @IndexName,
@Fragmentation

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT 'Reindexing ' + ISNULL(@SchemaName, 'dbo') + '.' +
          @TableName + '.' + @IndexName
    SET @Command = 'ALTER INDEX [' + @IndexName + '] ON [' +
                  ISNULL(@SchemaName, 'dbo') + '.' + @TableName + '] REBUILD'
    EXEC (@Command)
    FETCH NEXT FROM TableCursor
        INTO @SchemaName, @TableName, @IndexName, @Fragmentation
END
CLOSE TableCursor
DEALLOCATE TableCursor
```

We've now done what would be impossible using only set based commands. The ALTER INDEX command is expecting a single argument—providing that a recordset won't work. We get around the problem by combining the notion of a set operation (the SELECT that forms the basis for the cursor) with single-data-point operations (the data in the cursor).

In order to mix these set-based and individual data point operations, we had to walk through a series of steps. First, we declared the cursor and any necessary holding variables. We then “opened” the cursor — it was not until this point that the data was actually retrieved from the database. Next, we utilized the cursor by navigating through it. In this case, we only navigated forward, but, as we shall see, we could have created a cursor that could scroll forward and backward. Moving on, we closed the cursor (if the cursor had still had any open locks, they were released at this point), but memory continues to be allocated for the cursor. Finally, we deallocated the cursor. At this point, all resources in use by the cursor are freed for use by other objects in the system.

So just that quick, we have our first cursor. Still, this is really only the beginning. There is much more to cursors than meets the eye in this particular example. Next, we’ll go on and take a closer look at some of the powerful features that give cursors additional flexibility.

Types of Cursors and Extended Declaration Syntax

Cursors come in a variety of different flavors (we’ll visit them all before we’re done). The default cursor is forward-only (you can only move forward through the records, not backward) and read-only, but cursors can also be scrollable and updatable. They can also have a varying level of sensitivity to changes that are made to the underlying data by other processes.

The forward-only, read-only cursor is the default type of cursor in not only the native SQL Server cursor engine, but is also default from pretty much all the cursor models I’ve ever bumped into. It is extremely low in overhead, by comparison, to the other cursor choices, and is usually referred to as being a “firehose” cursor because of the sheer speed with which you can enumerate the data. Like a fire hose, it knows how to dump its contents in just one direction though (you can’t put the water back in a fire hose now can you?). Firehose cursors simply blow away the other cursor-based options in most cases, but don’t mistake this as a performance choice over set operations — even a firehose cursor is slow by comparison to most equivalent set operations.

Let’s start out by taking a look at a more extended syntax for cursors, and then we’ll look at all of the options individually:

```
DECLARE <cursor name> CURSOR  
[ LOCAL | GLOBAL ]  
[ FORWARD_ONLY | SCROLL ]  
[ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]  
[ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]  
[ TYPE_WARNING ]  
FOR <SELECT statement>  
[ FOR UPDATE [ OF <column name >[ ,...n ] ] ]
```

At first glance, it really looks like a handful, and indeed, there are a good many things to think about when declaring cursors (as I’ve said, probably the most important is along the lines of, “Do I really need to be doing this?”). The bright side is that several of these options imply one another, so once you’ve made one choice the others often start to fall into place quickly.

Chapter 15

Let's go ahead and apply the specific syntax in a step-by-step manner that attaches each part to the important concepts that go with it.

Scope

The LOCAL versus GLOBAL option determines the scope of the cursor, that is, what connections and processes can "see" the cursor. Most items that have scope will default to the more conservative approach, that is, the minimum scope (which would be LOCAL in this case). SQL Server cursors are something of an exception to this—the default is actually GLOBAL. Before we get too far into the ramifications of the LOCAL versus GLOBAL scope question, we had better digress for a moment as to what I mean by local and global in this context.

We are already dealing with something of an exception in that the default scope is set to what we're calling global rather than the more conservative option of local. The exception doesn't stop there though. In SQL Server, the notion of something being global versus local usually indicates that it can be seen by all connections rather than just the current connection. For the purposes of our cursor declaration, however, it refers to whether all processes (batches, triggers, spocs) in the current connection can see it versus just the current process.

Figure 15-1 illustrates this:

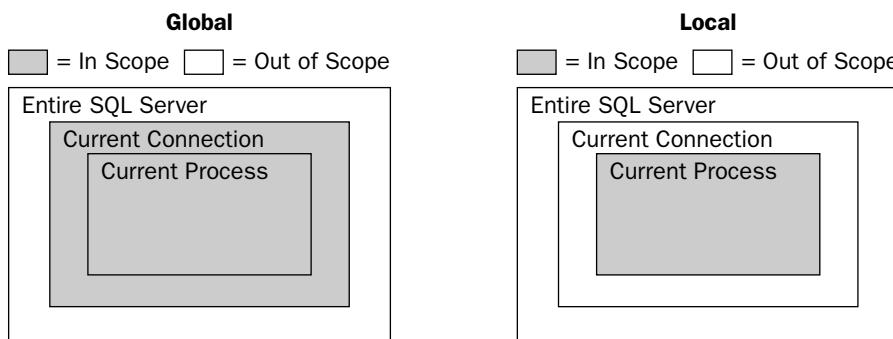


Figure 15-1

Now let's think about what this means, and test it a bit.

The ramifications to the global default fall, as you might expect, on both the pro and the con side of the things. Being global, it means that you can create a cursor within one sproc and refer to it from within a separate sproc—you don't necessarily have to pass references to it. The downside of this though is that, if you try to create another cursor with the same name, you're going to get an error.

Let's test this out with a brief sample. What we're going to do here is create a sproc that will create a cursor for us:

```
USE AdventureWorks  
GO  
  
CREATE PROCEDURE spCursorScope
```

```

AS

DECLARE @Counter      int,
        @OrderID       int,
        @CustomerID    int

DECLARE CursorTest  CURSOR
GLOBAL
FOR
    SELECT SalesOrderID, CustomerID
    FROM Sales.SalesOrderHeader

    SELECT @Counter = 1
OPEN CursorTest
FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID
PRINT 'Row ' + CAST(@Counter AS varchar) + ' has a SalesOrderID of ' +
      CONVERT(varchar,@OrderID) + ' and a CustomerID of ' + CAST(@CustomerID AS
      varchar)

WHILE (@Counter<=5) AND (@@FETCH_STATUS=0)
BEGIN
    SELECT @Counter = @Counter + 1
    FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID
    PRINT 'Row ' + CAST(@Counter AS varchar) + ' has a SalesOrderID of ' +
          CONVERT(varchar,@OrderID) + ' and a CustomerID of ' + CAST(@CustomerID AS
          varchar)
END

```

Notice several things in this sproc. First, I've declared holding variables to do a few things for us. The first, `@Counter`, will just keep tabs on things so we only have to move through a few records rather than moving through the entire recordset. The second and third, `@OrderID` and `@CustomerID`, respectively, will hold the values retrieved from the query as we go row by row through the result set.

Next, we declare the actual cursor. Note that I've explicitly set the scope. By default, if I had left off the `GLOBAL` keyword, then I would have still received a cursor that was global in scope.

You do not have to live by this default. You can use `sp_dboption` or `ALTER DATABASE` to set the “default to local cursor” option to True (set it back to False if you want to go back to global).

This happens to be yet another great example of why it makes sense to always explicitly state the options that you want—don't rely on defaults. Imagine if you were just relying on the default of `GLOBAL` and then someone changed that option in the system! I can just hear plenty of you out there saying, “Oh, no one would ever change that.” WRONG! This is exactly the kind of “small change” that people make to fix some problem somewhere. Depending on the obscurity of your cursor usage, it may be weeks before you run into the problem—by which time you've totally forgotten that the change was made.

We then go ahead and open the cursor and step through several records. Notice, however, that we do not close or deallocate the cursor—we just leave it open and available as we exit the sproc.

Chapter 15

I can't help but think of the old show Lost in Space here, with the robot constantly yelling "DANGER Will Robinson! DANGER!" Leaving cursors open like this willy nilly will lead you to a life of sorrow, frustration, and severe depression.

I'm doing it here to fully illustrate the concept of scope, but you would want to be extremely careful about this kind of usage. The danger lies in the notion that you would call this sproc without realizing that it doesn't clean up after itself. If you don't clean up (close and deallocate) the cursor outside the sproc, then you will create something of a resource leak in the form of this abandoned, but still active, cursor. You will also expose yourself to the possibility of errors should you call the same sproc again (it will try to declare and open the cursor again, but it already exists).

When we look into declaring our cursor for output, we will see a much more explicit and better choice for situations where we want to allow outside interaction with our cursors.

Now that we've enumerated several records and proven that our sproc is operating, we will then exit the sproc (remember, we haven't closed or deallocated the cursor). We'll then refer to the cursor from outside the sproc:

```
EXEC spCursorScope

DECLARE @Counter      int,
        @OrderID       int,
        @CustomerID    int

SET @Counter=6

WHILE (@Counter<=10) AND (@@FETCH_STATUS=0)
BEGIN
    PRINT 'Row ' + CAST(@Counter AS varchar) + ' has a SalesOrderID of ' +
          CAST(@OrderID AS varchar) + ' and a CustomerID of ' +
          CAST(@CustomerID AS varchar)
    SELECT @Counter = @Counter + 1
    FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID
END

CLOSE CursorTest
DEALLOCATE CursorTest
```

Okay, so let's walk through what's happening here.

First, we execute the sproc. As we've already seen, this sproc builds the cursor and then enumerates several rows. It exits, leaving the cursor open.

Next, we declare the very same variables that were declared in the sproc. Why do we have to declare them again, but not the cursor? Because it is only the cursor that is global by default. That is, our variables went away as soon as the sproc went out of scope—we can't refer to them anymore, or we'll get a variable undefined error. We must redeclare them.

The next code structure looks almost identical to one in our sproc—we're again looping through to enumerate several records.

Finally, once we've proven our point that the cursor is still alive outside the realm of the sproc, we're ready to close and deallocate the cursor. It is not until we close the cursor that we free up the memory

or tempdb space from the result set used in the cursor, and it is not until we deallocate that the memory taken up by the cursor variable and its query definition is freed.

Now, go ahead and create the sproc in the system (if you haven't already) and execute the script. You should wind up with a result that looks like this:

```
Row 1 has a SalesOrderID of 43659 and a CustomerID of 676
Row 2 has a SalesOrderID of 43660 and a CustomerID of 117
Row 3 has a SalesOrderID of 43661 and a CustomerID of 442
Row 4 has a SalesOrderID of 43662 and a CustomerID of 227
Row 5 has a SalesOrderID of 43663 and a CustomerID of 510
Row 6 has a SalesOrderID of 43664 and a CustomerID of 397

Row 7 has a SalesOrderID of 43665 and a CustomerID of 146
Row 8 has a SalesOrderID of 43666 and a CustomerID of 511
Row 9 has a SalesOrderID of 43667 and a CustomerID of 646
Row 10 has a SalesOrderID of 43668 and a CustomerID of 514
```

So, you can see that the cursor stayed open, and our loop outside the sproc was able to pick up right where the code inside the sproc had left off.

Now let's see what happens if we alter our sproc to have local scope:

```
USE AdventureWorks
GO

ALTER PROCEDURE spCursorScope
AS

DECLARE @Counter      int,
        @OrderID       int,
        @CustomerID    int

DECLARE CursorTest  CURSOR
LOCAL
FOR
    SELECT SalesOrderID, CustomerID
    FROM Sales.SalesOrderHeader

SELECT @Counter = 1
OPEN CursorTest
FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID
PRINT 'Row ' + CAST(@Counter AS varchar) + ' has a SalesOrderID of ' +
      CAST(@OrderID AS varchar) + ' and a CustomerID of ' + CAST(@CustomerID AS
varchar)

WHILE (@Counter<=5) AND (@@FETCH_STATUS=0)
BEGIN
    SELECT @Counter = @Counter + 1
    FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID
    PRINT 'Row ' + CAST(@Counter AS varchar) + ' has a SalesOrderID of ' +
          CAST(@OrderID AS varchar) + ' and a CustomerID of ' + CAST(@CustomerID AS
varchar)
END
```

Chapter 15

It seems like only a minor change, but the effects are significant when we execute our script again:

```
Row 1 has a SalesOrderID of 43659 and a CustomerID of 676
Row 2 has a SalesOrderID of 43660 and a CustomerID of 117
Row 3 has a SalesOrderID of 43661 and a CustomerID of 442
Row 4 has a SalesOrderID of 43662 and a CustomerID of 227
Row 5 has a SalesOrderID of 43663 and a CustomerID of 510
Row 6 has a SalesOrderID of 43664 and a CustomerID of 397
```

```
Msg 16916, Level 16, State 1, Line 13
A cursor with the name 'CursorTest' does not exist.
Msg 16916, Level 16, State 1, Line 17
A cursor with the name 'CursorTest' does not exist.
Msg 16916, Level 16, State 1, Line 18
A cursor with the name 'CursorTest' does not exist.
```

Things ran just as they did before until we got out of the sproc. This time the cursor was no longer in scope as we came out of the sproc, so we were unable to refer to it, and our script ran into several errors. Later on in the chapter, we'll take a look at how to have a cursor with local scope but still be able to access it from outside the procedure in which it was created.

The big thing that you should have gotten out of this section is that you need to think about the scope of your cursors. They do not behave quite the way that other items for which you use the `DECLARE` statement do.

Scrollability

Like most of the concepts we'll be talking about throughout this chapter, *scrollability* applies to pretty much any cursor model you might face. The notion is actually fairly simple: Can I navigate in relatively any direction, or am I limited to only moving forward? The default is no—you can only move forward.

FORWARD_ONLY

A forward-only cursor is exactly what it sounds like. Since it is the default method, it probably doesn't surprise you to hear that it is the only type of cursor that we've been using up to this point. When you are using a forward-only cursor, the only navigation option that is valid is `FETCH NEXT`. You need to be sure that you're done with each record before you move onto the next because, once it's gone, there's no getting back to the previous record unless you close and reopen the cursor.

SCROLLABLE

Again, this is just as it sounds. You can "scroll" the cursor backward and forward as necessary. If you're using one of the APIs (ODBC, OLE DB, DB-Lib), then, depending on what object model you're dealing with, you can often navigate right to a specific record. Indeed, with ADO and ADO.NET you can even easily resort the data and add additional filters.

The cornerstone of scrolling is the `FETCH` keyword. You can use `FETCH` to move forward and backward through the cursor, as well as move to specific positions. The main arguments to `FETCH` are:

- `NEXT`—Move to the next record.
- `PRIOR`—Move to the previous record.

- FIRST—Move to the first record.
- LAST—Move to the last record.

We'll take a more in-depth look at `FETCH` later in the chapter, but, for now, be aware that `FETCH` exists and is what controls your navigation through the cursor set

Let's do a brief example to get across the concept of a scrollable cursor. We'll actually just use a slight variation of the sproc we created a little earlier in the chapter:

```
USE AdventureWorks
GO

CREATE PROCEDURE spCursorScroll
AS

DECLARE @Counter      int,
        @OrderID       int,
        @CustomerID    int

DECLARE CursorTest   CURSOR
LOCAL
SCROLL
FOR
    SELECT SalesOrderID, CustomerID
    FROM Sales.SalesOrderHeader

    SELECT @Counter = 1
    OPEN CursorTest
    FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID
    PRINT 'Row ' + CAST(@Counter AS varchar) + ' has a SalesOrderID of ' +
          CAST(@OrderID AS varchar) + ' and a CustomerID of ' + CAST(@CustomerID AS
varchar)

    WHILE (@Counter<=5) AND (@@FETCH_STATUS=0)
    BEGIN
        SELECT @Counter = @Counter + 1
        FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID
        PRINT 'Row ' + CAST(@Counter AS varchar) + ' has a SalesOrderID of ' +
              CAST(@OrderID AS varchar) + ' and a CustomerID of ' + CAST(@CustomerID AS
varchar)
    END

    WHILE (@Counter > 1) AND (@@FETCH_STATUS = 0)
    BEGIN
        SELECT @Counter = @Counter - 1
        FETCH PRIOR FROM CursorTest INTO @OrderID, @CustomerID
        PRINT 'Row ' + CONVERT(varchar,@Counter) + ' has an SalesOrderID of ' +
              CAST(@OrderID AS varchar) + ' and a CustomerID of ' + CAST(@CustomerID AS
varchar)
    END

    CLOSE CursorTest
    DEALLOCATE CursorTest
```

Chapter 15

The big differences are:

- ❑ The cursor is declared with the SCROLL option.
- ❑ We added a new navigation keyword—PRIOR—in the place of NEXT.
- ❑ We went ahead and closed and deallocated the cursor in the sproc rather than using an outside procedure (been there, done that).

The interesting part comes in the results. This one doesn't require the fancy test script—simply execute it:

```
EXEC spCursorScroll
```

And you'll see how the order values scroll forward and back:

```
Row 1 has a SalesOrderID of 43659 and a CustomerID of 676
Row 2 has a SalesOrderID of 43660 and a CustomerID of 117
Row 3 has a SalesOrderID of 43661 and a CustomerID of 442
Row 4 has a SalesOrderID of 43662 and a CustomerID of 227
Row 5 has a SalesOrderID of 43663 and a CustomerID of 510
Row 6 has a SalesOrderID of 43664 and a CustomerID of 397
Row 5 has an SalesOrderID of 43663 and a CustomerID of 510
Row 4 has an SalesOrderID of 43662 and a CustomerID of 227
Row 3 has an SalesOrderID of 43661 and a CustomerID of 442
Row 2 has an SalesOrderID of 43660 and a CustomerID of 117
Row 1 has an SalesOrderID of 43659 and a CustomerID of 676
```

As you can see, we were able to successfully navigate not only forward, as we did before, but also backward.

A forward-only cursor is far and away the more efficient choice of the two options. Think about the overhead for a moment—if it is read-only, then SQL Server really needs to keep track of the next record only—à la a linked list. In a situation where you may reposition the cursor in other ways, extra information must be stored in order to reasonably seek out the requested row. How exactly this is implemented depends on the specific cursor options you choose.

Some types of cursors imply scrollability; others do not. Some types of cursors are sensitive to changes in the data, and some are not. We'll look at some of these issues in the next section.

Cursor Types

The various APIs generally break cursors into four types:

- ❑ Static
- ❑ Keyset driven
- ❑ Dynamic
- ❑ Forward-only

How exactly these four types are implemented (and what, they're called) will sometimes vary slightly among the various APIs and object models, but their general nature is usually pretty much the same.

What makes the various cursor types different is their ability to be scrollable and their *sensitivity* to changes in the database over the life of the cursor. We've already seen what scrollability is all about, but the term "sensitivity" probably sounds like something you'd be more likely to read in *Men Are from Mars, Women Are from Venus* than in a programming book. Actually though, the concept of sensitivity is a rather critical one to think about when choosing your cursor type.

Whether a cursor is sensitive or not defines whether it notices changes in the database or not after the cursor is opened. It also defines just what it does about it once the change is detected. Let's look at this in its most extreme versions—static versus dynamic cursors. The static cursor, once created, is absolutely oblivious to any change to the database. The dynamic cursor, however, is effectively aware of every change (inserted records, deletions, updates, you name it) to the database as long as the cursor remains open. We'll explore the sensitivity issue as we look at each of the cursor types.

Static Cursors

A static cursor is one that represents a "snapshot" in time. Indeed, at least one of the data access object models refers to it as a snapshot recordset rather than a static one.

When a static cursor is created, the entire recordset is created in what amounts to a temporary table in `tempdb`. After the time that it's created, a static cursor changes for no one and nothing. That is, it is set in stone. Some of the different object models will let you update information in a static cursor, some won't, but the bottom line is always the same: you cannot write updates to the database via a static cursor.

Before we get too far into this brand of cursor, I'm going to go ahead and tell you that the situations where it makes sense to use a static cursor on the server side are extremely rare. I'm not saying they don't exist—they do—but they are very rare indeed.

Before you get into the notion of using a static cursor on the server side, ask yourself:

- Can I do this with a temporary table?
- Can I do this entirely on the client side?

Remember that a static cursor is kept by SQL Server in a private table in `tempdb`. If that's how SQL Server is going to be storing it anyway, why not just use a temporary table yourself? There are times when that won't give you what you need (record rather than set operations). However, if you are just after the concept of a snapshot in time, rather than record-based operations, build your own temp table using `SELECT INTO` and save yourself (and SQL Server) a lot of overhead.

If you're working in a client-server arrangement, static cursors are often better dealt with on the client side. By moving the entire operation to the client, you can cut the number of network roundtrips to the server substantially. Since you know that your cursor isn't going to be affected by changes to the database (after all, isn't that why you chose a static cursor in the first place), there's no reason to make contact with the server again regarding the cursor after it is created.

Chapter 15

Okay, so let's move on to an example of a static cursor. What we're going to do in this example is play around with the notion of creating a static cursor, then make changes and see what happens. We'll play with variations of this throughout the remainder of this part of the chapter as we look at each cursor type.

We'll start with building a table to test with, and then we'll build our cursor and manipulate it to see what's in it.

```
USE AdventureWorks
/* Build the table that we'll be playing with this time */
SELECT SalesOrderID, CustomerID
INTO CursorTable
FROM Sales.SalesOrderHeader
WHERE SalesOrderID BETWEEN 43661 AND 43665

-- Declare our cursor
DECLARE CursorTest CURSOR
GLOBAL          -- So we can manipulate it outside the batch
SCROLL          -- So we can scroll back and see the changes
STATIC          -- This is what we're testing this time
FOR
SELECT SalesOrderID, CustomerID
FROM CursorTable

-- Declare our two holding variables
DECLARE @SalesOrderID      int
DECLARE @CustomerID       varchar(5)

-- Get the cursor open and the first record fetched
OPEN CursorTest
FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID

-- Now loop through them all
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT CAST(@SalesOrderID AS varchar) + ' ' + @CustomerID
    FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID
END

-- Make a change. We'll see in a bit that this won't affect the cursor.
UPDATE CursorTable
    SET CustomerID = -111
    WHERE SalesOrderID = 43663

-- Now look at the table to show that the update is really there.
SELECT SalesOrderID, CustomerID
FROM CursorTable

-- Now go back to the top. We can do this since we have a scrollable cursor
FETCH FIRST FROM CursorTest INTO @SalesOrderID, @CustomerID

-- And loop through again.
WHILE @@FETCH_STATUS=0
```

```

BEGIN
    PRINT CONVERT(varchar(5),@SalesOrderID) + ' ' + @CustomerID
    FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID
END

-- Now it's time to clean up after ourselves
CLOSE CursorTest

DEALLOCATE CursorTest

DROP TABLE CursorTable

```

Let's take a look at what this gets us (Note that I've switched over to the "Results in text" option to make it easier to see my result sets and my PRINT messages together):

```

(5 row(s) affected)
43661    442
43662    227
43663    510
43664    397
43665    146

(1 row(s) affected)
SalesOrderID CustomerID
-----
43661        442
43662        227
43663        -111
43664        397
43665        146

(5 row(s) affected)

43661    442
43662    227
43663    510
43664    397
43665    146

```

There are several things to notice about what happened during the run on this script.

- ❑ First, even though we had a result set open against the table, we were still able to perform the update. In this case, it's because we have a static cursor — once it was created, it was disconnected from the actual records and no longer maintains any locks.
- ❑ Second, although we can clearly see that our update did indeed take place in the actual table, it did not affect the data in our cursor. Again, this is because, once created, our cursor took on something of a life of its own — it is no longer associated with the original data in any way.
- ❑ Under the heading of "one more thing," you could also notice that we made use of a new argument to the `FETCH` keyword — this time we went back to the top of our result set by using `FETCH FIRST`.

Keyset-Driven Cursors

When we talk about keysets with cursors, we're not talking your local locksmith. Instead, we're talking about maintaining a set of data that uniquely identifies the entire row in the database.

Keyset-driven cursors have the following high points:

- ❑ They require a unique index to exist on the table in question.
- ❑ Only the keyset is stored in tempdb—not the entire dataset.
- ❑ They are sensitive to changes to the rows that are already part of the keyset, including the possibility that they have been deleted.
- ❑ They are, however, not sensitive to new rows that are added after the cursor is created.
- ❑ Keyset cursors can be used as the basis for a cursor that is going to perform updates to the data.

Given that it has a name of “keyset” and that I’ve already said that the keyset uniquely identifies each row, it probably doesn’t shock you in any way that you must have a unique index of some kind (usually a primary key, but it could also be any index that is explicitly defined as unique) to create the keyset from.

The keys are all stored in a private table in tempdb. SQL Server uses this key as a method to find its way back to the data as you ask for a specific row in the cursor. The point to take note of here is that the actual data is being fetched, based on the key, at the time that you issue the `FETCH`. The great part about this is that the data for that particular row is up to date as of when the specific row is fetched. The downside (or upside depending on what you’re using the cursor for) is that it uses the keyset that is already created to do the lookup. This means that once the keyset is created, that is all the rows that will be included in your cursor. Any rows that were added after the cursor was created—even if they meet the conditions of the `WHERE` clause in the `SELECT` statement—will not be seen by the cursor. The rows that are already part of the cursor can, depending on the cursor options you chose, be updated by a cursor operation.

Let’s modify our earlier script to illustrate the sensitivity issue when we are making use of keyset-driven cursors:

```
USE AdventureWorks
/* Build the table that we'll be playing with this time */
SELECT SalesOrderID, CustomerID
INTO CursorTable
FROM Sales.SalesOrderHeader
WHERE SalesOrderID BETWEEN 43661 AND 43665

-- Now create a unique index on it in the form of a primary key
ALTER TABLE CursorTable
    ADD CONSTRAINT PKCursor
        PRIMARY KEY (SalesOrderID)

/* The IDENTITY property was automatically brought over when
** we did our SELECT INTO, but I want to use my own SalesOrderID
** value, so I'm going to turn IDENTITY_INSERT on so that I
** can override the identity value.
*/
SET IDENTITY_INSERT CursorTable ON

-- Declare our cursor
```

```

DECLARE CursorTest CURSOR
GLOBAL                      -- So we can manipulate it outside the batch
SCROLL                      -- So we can scroll back and see the changes
KEYSET                       -- This is what we're testing this time
FOR
SELECT SalesOrderID, CustomerID
FROM CursorTable

-- Declare our two holding variables
DECLARE @SalesOrderID      int
DECLARE @CustomerID        varchar(5)

-- Get the cursor open and the first record fetched
OPEN CursorTest
FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID

-- Now loop through them all
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT CAST(@SalesOrderID AS varchar) + '    ' + @CustomerID
    FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID
END

-- Make a change. We'll see that does affect the cursor this time.
UPDATE CursorTable
    SET CustomerID = -111
    WHERE SalesOrderID = 43663

-- Now we'll delete a record so we can see how to deal with that
DELETE CursorTable
    WHERE SalesOrderID = 43664

-- Now Insert a record. We'll see that the cursor is oblivious to it.
INSERT INTO CursorTable
    (SalesOrderID, CustomerID)
VALUES
    (-99999, -99999)

-- Now look at the table to show that the changes are really there.
SELECT SalesOrderID, CustomerID
FROM CursorTable

-- Now go back to the top. We can do this since we have a scrollable cursor
FETCH FIRST FROM CursorTest INTO @SalesOrderID, @CustomerID

/* And loop through again.
** This time, notice that we changed what we're testing for.
** Since we have the possibility of rows being missing (deleted)
** before we get to the end of the actual cursor, we need to do
** a little bit more refined testing of the status of the cursor.
*/
WHILE @@FETCH_STATUS != -1
BEGIN
    IF @@FETCH_STATUS = -2
    BEGIN
        PRINT ' MISSING! It probably was deleted.'
    END
END

```

Chapter 15

```
END
ELSE
BEGIN
    PRINT CAST(@SalesOrderID AS varchar) + ' ' + CAST(@CustomerID AS
varchar)
    END
    FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID
END

-- Now it's time to clean up after ourselves
CLOSE CursorTest

DEALLOCATE CursorTest

DROP TABLE CursorTable
```

The changes aren't really all that remarkable. We've gone ahead and added the required unique index. I happened to choose to do it as a primary key since that's what matches up best with the table we got this information out of, but it also could have been a unique index without the primary key. We also added something to insert a row of data so we can clearly see that the keyset doesn't see the row in question.

Perhaps the most important thing that we've changed is the condition for the `WHILE` loop on the final run through the cursor. Technically speaking, we should have made this change to both loops, but there is zero risk of a deleted record the first time around in this example, and I wanted the difference to be visible right within the same script.

The change was made to deal with something new we've added — the possibility that we might get to a record only to find that it's now missing. More than likely, someone has deleted it.

Let's take a look then at the results we get after running this:

```
(5 row(s) affected)
43661    442
43662    227
43663    510
43664    397
43665    146

(1 row(s) affected)

(1 row(s) affected)

(1 row(s) affected)
SalesOrderID CustomerID
-----
-99999      -99999
43661        442
43662        227
43663        -111
```

```
43665      146
(5 row(s) affected)

43661      442
43662      227
43663      -111
MISSING! It probably was deleted.
43665      146
```

Okay, let's walk through the highlights here.

Everything starts out pretty much as it did before. We see the same five rows in the first result set as we did last time. We then see an extra couple of "affected by" messages — these are for the `INSERT`, `UPDATE`, and `DELETE` statements that we added. Next comes the second result set. It's at this point that things get a bit more interesting.

In this next result set, we see the actual results of our `UPDATE`, `INSERT` and `DELETE` statements. Just as we think we're done, `SalesOrderID` 43664 has been deleted, and a new order with the `SalesOrderID` of -99999 has been inserted. That's what's in the table, but things don't appear quite as cozy in the cursor.

The next (and final) result set tells the tale on some differences in the way that things are presented in the cursor versus actually re-running the query. As it happens, we have exactly five rows — just like we started out with and just like our `SELECT` statement showed are in the actual table. But that's entirely coincidental.

In reality, there are a couple of key differences between what the cursor is showing and what the table is showing. The first presents itself rather boldly — our result set actually knows that a record is missing. You see, the cursor continues to show the key position in the keyset, but, when it went to do the lookup on the data, the data wasn't there anymore. Our `@@FETCH_STATUS` was set to -2, and we were able to test for it and report it. The `SELECT` statement showed us what data was actually there without any remembrance of the record ever having been there. The `INSERT`, on the other hand, is an entirely unknown quantity to the cursor. The record wasn't there when the cursor was created, so the cursor has no knowledge of its existence — it doesn't show up in our result set.

Keyset cursors can be very handy for dealing with situations where you need some sensitivity to changes in the data, but don't need to know about every insert right up to the minute. They can, depending on the nature of the result set you're after and the keyset, also provide some substantial savings in the amount of data that has to be duplicated and stored into `tempdb` — this can have some favorable performance impacts for your overall server.

WARNING!!! If you define a cursor as being of type `KEYSET` but do so on a table with no unique index, then SQL Server will implicitly convert your cursor to be `STATIC`. The fact that the behavior gets changed would probably be enough to ruffle your feathers a bit, but it doesn't stop there — it doesn't tell you about it. That's right; by default you get absolutely no warning about this conversion. Fortunately, you can watch out for this sort of thing by using the `TYPE_WARNING` option in your cursor. We'll look at this option briefly toward the end of the chapter.

Dynamic Cursors

Don't you just wish that you could be on a quiz show and have them answer a question like, "What's so special about a dynamic cursor?" Hmm, then again, I suppose their pool of possible contestants would be small, but those that decided to go for it would probably have the answer right away, "They are *dynamic*—right?" Exactly.

Well, almost exactly. Dynamic cursors fall just short of what I would call dynamic in the sense that they won't proactively tell you about changes to the underlying data. What gets them close enough to be called dynamic is that they are sensitive toward all changes to the underlying data. Of course, like most things in life, all this extra power comes with a high price tag.

If you want inserted records to be added to the cursor—no problem. If you want updated rows to appear properly updated in the cursor—no problem. If you want deleted records to be removed from the cursor set—no problem (although you can't really tell that something's been deleted since you won't see the missing record that you saw with a keyset cursor type). If, however, you want to have concurrency—uh oh, big problem (you're holding things open longer, so collisions with other users are more likely). If you want this to be low overhead—uh oh, big problem again (you are effectively requerying with every `FETCH`). Yes, dynamic cursors can be the bane of your performance existence, but, hey, that's life isn't it?

The long and the short of it is that you usually should avoid dynamic cursors.

Why all the hype and hoopla? Well, in order to understand some of the impacts that a dynamic cursor can have, you just need to realize a bit about how they work. You see, with a dynamic cursor, your cursor is essentially rebuilt every single time you issue a `FETCH`. That's right, the `SELECT` statement that forms the basis of your query, complete with its associated `WHERE` clause is effectively re-run. Think about that when dealing with large data sets. It brings just one word to mind—ugly. Very ugly indeed.

One of the things I've been taught since the dawn of my RDBMS time is that dynamic cursors are performance pigs—I've found this not to always be the case. This seems to be particularly true when the underlying tables are not very large in size. If you think about it for a bit, you might be able to come up with why a dynamic cursor can actually be slightly faster in terms of raw speed.

My guess as to what's driving this is the use of `tempdb` for keyset cursors. While a lot more work has to be done with each `FETCH` in order to deal with a dynamic cursor, the data for the requery will often be completely in cache (depending on the sizing and loading of your system). This means the dynamic cursor gets to work largely from RAM. The keyset cursor, on the other hand, is stored in `tempdb`, which is on disk (that is, much, much slower) for most systems.

As your table size gets larger, there is more diverse traffic hitting your server, the memory allocated to SQL Server gets smaller, and the more that keyset-driven cursors are going to have something of an advantage over dynamic cursors. In addition, raw speed isn't everything—you really need to think about concurrency issues too (we will look at the options for concurrency in detail later in the chapter), which can be more problematic in dynamic cursors. Still, don't count out dynamic cursors for speed alone if you're dealing with a server-side cursor with small data sets.

Let's go ahead and re-run our last script with only one modification—the change from `KEYSET` to `DYNAMIC`:

```
USE AdventureWorks
/* Build the table that we'll be playing with this time */
SELECT SalesOrderID, CustomerID
INTO CursorTable
FROM Sales.SalesOrderHeader
WHERE SalesOrderID BETWEEN 43661 AND 43665

-- Now create a unique index on it in the form of a primary key
ALTER TABLE CursorTable
    ADD CONSTRAINT PKCursor
        PRIMARY KEY (SalesOrderID)

/* The IDENTITY property was automatically brought over when
** we did our SELECT INTO, but I want to use my own SalesOrderID
** value, so I'm going to turn IDENTITY_INSERT on so that I
** can override the identity value.
*/
SET IDENTITY_INSERT CursorTable ON

-- Declare our cursor
DECLARE CursorTest CURSOR
GLOBAL                      -- So we can manipulate it outside the batch
SCROLL                       -- So we can scroll back and see the changes
DYNAMIC                      -- This is what we're testing this time
FOR
SELECT SalesOrderID, CustomerID
FROM CursorTable

-- Declare our two holding variables
DECLARE @SalesOrderID      int
DECLARE @CustomerID       varchar(5)

-- Get the cursor open and the first record fetched
OPEN CursorTest
FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID

-- Now loop through them all
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT CAST(@SalesOrderID AS varchar) + ' ' + @CustomerID
    FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID
END

-- Make a change. We'll see that does affect the cursor this time.
UPDATE CursorTable
    SET CustomerID = -111
    WHERE SalesOrderID = 43663

-- Now we'll delete a record so we can see how to deal with that
DELETE CursorTable
    WHERE SalesOrderID = 43664

-- Now Insert a record. We'll see that the cursor is oblivious to it.
```

Chapter 15

```
INSERT INTO CursorTable
    (SalesOrderID, CustomerID)
VALUES
    (-99999, -99999)

-- Now look at the table to show that the changes are really there.
SELECT SalesOrderID, CustomerID
FROM CursorTable

-- Now go back to the top. We can do this since we have a scrollable cursor
FETCH FIRST FROM CursorTest INTO @SalesOrderID, @CustomerID

/* And loop through again.
** This time, notice that we changed what we're testing for.
** Since we have the possibility of rows being missing (deleted)
** before we get to the end of the actual cursor, we need to do
** a little bit more refined testing of the status of the cursor.
*/
WHILE @@FETCH_STATUS != -1
BEGIN
    IF @@FETCH_STATUS = -2
    BEGIN
        PRINT '    MISSING! It probably was deleted.'
    END
    ELSE
    BEGIN
        PRINT CAST(@SalesOrderID AS varchar) + '    ' + CAST(@CustomerID AS
varchar)
    END
    FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID
END

-- Now it's time to clean up after ourselves
CLOSE CursorTest

DEALLOCATE CursorTest

DROP TABLE CursorTable
```

And the results:

```
(5 row(s) affected)
```

```
43661    442
43662    227
43663    510
43664    397
43665    146
```

```
(1 row(s) affected)
```

```
(1 row(s) affected)
```

```
(1 row(s) affected)
```

```

SalesOrderID CustomerID
-----
-99999      -99999
43661       442
43662       227
43663       -111
43665       146

(5 row(s) affected)

-99999      *
43661       442
43662       227
43663       -111
43665       146

```

The first two recordsets look exactly as they did last time. The change comes when we get to the third (and final) result set:

- There is no indication of a failed fetch, even though we deleted a record (no notification).
- The updated record shows the update (just as it did with a keyset).
- The inserted record now shows up in the cursor set.

Dynamic cursors are the most sensitive of all cursors. They are affected by everything you do to the underlying data. The downside is that they can provide some extra concurrency problems, and they can pound the system when dealing with larger data sets.

Technically speaking, and unlike a keyset cursor, a dynamic cursor can operate on a non-unique index. Avoid this at all costs (in my opinion, it should prevent you from doing this and throw an error). Under certain circumstances, it is quite possible to create an infinite loop because the dynamic cursor cannot keep track of where it is in the cursor set. The only sure-fire way of avoiding this is to either stay away from dynamic cursors or only work on tables with a truly unique index available.

FAST_FORWARD Cursors

Fast (from a cursor standpoint — queries make this or any other cursor look like a snail) is the operative word on this one. This one is the epitome of the term “firehose cursor” that is often used around forward-only cursors. I’ve always taken the analogy to imply the way that the data sort of spews forth — once out, you can’t put it back in. In short, you’re simply awash with data. With **FAST_FORWARD** cursors, you open the cursor, and do nothing else but deal with the data, move forward, and deallocate it (note that I didn’t say close it).

Now, it’s safe to say that calling this a cursor “type” is something of a misnomer. This kind of cursor has several different circumstances where it is automatically converted to other cursor types, but I think of them as being most like a keyset-driven cursor in the sense that membership is fixed — once the members of the cursor are established, no new records are added. Deleted rows show up as a missing record (`@@FETCH_STATUS` of `-2`). Keep in mind though that, if the cursor is converted to something else (via automatic conversion), it will take on the behavior of that new cursor type.

The nasty side here is that SQL Server doesn't tell you that the conversion has happened unless you have the `TYPE_WARNING` option added to your cursor definition.

As I said before, there are a number of circumstances where a `FAST_FORWARD` cursor is implicitly converted to another cursor type. Below is a table that outlines these conversions:

Condition	Converted to
The underlying query requires that a temporary table be built	Static
The underlying query is distributed in nature	Keyset
The cursor is declared as <code>FOR UPDATE</code>	Dynamic
A condition exists that would convert to keyset driven, but at least one underlying table does not have a unique index	Static

I've heard that there are other circumstances where a cursor will be converted, but I haven't seen any documentation of this, and I haven't run into it myself.

If you find that you are getting that most dreadful of all computer-related terms (unpredictable results), you can make use of `sp_describe_cursor` (a system stored procedure) to list out all the currently active options for your cursor.

It's worth noting that all `FAST_FORWARD` cursors are read-only in nature. You can explicitly set the cursor to have the `FOR UPDATE` option, but, as suggested in the preceding implicit conversion table, the cursor will be implicitly converted to dynamic.

Okay, so what exactly does a `FAST_FORWARD` cursor have that any of the other cursors wouldn't have if they were declared as being `FORWARD_ONLY`? Well, a `FAST_FORWARD` cursor will implement at least one of two tricks to help things along:

- ❑ The first is to pre-fetch data. That is, at the same time that you open the cursor, it automatically fetches the first row—this means that you save a roundtrip to the server if you are operating in a client-server environment using ODBC. Unfortunately, this is available only under ODBC.
- ❑ The second is the one that is a sure thing—auto-closing of the cursor. Since you are running a cursor that is forward-only, SQL Server can assume that you want the cursor closed once you reach the end of the recordset. Again, this saves a roundtrip and squeezes out a tiny bit of additional performance.

Choosing a cursor type is one of the most critical decisions when structuring a cursor. Choices that have little apparent difference in the actual output of the cursor task can have major differences in performance. Other affects can be seen in sensitivity to changes, concurrency issues, and updatability.

Concurrency Options

We got our first taste of concurrency issues back in our chapter on transactions and locks. As you recall, we deal with concurrency issues whenever there are issues surrounding two or more processes trying to get to the same data at essentially the same time. When dealing with cursors, however, the issue becomes just slightly stickier.

The problem is multi-fold:

- The operation tends to last longer (more time to have a concurrency problem).
- Each row is read at the time of the fetch, but someone may try to edit it before you get a chance to do your update.
- You may scroll forward and backward through the result set for what could be an essentially unlimited amount of time (I hope you never do that, but it's possible to do).

As with all concurrency issues, this tends to be more of a problem in a transaction environment than when running in a single statement situation. The longer the transaction, the more likely you are to have concurrency problems.

SQL Server gives us three different options for dealing with this issue:

- READ_ONLY
- SCROLL_LOCKS (equates to Pessimistic in most terminologies)
- OPTIMISTIC

Each of these has their own thing they bring to the party, so let's look at them one by one.

READ_ONLY

In a read-only situation, you don't have to worry about whether your cursor is going to try and obtain any kind of update or exclusive lock. You also don't have to worry about whether anyone has edited the data while you've been busy making changes of your own. Both of these make life considerably easier.

READ_ONLY is just what it sounds like. When you choose this option, you cannot update any of the data, but you also skip most (but not all) of the notion of concurrency entirely.

SCROLL_LOCKS

Scroll locks equate to what is more typically referred to as pessimistic locking in the various APIs and object models. In its simplest form, it means that, as long as you are editing this record, no one else is allowed to edit it. The specifics of implementation of duration of this vary depending on:

- Whether you're in a transaction or not
- What transaction isolation level you've set

Chapter 15

Note that this can be different from what we saw with update locks back in our locking and transaction chapter.

With update locks, we prevented other users from updating the data. This lock was held for the duration of the transaction. If it was a single statement transaction, then the lock was not released until every row affected by the update was complete.

Scroll locks work identically to update locks with only one significant exception—the duration the lock is held. With scroll locks, there is much more of a variance depending on whether the cursor is participating in a multi-statement transaction or not. Assuming for the moment that you do not have a transaction wrapped around the cursor, then the lock is held only on the current record in the cursor—that is, from the time the record is first fetched until the next record (or end of the result set) is fetched. Once you move on to the next record, the lock is removed from the prior record.

Let's take a look at this through a significantly pared down version of the script we've been using through much of this chapter:

```
USE AdventureWorks
/* Build the table that we'll be playing with this time */
SELECT SalesOrderID, CustomerID
INTO CursorTable
FROM Sales.SalesOrderHeader
WHERE SalesOrderID BETWEEN 43661 AND 43665

-- Now create a unique index on it in the form of a primary key
ALTER TABLE CursorTable
ADD CONSTRAINT PKCursor
PRIMARY KEY (SalesOrderID)

/* The IDENTITY property was automatically brought over when
** we did our SELECT INTO, but I want to use my own SalesOrderID
** value, so I'm going to turn IDENTITY_INSERT on so that I
** can override the identity value.
*/
SET IDENTITY_INSERT CursorTable ON

-- Declare our cursor
DECLARE CursorTest CURSOR
GLOBAL          -- So we can manipulate it outside the batch
SCROLL          -- So we can scroll back and see the changes
DYNAMIC         -- This is what we're testing this time
SCROLL_LOCKS
FOR
SELECT SalesOrderID, CustomerID
FROM CursorTable

-- Declare our two holding variables
DECLARE @SalesOrderID      int
```

```
DECLARE @CustomerID  varchar(5)

-- Get the cursor open and the first record fetched
OPEN CursorTest
FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID
```

You'll not see much of our usual gray in the preceding code block (to indicate that changes were made on that line) because only one line was added. The remainder of the changes were deletions of lines, so there's nothing for me to make gray for you. Just make sure that you've made the appropriate changes if you're going to try and run this one.

What we've done is toss out most of the things that were happening, and we've refocused ourselves back on the cursor. Perhaps the biggest thing to notice though is a couple of key things that we have deliberately omitted even though they are things that would normally cause problems if we try to operate without them:

- We do not have a `CLOSE` on our cursor, nor do we deallocate it at this point.
- We don't even scroll any farther than getting the first row fetched.

The reason we've left the cursor open is to create a situation where the state of the cursor being open lasts long enough to play around with the locks somewhat. In addition, we fetch only the first row because we want to make sure that there is an active row (the way we had things before, we would have been to the end of the set before we started running with other, possibly conflicting, statements).

What you want to do is execute the preceding and then open a completely separate connection window with AdventureWorks active. Then run a simple test in the new connection window:

```
SELECT * FROM CursorTable
```

If you haven't been grasping what I've been saying in this section, you might be a tad surprised by the results:

SalesOrderID	CustomerID
43661	442
43662	227
43663	510
43664	397
43665	146

(5 row(s) affected)

Based on what we know about locks (from Chapter 12), you would probably expect the preceding `SELECT` statement to be blocked by the locks on the current record. Not so with scroll locks. The lock is only on the record that is currently in the cursor, and, perhaps more importantly, the lock only prevents updates to the record. Any `SELECT` statements (such as ours) can see the contents of the cursor without any problems.

Chapter 15

Now that we've seen how things work, go back to the original window and run the code to clean things up. This is back to the same code we've worked with for much of this chapter:

```
-- Now it's time to clean up after ourselves  
CLOSE CursorTest  
  
DEALLOCATE CursorTest  
  
DROP TABLE CursorTable
```

Don't forget to run the preceding clean up code!!! If you forget, then you'll have an open transaction sitting in your system until you terminate the connection. SQL Server should clean up any open transactions (by rolling them back) when the connection is broken, but I've seen situations where you run the database consistency checker (DBCC) and find that you have some really old transactions — SQL Server missed cleaning up after itself.

OPTIMISTIC

Optimistic locking creates a situation where no scroll locks of any kind are set on the cursor. The assumption is that, if you do an update, you want people to still be able to get at your data. You're being optimistic because you are essentially guessing (hoping may be a better word) that no one will edit your data between when you fetched it into the cursor and when you applied your update.

The optimism is not necessarily misplaced. If you have a lot of records and not that many users, then the chances of two people trying to edit the same record at the same time are very small (depending on the nature of your business processes). Still, if you get this optimistic, then you need to also be prepared for the possibility that you will be wrong — that is, that someone has altered the data in between when you performed the fetch and when you went to actually update the database.

If you happen to run into this problem, SQL Server will issue an error with a value in @@ERROR of 16394. When this happens, you need to completely re-fetch the data from the cursor (so you know what changes were being made) and either rollback the transaction or try the update again.

Detecting Conversion of Cursor Types: TYPE_WARNING

This one is really pretty simple. If you add this option to your cursor, then you will be notified if an implicit conversion is made on your cursor. Without this statement, the conversion just happens with no notification. If the conversion wasn't an anticipated behavior, then there's a good chance that you're going to see the most dreaded of all computer terms (unpredictable results).

This is perhaps best understood with an example, so let's go back and run a variation again of the cursor that we've been using throughout most of the chapter.

In this instance, we're going to take out the piece of code that creates a key for the table. Remember that without a unique index on a table, a keyset will be implicitly converted to a static cursor:

```
USE AdventureWorks  
  
/* Build the table that we'll be playing with this time */
```

```
SELECT OrderID, CustomerID
INTO CursorTable
FROM Orders
WHERE OrderID BETWEEN 10701 AND 10705

-- Declare our cursor
DECLARE CursorTest CURSOR
GLOBAL          -- So we can manipulate it outside the batch
SCROLL          -- So we can scroll back and see the changes
KEYSET
TYPE_WARNING
FOR
SELECT OrderID, CustomerID
FROM CursorTable

-- Declare our two holding variables
DECLARE @OrderID      int
DECLARE @CustomerID   varchar(5)

-- Get the cursor open and the first record fetched
OPEN CursorTest
FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID

-- Now loop through them all
WHILE @@FETCH_STATUS=0
BEGIN
    PRINT CONVERT(varchar(5),@OrderID) + ' ' + @CustomerID
    FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID
END

-- Now it's time to clean up after ourselves
CLOSE CursorTest

DEALLOCATE CursorTest

DROP TABLE CursorTable
```

There's nothing particularly special about this one. I'm considering it to be something of a complete rewrite only because we've deleted so much from the original and it's been so long since we've seen it. The creation of the table and cursor is pretty much the same as when we did our keyset-driven cursor much earlier in the chapter. The major changes are the removal of blocks of code that we don't need for this illustration along with the addition of the `TYPE_WARNING` option in the cursor declaration.

Now we come up with some interesting results:

```
(5 row(s) affected)
The created cursor is not of the requested type.
43661    442
43662    227
43663    510
43664    397
43665    146
```

Chapter 15

Everything ran okay—we just saw a statement that was meant solely as a warning. The results may not be what you expected given that the cursor was converted.

The downside here is that you get a message sent out, but no error. Programmatically speaking, there is essentially no way to tell that you received this message—which makes this option fairly useless in a production environment. Still, it can often be quite handy when you’re trying to debug a cursor to determine why it isn’t behaving in the expected fashion.

FOR <SELECT>

This section of the cursor declaration is at the very heart of the matter. This is a section that is required under even the most basic of cursor syntax, and that’s because it’s the one and only clause that determines what data should be placed in the cursor.

Almost any SELECT statement is valid—even those including an ORDER BY clause. As long as your SELECT statement provides a single result set, you should be fine. Examples of options that would create problems would be any of the summary options such as a CUBE or ROLLUP.

FOR UPDATE

By default, any cursor that is updatable at all is completely updatable—that is, if one column can be edited then any of them can.

The FOR UPDATE <column list> option allows you to specify that only certain columns are to be editable within this cursor. If you include this option, then only the columns in your column list will be allowed to be updatable. Any columns not explicitly mentioned will be considered to be read-only.

Navigating the Cursor: The FETCH Statement

I figure that whoever first created the SQL cursor syntax must have really liked dogs. They probably decided to think of the data they were after as being the bone, with SQL Server the faithful bloodhound. From this I’m guessing, the FETCH keyword was born.

It’s an apt term if you think about it. In a nutshell, it tells SQL Server to “go get it boy!” With that, our faithful mutt (in the form of SQL Server) is off to find the particular bone (row) we were after. We’ve gotten a bit of a taste of the FETCH statement in some of the previous cursors in this chapter, but it’s time to look at this very important statement more closely.

FETCH actually has many more options than what we’ve seen so far. Up to this point, we’ve seen three different options for FETCH (NEXT, PREVIOUS, and FIRST). These really aren’t a bad start. Indeed, we really only need to add one more for the most basic set of cursor navigation commands, and a few after that for the complete set.

Let's look at each of the cursor navigation commands and see what they do for us:

FETCH Option	Description
NEXT	This moves you forward exactly one row in the result set and is the backbone option. Ninety percent or more of your cursors won't need any more than this. Keep this in mind when deciding to declare as FORWARD_ONLY or not. When you try to do a <code>FETCH NEXT</code> and it results in moving beyond the last record, you will have a <code>@@FETCH_STATUS</code> of -1.
PRIOR	As you have probably surmised, this one is the functional opposite of <code>NEXT</code> . This moves backward exactly one row. If you performed a <code>FETCH PRIOR</code> when you were at the first row in the result set, then you will get a <code>@@FETCH_STATUS</code> of -1 just as if you had moved beyond the end of the file.
FIRST	Like most cursor options, this one says what it is pretty clearly. If you perform a <code>FETCH FIRST</code> , then you will be at the first record in the recordset. The only time this option should generate a <code>@@FETCH_STATUS</code> of -1 is if the result set is empty.
LAST	The functional opposite of <code>FIRST</code> , <code>FETCH LAST</code> moves you to the last record in the result set. Again, the only way you'll get a -1 for <code>@@FETCH_STATUS</code> on this one is if you have an empty result set.
ABSOLUTE	With this one, you supply an integer value that indicates how many rows you want from the beginning of the cursor. If the value supplied is negative, then it is that many rows from the end of the cursor. Note that this option is not supported with dynamic cursors (since the membership in the cursor is redone with every fetch, you can "really know where you're at"). This equates roughly to navigating to a specific "absolute position" in a few of the client access object models.
RELATIVE	No — this isn't your mother-in-law kind of thing. Instead, this is about navigating by moving a specified number of rows forward or backward relative to the current row.

We've already gotten a fair look at a few of these in our previous cursors. The other navigational choices work pretty much the same.

Altering Data within Your Cursor

Up until now, we've kind of glossed over the notion of changing data directly in the cursor. Now it's time to take a look at updating and deleting records within a cursor.

Since we're dealing with a specific row rather than set data, we need some special syntax to tell SQL Server that we want to update. Happily, this syntax is actually quite easy given that you already know how to perform an `UPDATE` or `DELETE`.

Chapter 15

Essentially, we're going to update or delete data in the table that is underlying our cursor. Doing this is as simple as running the same UPDATE and DELETE statements that we're now used to, but qualifying them with a WHERE clause that matches our cursor row. We just add one line of syntax to our DELETE or UPDATE statement:

```
WHERE CURRENT OF <cursor name>
```

Nothing remarkable about it at all. Just for grins though, we'll go ahead and implement a cursor using this syntax:

```
USE AdventureWorks
/* Build the table that we'll be playing with this time */
SELECT SalesOrderID, CustomerID
INTO CursorTable
FROM Sales.SalesOrderHeader
WHERE SalesOrderID BETWEEN 43661 AND 43665

-- Now create a unique index on it in the form of a primary key
ALTER TABLE CursorTable
    ADD CONSTRAINT PKCursor
        PRIMARY KEY (SalesOrderID)

/* The IDENTITY property was automatically brought over when
** we did our SELECT INTO, but I want to use my own OrderID
** value, so I'm going to turn IDENTITY_INSERT on so that I
** can override the identity value.
*/
SET IDENTITY_INSERT CursorTable ON

-- Declare our cursor
DECLARE CursorTest CURSOR
SCROLL           -- So we can scroll back and see if the changes are there
KEYSET
FOR
SELECT SalesOrderID, CustomerID
FROM CursorTable

-- Declare our two holding variables
DECLARE @SalesOrderID      int
DECLARE @CustomerID       varchar(5)

-- Get the cursor open and the first record fetched
OPEN CursorTest
FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID

-- Now loop through them all
WHILE @@FETCH_STATUS=0
BEGIN
    IF (@SalesOrderID % 2 = 0)      -- Even number, so we'll update it
```

```

BEGIN
    -- Make a change. This time though, we'll do it using cursor syntax
    UPDATE CursorTable
        SET CustomerID = -99999
        WHERE CURRENT OF CursorTest
END

ELSE                                -- Must be odd, so we'll delete it.
BEGIN
    -- Now we'll delete a record so we can see how to deal with that
    DELETE CursorTable
        WHERE CURRENT OF CursorTest
END
FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID
END

-- Now go back to the top. We can do this since we have a scrollable cursor
FETCH FIRST FROM CursorTest INTO @SalesOrderID, @CustomerID

-- And loop through again.
WHILE @@FETCH_STATUS != -1
BEGIN
    IF @@FETCH_STATUS = -2
    BEGIN
        PRINT ' MISSING! It probably was deleted.'
    END
    ELSE
    BEGIN
        PRINT CAST(@SalesOrderID AS varchar) + ' ' + CAST(@CustomerID AS
varchar)
    END
    FETCH NEXT FROM CursorTest INTO @SalesOrderID, @CustomerID
END

-- Now it's time to clean up after ourselves
CLOSE CursorTest

DEALLOCATE CursorTest

DROP TABLE CursorTable

```

Again, I'm treating this one as an entirely new cursor. We've done enough deletions, additions, and updates that I suspect you'll find it easier to just key things in a second time rather than having to look through row by row to see what you might have missed.

We are also again using the modulus operator (%) that we saw earlier in the book. Remember that it gives us nothing but the remainder. Therefore, if the remainder of any number divided by 2 is zero, then we know the number was an even number.

Chapter 15

The rest of the nuts and bolts of this don't require any rocket science, yet we can quickly tell that we got some results:

```
(5 row(s) affected)

(1 row(s) affected)
    MISSING! It probably was deleted.
43662 *
    MISSING! It probably was deleted.
43664 *
    MISSING! It probably was deleted.
```

You can see the multiple “1 row affected” that is the returned message for any row that was affected by the UPDATE and DELETE statements. When we get down to the last result set enumeration, you can quickly tell that we deleted all the odd numbers (which is what we told our code to do), and that we updated the even numbered rows with a new CustomerID.

No tricks—just a WHERE clause that makes use of the WHERE CURRENT argument.

Summary

Cursors give us those memories of the old days—when we could address things row by row. Ahhh, it sounds so romantic with that “old days” kind of thought. WRONG! I’d stick to set operations any day if I thought I could get away with it.

The fact is that set operations can’t do everything. Cursors are going to be the answer any time a solution must be done on a row-by-row basis. Notice that I used the word “must” in there, and that’s the way you should think of it. Cursors are great of taking care of some problems that can’t be solved for any other means.

That being said, remember to avoid cursor use wherever possible. Cursors are resource pigs and will almost always produce 100 times or worse negative performance impact. It is extremely tempting—especially if you come from the mainframe world or from a dBase background—to just keep thinking in that row-by-row method. Don’t fall into that trap! Cursors are meant to be used only when no other options are available.

16

XML Integration

Extensible Markup Language (XML)—looking back at its history is something of a funny thing to me. Part of its strength lies in its simplicity, so it would seem like it wouldn't change much. Indeed, the basic rules of it haven't changed at all—but all the things surrounding XML (such as how to access data stored in XML) have gone through many changes. Likewise, the way that SQL Server supports XML has seen some fairly big changes from the time it was first introduced to the relatively massive support seen in SQL Server 2005.

So, to continue my “it's a funny thing” observation, I realize that as recently as when I did the *Beginning* edition for SQL Server 2005 I referred to XML support as being an “extra”—what a truly silly thing for me to say. Yeah, yeah, yeah—I've tempered that “extra” comment with the notion that it's only because XML support isn't really required to have a working SQL Server, but I've come to realize in today's world that it isn't much of a working SQL Server without support for XML. Indeed, as we continue through the rest of what I have generally referred to as “extras” in the past (Reporting Services, Integration Services, Connectivity and more), you'll see that even SQL Server stores many of the definition objects for these extra services using XML!

So, with all that said, in this chapter we'll look at:

- ❑ The XML data type
- ❑ XML schema collections
- ❑ Methods of representing your relational data as XML
- ❑ Methods of querying data that we have stored natively in XML (XQuery, Microsoft's “XDL” language, and other methods)
- ❑ XML indexes
- ❑ HTTP endpoints and SOAP support

Some of these are actually embedded within each other, so let's get to taking a look so we can see how they mix.

This chapter assumes that you have an existing knowledge of at least basic XML rules and constructs. If you do not have that foundation knowledge, I strongly recommend picking up a copy of a Wrox book like *Beginning XML, 3rd Edition*, by David Hunter, et al. (Wiley 2005) or another XML-specific book before getting too far into this chapter.

The XML Data Type

The addition of the XML data type signals the end of what has been, to me, a rather long road. For the first time, SQL Server takes data that is in XML format and recognizes it as truly being XML data. In previous versions, there were an increasing number of ways to address XML data, but all of it was done from the foundation of basic character data. XML now recognizes XML as XML and that opens up a host of new possibilities from indexing to data validation.

The number of different things going on here is massive. Among the various things that we need to talk about when discussing the XML data type include:

- ❑ **Schema collections**—A core concept of XML is the notion of allowing XML to be associated with schema documents. XML schemas define the rules that allow us to determine whether our XML is “valid” (that is, does it meet the rules for that this particular kind of XML document is supposed to do). XML schema collections in SQL Server are a way of storing schemas and allowing SQL Server to know that is what they are—validation documents. You can associate instances of XML data (column data or variables for example) with XML schemas, and SQL Server will apply the schema to each instance of that XML to determine whether it is valid XML or not.
- ❑ **Enforcing constraints**—We’ve already dealt with the idea of requiring a column to meet certain criteria before we’ll let it into our table, but what about XML? XML allows for multiple pieces of discrete data to be stored within just one column—how do we validate those individual pieces of data? The XML data type understands XML, and, while direct definition of constraints is not allowed, we can utilize wrapper functions to define constraints for specific nodes within our XML.
- ❑ **XML data type methods**—When referring to a column or variable that is typed XML, you can utilize several methods that are intrinsic to that data type. For example, you can test for the existence of a certain node or attribute, execute XDL (a Microsoft-defined extension to XQuery that allows for data modification), or query the value of a specific node or attribute.

Let’s get more specific.

Defining a Column as Being of XML Type

We’ve already seen the most basic definition of an XML column. For example, if we examined the most basic definition of the `Production.ProductModel` table in the AdventureWorks database, it would look something like this:

```
CREATE TABLE Production.ProductModel
(
    ProductModelID      int IDENTITY(1,1) PRIMARY KEY NOT NULL,
    Name                dbo.Name NOT NULL,
    CatalogDescription  xml NULL,
    Instructions        xml NULL,
    ModifiedDate        datetime NOT NULL
        CONSTRAINT DF_ProductModel_ModifiedDate DEFAULT (GETDATE()),
)
```

So, let's ask ourselves what we have here in terms of our two XML columns.

1. We have defined them as XML, so we will have our XML data type methods available to us (more on those coming up soon).
2. We have allowed NULLs but could have just as easily chosen NOT NULL as a constraint. Note, however, that the NOT NULL would be enforced on whether the row had any data for that column, not whether that data was valid.
3. Our XML is considered "non-typed XML." That is, since we have not associated any schema with it, SQL Server doesn't really know anything about how this XML is supposed to behave to be considered "valid."

The first of these is implied in any column that is defined with the data type XML rather than just plain text. We will see much more about this in our next XML data type section.

The second goes with any data type in SQL Server — we can specify whether we allow NULL data or not for that column.

So, the real meat in terms changes we can make at definition time has to do with whether we specify our XML column as being typed or non-typed XML. The non-typed definition we used in the preceding example means that SQL Server knows very little about any XML stored in the column and, therefore, can do little to police its validity. If we set the column up as being typed XML, then we are providing much more definition about what is considered "valid" for any XML that goes in our column.

The AdventureWorks database already has schema collections that match the validation we want to place on our two XML columns, so let's look at how we would change our CREATE statement to adjust to typed XML:

```
CREATE TABLE Production.ProductModel
(
    ProductModelID      int IDENTITY(1,1) PRIMARY KEY NOT NULL,
    Name                dbo.Name NOT NULL,
    CatalogDescription  xml
        (CONTENT [Production].[ProductDescriptionSchemaCollection]) NULL,
    Instructions        xml
        (CONTENT [Production].[ManuInstructionsSchemaCollection]) NULL,
    ModifiedDate        datetime NOT NULL
        CONSTRAINT DF_ProductModel_ModifiedDate DEFAULT (GETDATE())
)
```

This represents the way it is defined in the actual AdventureWorks sample. In order to insert a record into the Production.ProductModel table, you must either leave the CatalogDescription and Instructions fields blank or supply XML that is valid when tested against their respective schemas.

XML Schema Collections

XML schema collections are really nothing more than named persistence of one or more schema documents into the database. The name amounts to a handle to your set of schemas. By referring to that collection, you are indicating that the XML typed column or variable must be valid when matched against all of the schemas in that collection.

We can view existing schema collections. To do this, we utilize the built-in `XML_SCHEMA_NAMESPACE()` function. The syntax looks like this:

`XML_SCHEMA_NAMESPACE(<SQL Server schema> , <xml schema collection> , [<namespace>])`

This is just a little confusing, so let's touch on these parameters just a bit:

Parameter	Description
SQL Server schema	This is your relational database schema (not to be confused with the XML schema). For example, for the table <code>Production.ProductModel</code> , <code>Production</code> is the relational schema. For <code>Sales.SalesOrderHeader</code> , <code>Sales</code> is the relational schema.
xml schema collection	The name used when the XML schema collection was created. In your create table example previously, you referred to the <code>ProductDescriptionSchemaCollection</code> and <code>ManuInstructionSSchemaCollection</code> XML schema collections.
namespace	Optional name for a specific namespace within the XML schema collection. Remember that XML schema collections can contain multiple schema documents — this would return anything that fell within the specified namespace.

So, to use this for the `Production.ManuInstructionsSchemaCollection` schema collection, we would make a query like this:

```
SELECT XML SCHEMA NAMESPACE('Production', 'ManuInstructionsSchemaCollection')
```

This spews forth a ton of unformatted XML:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:t="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelManuInstructions"
targetNamespace="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelManuInstructions" elementFormDefault="qualified"><xsd:element
name="root"><xsd:complexType mixed="true"><xsd:complexContent mixed="true">
<xsd:restriction base="xsd:anyType"><xsd:sequence><xsd:element name="Location"
maxOccurs="unbounded"><xsd:complexType mixed="true"><xsd:complexContent
mixed="true"><xsd:restriction base="xsd:anyType"><xsd:sequence><xsd:element
name="step" type="t:StepType" maxOccurs="unbounded" /></xsd:sequence><xsd:attribute
name="LocationID" type="xsd:integer" use="required" /><xsd:attribute
name="SetupHours" type="xsd:decimal" /><xsd:attribute name="MachineHours"
type="xsd:decimal" /><xsd:attribute name="LaborHours" type="xsd:decimal"
/><xsd:attribute name="LotSize" type="xsd:decimal" /></xsd:restriction>
```

```
</xsd:complexContent></xsd:complexType></xsd:element></xsd:sequence></xsd:restriction>
</xsd:complexContent></xsd:complexType></xsd:element><xsd:complexType name="StepType" mixed="true"><xsd:complexContent mixed="true"><xsd:restriction base="xsd:anyType">
<xsd:choice minOccurs="0" maxOccurs="unbounded"><xsd:element name="tool" type="xsd:string" /><xsd:element name="material" type="xsd:string" /><xsd:element name="blueprint" type="xsd:string" /><xsd:element name="specs" type="xsd:string" /><xsd:element name="diag" type="xsd:string" /></xsd:choice></xsd:restriction>
</xsd:complexContent></xsd:complexType></xsd:schema>
```

SQL Server strips out any whitespace between tags, so if you create a schema collection with all sorts of pretty indentations for readability, SQL Server will remove them for the sake of efficient storage.

Note that the default number of characters returned for text results in Management Studio is only 256 characters. If you're using text view, you will want to go Tools⇒ Options⇒Query Results⇒SQL Server⇒Results to Text and change the maximum number of characters displayed.

Creating, Altering, and Dropping XML Schema Collections

The CREATE, ALTER, and DROP notions for XML schema collections work in a manner that is *mostly* consistent with how other such statements have worked thus far in SQL Server. We'll run through them here, but pay particular attention to the ALTER statement, as it is the one that has a few quirks versus other ALTER statements we've worked with.

CREATE XML SCHEMA COLLECTION

Again, the CREATE is your typical CREATE <object type> <object name> syntax that we've seen throughout the book, and uses the AS keyword we've seen with stored procedures, views, and other less structured objects:

```
CREATE XML SCHEMA COLLECTION [<SQL Server schema>.] <collection name>
AS { <schema text> | <variable containing the schema text> }
```

So if, for example, we wanted to create an XML schema collection that is similar to the Production.ManuInstructionsSchemaCollection collection in AdventureWorks, we might execute something like the following:

```
CREATE XML SCHEMA COLLECTION ProductDescriptionSchemaCollectionSummaryRequired AS
'<xsd:schema
targetNamespace="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain"
xmlns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain"
elementFormDefault="qualified"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
<xsd:element name="Warranty" >
<xsd:complexType>
<xsd:sequence>
<xsd:element name="WarrantyPeriod" type="xsd:string" />
```

```
        <xsd:element name="Description" type="xsd:string" />
    </xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
<xs:schema
targetNamespace="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription"
xmlns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription"
elementFormDefault="qualified"
xmlns:mstns="http://tempuri.org/XMLSchema.xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:wm="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain" >
    <xs:import
namespace="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain" />
    <xs:element name="ProductDescription" type="ProductDescription" />
    <xs:complexType name="ProductDescription">
        <xs:sequence>
            <xs:element name="Summary" type="Summary" minOccurs="1" />
        </xs:sequence>
        <xs:attribute name="ProductModelID" type="xs:string" />
        <xs:attribute name="Product modelName" type="xs:string" />
    </xs:complexType>
    <xs:complexType name="Summary" mixed="true" >
        <xs:sequence>
            <xs:any processContents="skip"
namespace="http://www.w3.org/1999/xhtml" minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>
</xs:schema>'
```

Note that the URL portion of the namespace declaration must be entered on a single line. They are shown here word wrapped onto multiple lines because there is a limit to how many characters we can show per line in print. Make sure you include the entire URL on a single line.

This one happens to be just like the Production.ManuInstructionsSchemaCollection schema collection, but I've altered the schema to require the summary element rather than having it optional. Since the basic structure is the same, I utilized the same namespaces.

ALTER XML SCHEMA COLLECTION

This one is just slightly different from other ALTER statements in the sense that it is limited to just adding new pieces to the collection. The syntax looks like this:

```
ALTER XML SCHEMA COLLECTION [<SQL Server schema>.] <collection name>
    ADD { <schema text> | <variable containing the schema text> }
```

I would not be at all surprised if the functionality of this is boosted a bit in a later service pack, but, in the meantime, let me stress again that this is a tool for adding to your schema collection rather than changing or removing what's there.

DROP XML SCHEMA COLLECTION

This is one of those classic “does what it says” things and works just like any other DROP:

```
DROP XML SCHEMA COLLECTION [ <SQL Server schema>.] <collection name>
```

So, to get rid of our ProductDescriptionSchemaCollectionSummaryRequired schema collection we created earlier, we could execute:

```
DROP XML SCHEMA COLLECTION ProductDescriptionSchemaCollectionSummaryRequired
```

And it's gone.

XML Data Type Methods

The XML data type carries several intrinsic methods with it. These methods are unique to the XML data type, and no other current data type has anything that is at all similar. The syntax within these methods varies a bit because they are based on different, but mostly industry-standard, XML access methods. The basic syntax for calling the method is standardized though:

```
<instance of xml data type>. <method>
```

There are a total of five methods available:

- ❑ .query — An implementation of the industry-standard XQuery language. This allows you to access your XML by running XQuery-formatted queries. XQuery allows for the prospect that you may be returning multiple pieces of data rather than a discreet value.
- ❑ .value — This one allows you to access a discreet value within a specific element or attribute.
- ❑ .modify — This is Microsoft’s own extension to XQuery. Whereas XQuery is limited to requesting data (no modification language), the modify method extends XQuery to allow for data modification.
- ❑ .nodes — Used to break up XML data into individual, more relational-style rows.
- ❑ .exist — Much like the IF EXISTS clause we use extensively in standard SQL, the exist() XML data type method tests to see whether a specific kind of data exists. In the case of exist(), the test is to see whether a particular node or attribute has an entry in the instance of XML you’re testing.

.query (SQL Server’s Implementation of XQuery)

.query is an implementation of the industry standard XQuery language. The result works much like a SQL query, except that the results are for matching XML data nodes rather than relational rows and columns.

.query requires a parameter that is a valid XQuery to be run against your instance of XML data. For example, if we wanted the steps out of the product documentation for ProductID 66, we could run the following:

```
SELECT ProductModelID, Instructions.query('declare namespace
PI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
/PI:root/PI:Location/PI:step') AS Steps
FROM Production.ProductModel
WHERE ProductModelID = 66
```

Chapter 16

Note that the URL portion of the namespace declaration must be entered on a single line. They are shown here word wrapped onto multiple lines because there is a limit to how many characters we can show per line in print. Make sure you include the entire URL on a single line.

The result is rather verbose, so I've truncated the right side of it, but you can see that we've trimmed things down such that we're getting only those nodes at the step level or lower in the XML hierarchy.

```
ProductModelID Steps
-----
66      <PI:step xmlns:PI="http://schemas.microsoft.com/sqlser...
          Put the <PI:material>Seat post Lug (Product N...
          </PI:step><PI:step xmlns:PI="http://schemas.micro...
          Insert the <PI:material>Pinch Bolt (Product N...
          </PI:step><PI:step xmlns:PI="http://schemas.micro...
          Attach the <PI:material>LL Seat (Product Numb...
          </PI:step><PI:step xmlns:PI="http://schemas.micro...
          Inspect per specification <PI:specs>FI-620</P...
          </PI:step>
```

(1 row(s) affected)

It's also worth pointing out that all the XML still came in one column in one row per data row in the database.

It bears repeating that .query cannot modify data—it is a read-only operation

Notice, by the way, my need to declare the namespace in this. Since a namespace is declared as part of the referenced schema collection, you can see how it really expands and virtually destroys the readability of our query. We can fix that by using the `WITH XMLNAMESPACES()` declaration:

```
WITH XMLNAMESPACES ('http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions' AS PI)

SELECT ProductModelID, Instructions.query('/PI:root/PI:Location/PI:step') AS Steps
FROM Production.ProductModel
WHERE ProductModelID = 66
```

Note that the URL portion of the namespace declaration must be entered on a single line. They are shown here word wrapped onto multiple lines because there is a limit to how many characters we can show per line in print. Make sure you include the entire URL on a single line.

Gives you a somewhat more readable query, but yields the same result set.

.value

The `.value` method is all about querying out discrete data. It uses an XPath syntax to locate a specific node and extract a scalar value. The syntax looks like this:

```
<instance of xml data type>.value (<XPath location>, <non-xml SQL Server Type>)
```

The trick here is making certain that the XPath specified really will return a discrete value.

If, for example, we wanted to know the value of the `LaborHours` attribute in the first `Location` element for `ProductModelID` 66, we might write something like:

```
WITH XMLNAMESPACES ('http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions' AS PI)

SELECT ProductModelID,
       Instructions.value('(/PI:root/PI:Location/@LaborHours)[1]',
                           'decimal (5,2)') AS Location
  FROM Production.ProductModel
 WHERE ProductModelID = 66
```

Note that the URL portion of the namespace declaration must be entered on a single line. They are shown here word wrapped onto multiple lines because there is a limit to how many characters we can show per line in print. Make sure you include the entire URL on a single line.

Check the results:

ProductModelID	Location
66	1.50

(1 row(s) affected)

Note that SQL Server has extracted just the specified attribute value (in this case, the `LaborHours` attribute of the `Location` node) as a discrete piece of data. The data type of the returned values must be castable into a non-XML type in SQL Server, and must return a scalar value—that is, you cannot have multiple rows.

.modify

Ah, here things get just a little interesting.

XQuery, left in its standard W3C form, is a read-only kind of thing—that is, it is great for selecting out data but offers no equivalents to `INSERT`, `UPDATE`, or `DELETE`. Bummer deal! Well, Microsoft is apparently having none of that and has done its own extension to XQuery to provide data manipulation for XQuery. This extension to XQuery is called XML Data Manipulation Language, or XML DML. XML DML adds three new commands to XQuery:

- insert
- delete
- replace value of

Note that these new commands, like all XML keywords, are case sensitive.

Each of these does what it implies, with `replace value of` taking the place of SQL's `UPDATE` statement.

Chapter 16

If, for example, we wanted to increase the original 1.5 labor hours in our .value example, we might write something like:

```
WITH XMLNAMESPACES ('http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions' AS PI)

UPDATE Production.ProductModel
SET Instructions.modify('replace value of (/PI:root/PI:Location/@LaborHours)[1]
with 1.75')
WHERE ProductModelID = 66
```

Note that the URL portion of the namespace declaration must be entered on a single line. They are shown here word wrapped onto multiple lines because there is a limit to how many characters we can show per line in print. Make sure you include the entire URL on a single line.

Now if we re-run our .value command:

```
WITH XMLNAMESPACES ('http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/
ProductModelManuInstructions' AS PI)

SELECT ProductModelID, Instructions.value('(/PI:root/PI:Location/@LaborHours)[1]',
'decimal (5,2)' ) AS Location
FROM Production.ProductModel
WHERE ProductModelID = 66
```

Note that the URL portion of the namespace declaration must be entered on a single line. They are shown here word wrapped onto multiple lines because there is a limit to how many characters we can show per line in print. Make sure you include the entire URL on a single line.

We get a new value:

ProductModelID	Location
66	1.75

(1 row(s) affected)

Note the way that this is essentially an UPDATE within an UPDATE. We are modifying the SQL Server row, so we must use an UPDATE statement to tell SQL Server that our row of relational data (which just happens to have XML within it) is to be updated. We must also use the replace value of keyword to specify the XML portion of the update.

.nodes

.nodes is used to take blocks of XML and separate what would have, were it stored in a relational form, been multiple rows of data. Taking one XML document and breaking it out into individual parts in this way is referred to as *shredding* the document.

What we are doing with .nodes is essentially breaking the instances of XML data into their own table (with as many rows as there are instances of data meeting that XQuery criteria). As you might expect, this means we need to treat .nodes results as a table rather than a column. The primary difference

between .nodes and a typical table is that we must *cross apply* our .nodes results back to the specific table that we are sourcing our XML data from. So, .nodes really involves more syntax than just ".nodes"—think of it somewhat like a join, but using the special CROSS APPLY keyword in the place of the JOIN and .nodes instead of the ON clause. It looks like this:

```
SELECT <column list>
FROM <source table>
CROSS APPLY <column name>.nodes(<XQuery>) AS <table alias for your .nodes results>
```

This is fairly confusing stuff, so let's look back at our .value example earlier. We see a query that looked for a specific entry and, therefore, got back exactly one result:

```
WITH XMLNAMESPACES ('http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/
ProductModelManuInstructions' AS PI)

SELECT ProductModelID,
       Instructions.value('/PI:root/PI:Location/@LaborHours')[1]',
       'decimal (5,2)' ) AS Location
FROM Production.ProductModel
WHERE ProductModelID = 66
```

Note that the URL portion of the namespace declaration must be entered on a single line. They are shown here word wrapped onto multiple lines because there is a limit to how many characters we can show per line in print. Make sure you include the entire URL on a single line.

.value expects a scalar result, so we needed to make certain our XQuery would return just that single value per individual row of XML. .nodes tells SQL Server to use XQuery to map to a specific location and treat each entry found in that XQuery to be an individual row instead.

Let's modify our .value example to return all LocationID's and their respective labor hours. We want to be able to perform queries against the data in our XML as though it were relational data, so we need to break up our LocationID and LaborHours information into columns just as if they were in a relational table.

```
WITH XMLNAMESPACES ('http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/
ProductModelManuInstructions' AS PI)

SELECT pm.ProductModelID,
       pmi.Location.value('./@LocationID', 'int') AS LocationID,
       pmi.Location.value('./@LaborHours', 'decimal(5,2)') AS LaborHours
FROM Production.ProductModel pm
CROSS APPLY pm.Instructions.nodes('/PI:root/PI:Location') AS pmi(Location)
```

Note that the URL portion of the namespace declaration must be entered on a single line. They are shown here word wrapped onto multiple lines because there is a limit to how many characters we can show per line in print. Make sure you include the entire URL on a single line.

Notice that through the use of our .nodes method, we are essentially turning one table (ProductModel) into two tables (the source table and the .nodes results from the Instructions column within the ProductModel table). Take a look at the results:

Chapter 16

ProductModelID	LocationID	LaborHours
7	10	2.50
7	20	1.75
7	30	1.00
7	45	0.50
7	50	3.00
7	60	4.00
10	10	2.00
10	20	1.50
10	30	1.00
10	4	1.50
10	50	3.00
10	60	4.00
43	50	3.00
44	50	3.00
47	10	1.00
47	20	1.00
47	50	3.50
48	10	1.00
48	20	1.00
48	50	3.50
53	50	0.50
66	50	1.75
67	50	1.00

(23 row(s) affected)

As you can see, we are getting back multiple rows for many of what was originally a single row in the ProductModel table. For example, ProductModelID 7 had six different instances of the Location element, so we received six rows instead of just the single row that existed in the ProductModel table.

While this is, perhaps, the most complex of the various XML data type methods, the power that it gives transform XML data for relational use is virtually limitless.

.exist

.exist works something like the EXISTS statement in SQL. It accepts an expression (in this case, an XQuery expression rather than a SQL expression) and will return a Boolean indication of whether the expression was true or not. (NULL is also a possible outcome.)

If, in our .modify example, we had wanted to show rows that contain steps that had spec elements, we could use .exist:

```
WITH XMLNAMESPACES ('http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/
ProductModelManuInstructions' AS PI)

SELECT ProductModelID, Instructions
FROM Production.ProductModel
WHERE Instructions.exist('/PI:root/PI:Location/PI:step/PI:specs') = 1
```

Pay particular attention to at what point the test condition is being applied!

For example, the code would show us rows where at least one step had a spec element in it—it does not necessarily require that every step has the spec element. If we wanted every element to be tested, we would either need to pull the elements out as individual rows (using .nodes) or place the test condition in the XQuery.

Note that the URL portion of the namespace declaration must be entered on a single line. They are shown here word wrapped onto multiple lines because there is a limit to how many characters we can show per line in print. Make sure you include the entire URL on a single line.

Enforcing Constraints beyond the Schema Collection

We are, of course, used to the concept of constraints by now. We've dealt with them extensively in this book, and, since this is a *Professional* title, you no doubt had seen at least a little bit of them before you ever picked this book up. Well, if our relational database needs constraints, it follows that our XML data does. Indeed, we've already implemented much of that idea through the use of schema collections. But what if we want to enforce requirements that go beyond the base schema?

Surprisingly, you cannot apply XML data type methods within a constraint declaration. How do you get around this problem? Well, wrap the tests up in a user-defined function (UDF), and then utilize that function in your constraint.

I have to admit I'm somewhat surprised that the methods are not usable within the CONSTRAINT declaration, but things like functions are. All I can say is "go figure. . ." I'll just quietly hope they fix this in a future release, as it seems a significant oversight on something that shouldn't have been all that difficult (yeah, I know—easy for me to say since they have to write that code, not me!).

Retrieving Relational Data in XML Format

This is an area that SQL Server already had largely figured out prior to the 2005 release. We had couple of different options, and we had still more options within those options—between them all, things have been pretty flexible for quite some time. Let's take a look.

The FOR XML Clause

This clause is at the root of most of the different integration models available. With the exception of XML mapping schemas (fairly advanced, but we'll touch on them briefly later in the chapter) and the use of XPath, FOR XML will serve as the way of telling SQL Server that it's XML that you want back, not the more typical result set. It is essentially just an option added onto the end of the existing T-SQL SELECT statement.

Let's look back at the SELECT statement syntax from Chapter 3:

```
SELECT <column list>
[FROM <source table(s)>]
[WHERE <restrictive condition>]
[GROUP BY <column name or expression using a column in the SELECT list>]
[HAVING <restrictive condition based on the GROUP BY results>]
[ORDER BY <column list>]
[FOR XML {RAW|AUTO|EXPLICIT|PATH}
    [, XMLDATA][, ELEMENTS][, BINARY base64]]
[OPTION (<query hint>, [, ...n])]
```

Chapter 16

Most of this should seem pretty trivial by now — after all, we've been using this syntax throughout a lot of hard chapters by this time — but it's time to focus in on that `FOR XML` line.

`FOR XML` provides four different initial options for how you want your XML formatted in the results:

- ❑ `RAW`—This sends each row of data in your result set back as a single data element, with the element name of "row" and with each column listed as an attribute of the "row" element. Even if you join multiple tables, `RAW` outputs the results with the same number of elements as you would have rows in a standard SQL query.
- ❑ `AUTO`—This option labels each element with either the table name or table name alias that the data is sourced from. If there is data output from more than one table in the query, the data from each table is split into separate, nested elements. If `AUTO` is used, then an additional option, `ELEMENTS`, is also supported if you would like column data presented as elements rather than as attributes.
- ❑ `EXPLICIT`—This one is certainly the most complex to format your query with, but the end result is that you have a high degree of control of what the XML looks like finally. With this option, you define something of a hierarchy to the data that's being returned, and then format your query such that each piece of data belongs to a specific hierarchy level (and gets assigned a tag accordingly) as desired. This choice has largely been supplanted by the `PATH` option and is here for backward compatibility.
- ❑ `PATH`—This was added in SQL Server 2005 to try to provide the level of flexibility of `EXPLICIT` in a more usable format — this is generally going to be what you want to use when you need a high degree of control of the format of the output.

Note that none of these options provide the required root element. If you want the XML document to be considered to be "well formed," then you will need to wrap the results with a proper opening and closing tag for your root element. While this is in some ways a hassle, it is also a benefit — it means that you can build more complex XML by stringing multiple XML queries together and wrapping the different results into one XML file.

In addition to the major formatting options, there are other optional parameters that further modify the output that SQL Server provides in an XML Query:

- ❑ `XMLELEMENT`—This tells SQL Server that you would like to apply an XML schema onto the front of the results. The schema will define the structure (including data types) and rules of the XML data that follows.
- ❑ `ELEMENTS`—This option is available only when you are using the `AUTO` formatting option. It tells SQL Server that you want the columns in your data returned as nested elements rather than as attributes.
- ❑ `BINARY BASE64`—This tells SQL Server to encode any binary columns (binary, varbinary, image) in base64 format. This option is implied (SQL Server will use it even if you don't state it) if you are also using the `AUTO` option. It is not implied but is currently the only effective option for `EXPLICIT` and `RAW` queries — eventually, the plan is to have these two options automatically provide a URL link to the binary data (unless you say to do the base64 encoding), but this is not yet implemented.

- TYPE—Tells SQL Server to return the results reporting the XML data type instead of the default Unicode character type.
- ROOT—This option will have SQL Server add the root node for you so you don't have to. You can either supply a name for your root or use the default (root).

Let's explore all these options in a little more detail.

RAW

This is something of the “no fuss, no muss” option. The idea here is to just get it done—no fanfare, no special formatting at all—just the absolute minimum to translate a row of relational data into an element of XML data. The element is named “row” (creative, huh?), and each column in the select list is added as an attribute using whatever name the column would have appeared with if you had been running a more traditional SELECT statement.

One downside to the way in which attributes are named is that you need to make certain that every column has a name. Normally, SQL Server will just show no column heading if you perform an aggregation or other calculated column and don't provide an alias—when doing XML queries, everything MUST have a name, so don't forget to alias calculated columns.

So, let's start things out with something relatively simple. Imagine that our manager has asked us to provide a query that lists a few customers' orders—say CustomerIDs 1 and 2. After cruising through just the first five or so chapters of the book, you would probably say “No Problem!” and supply something like:

```
SELECT c.ContactID,
       c.LastName,
       c.FirstName,
       soh.SalesOrderID,
       soh.OrderDate
  FROM Person.Contact c
 JOIN Sales.SalesOrderHeader soh
    ON c.ContactID = soh.ContactID
 WHERE c.ContactID = 1 OR c.ContactID = 2
```

So, you go hand your boss the results:

1	Achong	Gustavo	44132	2001-09-01
00:00:00.000				
1	Achong	Gustavo	45579	2002-03-01
00:00:00.000				
...				
...				
2	Abel	Catherine	65157	2004-03-01
00:00:00.000				
2	Abel	Catherine	71782	2004-06-01
00:00:00.000				

Chapter 16

Easy, right? Well, now the boss comes back and says, “Great — now I’ll just have Billy Bob write something to turn this into XML — too bad that will probably take a day or two.” This is your cue to step in and say, “Oh, why didn’t you say so?” and simply add three key words:

```
SELECT c.ContactID,
       c.LastName,
       c.FirstName,
       soh.SalesOrderID,
       soh.OrderDate
  FROM Person.Contact c
 JOIN Sales.SalesOrderHeader soh
    ON c.ContactID = soh.ContactID
 WHERE c.ContactID = 1 OR c.ContactID = 2
 FOR XML RAW
```

You have just made the boss very happy. The output is a one-to-one match versus what we would have seen in the result set had we ran just a standard SQL query:

```
<row ContactID="1" LastName="Achong" FirstName="Gustavo" SalesOrderID="44132"
      OrderDate="2001-09-01T00:00:00" />
<row ContactID="1" LastName="Achong" FirstName="Gustavo" SalesOrderID="45579"
      OrderDate="2002-03-01T00:00:00" />
<row ContactID="1" LastName="Achong" FirstName="Gustavo" SalesOrderID="46389"
      OrderDate="2002-06-01T00:00:00" />
<row ContactID="1" LastName="Achong" FirstName="Gustavo" SalesOrderID="47454"
      OrderDate="2002-09-01T00:00:00" />
<row ContactID="1" LastName="Achong" FirstName="Gustavo" SalesOrderID="48395"
      OrderDate="2002-12-01T00:00:00" />
<row ContactID="1" LastName="Achong" FirstName="Gustavo" SalesOrderID="49495"
      OrderDate="2003-03-01T00:00:00" />
<row ContactID="1" LastName="Achong" FirstName="Gustavo" SalesOrderID="50756"
      OrderDate="2003-06-01T00:00:00" />
<row ContactID="2" LastName="Abel" FirstName="Catherine" SalesOrderID="53459"
      OrderDate="2003-09-01T00:00:00" />
<row ContactID="2" LastName="Abel" FirstName="Catherine" SalesOrderID="58907"
      OrderDate="2003-12-01T00:00:00" />
<row ContactID="2" LastName="Abel" FirstName="Catherine" SalesOrderID="65157"
      OrderDate="2004-03-01T00:00:00" />
<row ContactID="2" LastName="Abel" FirstName="Catherine" SalesOrderID="71782"
      OrderDate="2004-06-01T00:00:00" />
```

Let me just issue a reminder that Management Studio will truncate any column where the length exceeds the number set in the Tools→Options menu in the Query Results “Results to Text” node (maximum is 8192). This issue exists in the results window (grid or text, though grid will allow larger numbers if the data is XML) and if you output directly to a file. This is an issue with the tool — not SQL Server itself. If you use another method to retrieve results (ADO.NET for example), you shouldn’t encounter an issue with this.

Also, be aware that I’ve added carriage returns in the above results for clarity’s sake — SQL Server just runs all the elements together to make them more compact.

We have one element in XML for each row of data our query produced. All column information, regardless of what table was the source of the data, is represented as an attribute of the “row” element. The downside

of this is that we haven't represented the true hierarchical nature of your data—orders are placed only by customers. The upside, however, is that the XML DOM—if that's the model you're using—is going to be much less deep and, hence, will have a slightly smaller footprint in memory and perform better, depending on what you're doing.

AUTO

AUTO takes a somewhat different approach to our data than RAW does. AUTO tries to format things a little better for you—naming elements based on the table (or the table alias if you use one). In addition, AUTO recognizes the notion that our data probably has some underlying hierarchical notion to it that is supposed to be represented in the XML.

Let's go back to our customer orders example from the last section. This time, we'll make use of the AUTO option, so we can see the difference versus the rather plain output we got with RAW.

```
SELECT c.ContactID,
       c.LastName,
       c.FirstName,
       soh.SalesOrderID,
       soh.OrderDate
  FROM Person.Contact c
 JOIN Sales.SalesOrderHeader soh
    ON c.ContactID = soh.ContactID
 WHERE c.ContactID = 1 OR c.ContactID = 2
 FOR XML AUTO
```

The first apparent difference is that the element name has changed to be that of the name or alias of the table that is the source of the data—you'll want to consider this when choosing the aliases for your tables in a FOR XML AUTO query. Perhaps an even more significant difference appears when we look at the XML more thoroughly (I have again cleaned up the output a bit for clarity):

```
<c ContactID="1" LastName="Achong" FirstName="Gustavo">
  <soh SalesOrderID="44132" OrderDate="2001-09-01T00:00:00"/>
  <soh SalesOrderID="45579" OrderDate="2002-03-01T00:00:00"/>
  <soh SalesOrderID="46389" OrderDate="2002-06-01T00:00:00"/>
  <soh SalesOrderID="47454" OrderDate="2002-09-01T00:00:00"/>
  <soh SalesOrderID="48395" OrderDate="2002-12-01T00:00:00"/>
  <soh SalesOrderID="49495" OrderDate="2003-03-01T00:00:00"/>
  <soh SalesOrderID="50756" OrderDate="2003-06-01T00:00:00"/>
</c>
<c ContactID="2" LastName="Abel" FirstName="Catherine">
  <soh SalesOrderID="53459" OrderDate="2003-09-01T00:00:00"/>
  <soh SalesOrderID="58907" OrderDate="2003-12-01T00:00:00"/>
  <soh SalesOrderID="65157" OrderDate="2004-03-01T00:00:00"/>
  <soh SalesOrderID="71782" OrderDate="2004-06-01T00:00:00"/>
</c>
```

Data that is sourced from our second table (as determined by the SELECT list) is nested inside the data sourced from the first table. In this case, our soh elements are nested inside our c elements. If a column from the SalesOrderHeader table were listed first in our select list, then Contact would be nested inside SalesOrderHeader.

Chapter 16

Pay attention to this business of the ordering of your SELECT list! Think about the primary question your XML query is meant to answer. Arrange your SELECT list such that the style that it produces is fitting for the goal of your XML. Sure, you could always style it into the different form—but why do that if SQL Server could have just produced it for you that way in the first place?

The downside to using AUTO is that the resulting XML data model ends up being slightly more complex. Also, AUTO is currently not compatible with a GROUP BY clause. The upside is that the data is more explicitly broken up into a hierarchical model. This makes life easier for situations where the elements are more significant breaking points—such as where you have a doubly sorted report (for example, SalesOrderHeader rows sorted within Contact rows).

EXPLICIT

The word *explicit* is an interesting choice for this option—it loosely describes the kind of language you’re likely to use while trying to create your query. The EXPLICIT option takes much more effort to prepare, but it also rewards that effort with very fine granularity of control over what’s an element and what’s an attribute, as well as what elements are nested in what other elements.

EXPLICIT enables you to define each level of the hierarchy and how each level is going to look. In order to define the hierarchy, you create what is internally called the universal table. The universal table is, in many respects, just like any other result set you might produce in SQL Server. It is usually produced by making use of UNION statements to piece it together one level at a time, but you could, for example, build much of the data in a UDF and then make a SELECT against that to produce the final XML. The big difference between the universal table and a more traditional result set is that you must provide sufficient metadata right within your result set such that SQL Server can then transform that result set into an XML document in the schema you desire.

What do I mean by “sufficient metadata”? Well, to give you an idea of just how complex this can be, let’s look at a real universal table—one used by a code example we’ll examine a little later in the section:

Tag	Parent	c!1!ContactID	c!1!LastName	c!1!FirstName	soh!2!SalesOrderID	soh!2!OrderDate
1	NULL	1	Achong	Gustavo	NULL	NULL
2	1	1	Achong	Gustavo	44132	00:00:0
2	1	1	Achong	Gustavo	45579	00:00:0
2	1	1	Achong	Gustavo	46389	00:00:0
2	1	1	Achong	Gustavo	47454	00:00:0
2	1	1	Achong	Gustavo	48395	00:00:0
2	1	1	Achong	Gustavo	49495	00:00:0
2	1	1	Achong	Gustavo	50756	00:00:0
1	NULL	2	Abel	Catherine	NULL	NULL
2	1	2	Abel	Catherine	53459	00:00:0
2	1	2	Abel	Catherine	58907	00:00:0
2	1	2	Abel	Catherine	65157	00:00:0
2	1	2	Abel	Catherine	71782	00:00:0

This is what the universal table we would need to build would look like in order to make our EXPLICIT return exactly the same results that we received with our AUTO query in the last example.

You're first inclination might be to say, "Hey, if this is just producing the same thing as AUTO, why use it?" Well, this particular example happens to be producible using AUTO—I'm using this one on purpose to illustrate some functional differences compared to something you've already seen. We will, however, see later in this section that EXPLICIT will allow us to do the "extras" in formatting that aren't possible with AUTO or RAW (but are with PATH)—so please bear with me on this one.

You should note several things about this result set:

- It has two special metadata columns—*Tag* and *Parent*—added to it that do not, otherwise, relate to the data (they didn't come from table columns).
- The actual column names are adhering to a special format (which happens to supply additional metadata).
- The data has been ordered based on the hierarchy.

Each of these items is critical to our end result, so, before we start working a complete example, let's look at what we need to know to build it.

Tag and Parent

XML is naturally hierarchical in nature (elements are contained with other elements, which essentially creates a parent-child relationship). *Tag* and *Parent* are columns that define the relationship of each row to the element hierarchy. Each row is assigned to a certain tag level (which will later have an element name assigned to it)—that level, as you might expect, goes in the *Tag* column. *Parent* then supplies reference information that indicates what the next highest level in the hierarchy is—by doing this, SQL Server knows at what level this row needs to be nested or assigned as an attribute (what it's going to be—element or attribute—will be figured out based on the column name—but we'll get to that in our next section). If *Parent* is NULL, then SQL Server knows that this row must be a top-level element or an attribute of that element.

So, if we had data that looked like this:

Tag	Parent
1	NULL
2	1

then the first row would be related to a top-level element (an attribute of the outer element or the element itself), and the second would be related to an element that was nested inside the top-level element (its *Parent* value of 1 matches with the *Tag* value of the first).

Column Naming

Frankly, this was the most confusing part of all when I first started looking at EXPLICIT. While *Tag* and *Parent* have nice neat demarcation points (they are each their own column), the name takes several pieces of metadata and crams them together as one thing—the only way to tell where one stops and the next begins is by separating them by an exclamation mark (!).

Chapter 16

The naming format looks like this:

```
<element name>!<tag>!<attribute name>[!{element|hide|ID|IDREF|IDREFS|xml|  
xmltext|cdata}]
```

The element name is, of course, just that—what you want to be the name of the element in the XML. For any given tag level, once you define a column with one name, any other column with that same tag must have the same name as the previous column(s) with that tag number. So, if you have a column already defined as [MyElement!2!MyCol], then another column could be named [MyElement!2!MyOtherCol], but [SomeOtherName!2!MyOtherCol] could not be.

The tag relates the column to rows with a matching tag number. When SQL Server looks at the universal table, it reads the tag number and then analyzes the columns with the same tag number. So, when SQL Server sees the row:

Tag	Parent	c!1!ContactID	c!1!LastName	c!1!FirstName	soh!2!SalesOrderID	soh!2!OrderDate
1	NULL	1	Achong	Gustavo	NULL	NULL

it can look at the tag number, see that it is 1, and know that it should process c!1!ContactID, c!1!LastName, and c!1!FirstName, but that it doesn't have to process soh!2!SalesOrderID, for example.

That takes us to the attribute name, which begins the next phase of getting more complex (hey, we still have one more to go after this!). If you do not specify a directive (which comes next), then the attribute is required and is the name of the XML attribute that this column will supply a value for. The attribute will be in the XML as part of the element specified in the column name.

If you do specify a directive, then the attribute falls into three different camps:

- It's prohibited**—That is, you must leave the attribute blank (you do still use a bang (!) to mark its place though). This is the case if you use a CDATA directive.
- It's optional**—That is, you can supply the attribute but don't have to. What happens in this case varies depending on the directive that you've chosen.
- It's still required**—This is true for the elements and xml directives. In this case, the name of the attribute will become the name of a totally new element that will be created as a result of the elements or xml directive.

So, now that we have enough of the naming down to meet the minimum requirements for a query, let's go ahead and look at an example of what kind of query produces what kind of results.

We will start with the query to produce the same basic data that we used in our RAW and AUTO examples. You will notice that EXPLICIT has a much bigger impact on the code than we saw when we went with RAW and AUTO. With both RAW and AUTO, we added the FOR XML clause at the end, and we were largely done. With EXPLICIT, we will quickly see that we need to entirely rethink the way our query comes together.

It looks like this (yuck):

```
USE AdventureWorks

SELECT 1
      , NULL
      , c.ContactID
      , c.LastName
      , c.FirstName
      , NULL
      , NULL
      , NULL
      , NULL
      , NULL
      , as Tag,
      , as Parent,
      , as [c!1!ContactID],
      , as [c!1!LastName],
      , as [c!1!FirstName],
      , as [soh!2!SalesOrderID],
      , as [soh!2!OrderDate]
FROM Person.Contact c
WHERE c.ContactID = 1 OR c.ContactID = 2

UNION ALL

SELECT 2,
      , 1,
      , c.ContactID,
      , c.LastName,
      , c.FirstName,
      , soh.SalesOrderID,
      , soh.OrderDate
FROM Person.Contact c
JOIN Sales.SalesOrderHeader soh
  ON c.ContactID = soh.ContactID
WHERE c.ContactID = 1 OR c.ContactID = 2
ORDER BY [c!1!ContactID], [soh!2!SalesOrderID]
FOR XML EXPLICIT
```

Notice that we use the FOR XML clause only once—after the last query in the UNION.

I reiterate — yuck! But, ugly as it is, with just a few changes, I could change my XML into forms that AUTO wouldn't give me.

As a fairly simple illustration, let's make a couple of small alterations to our requirements for this query. What if we decided that we wanted the `LastName` information to be an attribute of the `soh` rather than (or, as it happens, in addition to) the `c` element? With `AUTO`, we would need some trickery in order to get this (for every row, we would need to look up the `Contact` again using a correlated subquery — `AUTO` won't let you use the same value in two places). If you had multiple lookups, your code could get very complex — indeed, you might not be able to get what you're after at all. With `EXPLICIT`, this is all relatively easy (at least, by `EXPLICIT`'s definition of easy).

To do this with **EXPLICIT**, we just need to reference the `LastName` in our `SELECT` list again, but associate the new instance of it with `sob` instead of `c`:

```
USE AdventureWorks

SELECT 1           as Tag,
       NULL        as Parent,
       c.ContactID as [!!ContactID]
```

Chapter 16

```
c.LastName      as [c!1!LastName],  
c.FirstName    as [c!1!FirstName],  
NULL          as [soh!2!SalesOrderID],  
NULL          as [soh!2!OrderDate],  
NULL          as [soh!2!LastName]  
  
FROM Person.Contact c  
WHERE c.ContactID = 1 OR c.ContactID = 2  
  
UNION ALL  
  
SELECT 2,  
       1,  
       c.ContactID,  
       c.LastName,  
       c.FirstName,  
       soh.SalesOrderID,  
       soh.OrderDate,  
       soh.OrderDate,  
       c.LastName  
  
FROM Person.Contact c  
JOIN Sales.SalesOrderHeader soh  
  ON c.ContactID = soh.ContactID  
WHERE c.ContactID = 1 OR c.ContactID = 2  
ORDER BY [c!1!ContactID], [soh!2!SalesOrderID]  
FOR XML EXPLICIT
```

Execute this, and you get pretty much the same results as before, only this time you received the additional attribute you were looking for in your `soh` element.

```
<c ContactID="1" LastName="Achong" FirstName="Gustavo">  
  <soh SalesOrderID="44132" OrderDate="2001-09-01T00:00:00" LastName="Achong" />  
  <soh SalesOrderID="45579" OrderDate="2002-03-01T00:00:00" LastName="Achong" />  
  <soh SalesOrderID="46389" OrderDate="2002-06-01T00:00:00" LastName="Achong" />  
  <soh SalesOrderID="47454" OrderDate="2002-09-01T00:00:00" LastName="Achong" />  
  <soh SalesOrderID="48395" OrderDate="2002-12-01T00:00:00" LastName="Achong" />  
  <soh SalesOrderID="49495" OrderDate="2003-03-01T00:00:00" LastName="Achong" />  
  <soh SalesOrderID="50756" OrderDate="2003-06-01T00:00:00" LastName="Achong" />  
</c>  
<c ContactID="2" LastName="Abel" FirstName="Catherine">  
  <soh SalesOrderID="53459" OrderDate="2003-09-01T00:00:00" LastName="Abel" />  
  <soh SalesOrderID="58907" OrderDate="2003-12-01T00:00:00" LastName="Abel" />  
  <soh SalesOrderID="65157" OrderDate="2004-03-01T00:00:00" LastName="Abel" />  
  <soh SalesOrderID="71782" OrderDate="2004-06-01T00:00:00" LastName="Abel" />  
</c>
```

This example is really just for starters. You can utilize directives to achieve far more flexibility—shaping and controlling both your data and your schema output (if you use the `XMldata` option).

Directives are a real pain to understand. Once you do understand them, they aren't all that bad to deal with, though they can still be confusing at times (some of them work pretty counterintuitively and behave differently in different situations). My personal opinion (and the members of the dev team I know are going to shoot me for saying this) is that someone at Microsoft had a really bad day and decided to make something that would inflict as much pain as he/she was feeling but would be so cool that people wouldn't be able to help themselves but use it.

All together, there are eight possible directives you can use. Some can be used in the same level of the hierarchy — others are mutually exclusive within a given hierarchy level.

The purpose behind directives is to allow you to tweak your results. Without directives, the `EXPLICIT` option would have little or no value (`AUTO` would take care of most “real” things that you can do with `EXPLICIT` if you don’t use directives, even though, as I indicated earlier, you sometimes have to get a little tricky). So, with this in mind, let’s look at what directives are available.

element

This is probably the easiest of all the directives to understand. All it does is indicate that you want the column in question to be added as an element rather than an attribute. The element will be added as a child to the current tag. For example, let’s say that our manager from the previous examples has indicated that he or she needs the `OrderDate` to be represented as its own element. This can be accomplished as easily as adding the `element` directive to the end of our `OrderDate` field:

```
USE AdventureWorks

SELECT 1      as Tag,
       NULL     as Parent,
       c.ContactID as [c!1!ContactID],
       c.LastName  as [c!1!LastName],
       c.FirstName as [c!1!FirstName],
       NULL       as [soh!2!SalesOrderID],
       NULL       as [soh!2!OrderDate!element]
FROM Person.Contact c
WHERE c.ContactID = 1 OR c.ContactID = 2

UNION ALL

SELECT 2,
       1,
       c.ContactID,
       c.LastName,
       c.FirstName,
       soh.SalesOrderID,
       soh.OrderDate
FROM Person.Contact c
JOIN Sales.SalesOrderHeader soh
  ON c.ContactID = soh.ContactID
WHERE c.ContactID = 1 OR c.ContactID = 2
ORDER BY [c!1!ContactID], [soh!2!SalesOrderID]
FOR XML EXPLICIT
```

Suddenly, we have an extra element instead of an attribute:

```
<c ContactID="1" LastName="Achong" FirstName="Gustavo">
  <soh SalesOrderID="44132">
    <OrderDate>2001-09-01T00:00:00</OrderDate>
  </soh>
  <soh SalesOrderID="45579">
    <OrderDate>2002-03-01T00:00:00</OrderDate>
  </soh><soh SalesOrderID="46389">
    <OrderDate>2002-06-01T00:00:00</OrderDate>
  </soh>
```

Chapter 16

```
<soh SalesOrderID="47454">
    <OrderDate>2002-09-01T00:00:00</OrderDate>
</soh>
<soh SalesOrderID="48395">
    <OrderDate>2002-12-01T00:00:00</OrderDate>
</soh>
<soh SalesOrderID="49495">
    <OrderDate>2003-03-01T00:00:00</OrderDate>
</soh>
<soh SalesOrderID="50756">
    <OrderDate>2003-06-01T00:00:00</OrderDate>
</soh>
</c>
<c ContactID="2" LastName="Abel" FirstName="Catherine">
    <soh SalesOrderID="53459">
        <OrderDate>2003-09-01T00:00:00</OrderDate>
    </soh>
    <soh SalesOrderID="58907">
        <OrderDate>2003-12-01T00:00:00</OrderDate>
    </soh>
    <soh SalesOrderID="65157">
        <OrderDate>2004-03-01T00:00:00</OrderDate>
    </soh>
    <soh SalesOrderID="71782">
        <OrderDate>2004-06-01T00:00:00</OrderDate>
    </soh>
</c>
```

xml

This directive is essentially just like the `element` directive. It causes the column in question to be generated as an element rather than an attribute. The differences between the `xml` and `element` directives will be seen only if you have special characters that require encoding—for example, the “=” sign is reserved in XML. If you need to represent an =, then you need to “encode” it (for =, it would be encoded as `&eq;`). With the `element` directive, the content of the element is automatically encoded. With `xml`, the content is passed straight into the resulting XML without encoding. If you use the `xml` directive, no other item at this level (the number) can have a directive other than `hide`.

hide

`Hide` is another simple one that does exactly what it says it does—hides the results of that column.

Why in the world would you want to do that? Well, sometimes we include columns for reasons other than output. For example, in a normal query, we can perform an `ORDER BY` based on columns that do not appear in the `SELECT` list. For `UNION` queries, however, we can’t do that—we have to specify a column in the `SELECT` list because it’s the one thing that unites all the queries that we are performing the `UNION` on.

Let’s use a little example of tracking some product sales. We’ll say that we want a list of all of our products as well as the `SalesOrderIDs` of the orders they shipped on and the date that they shipped. We only want the `ProductID`, but we want the `ProductID` to be sorted such that any given product is near similar products—that means we need to sort based on the `ProductSubcategoryID`, but we do not want the `ProductSubcategoryID` to be included in the end results.

We can start out by building the query without the directive—that way we can see that our sort is working.

```

SELECT 1
      ,NULL          as Tag,
      ,NULL          as Parent,
      ,p.ProductID   as [Product!1!ProductID],
      ,p.ProductSubcategoryID as [Product!1!ProductSubcategoryID],
      ,NULL          as [Order!2!OrderID],
      ,NULL          as [Order!2!OrderDate]
FROM Production.Product p
JOIN Sales.SalesOrderDetail AS sod
    ON p.ProductID = sod.ProductID
JOIN Sales.SalesOrderHeader AS soh
    ON sod.SalesOrderID = soh.SalesOrderID
WHERE soh.OrderDate BETWEEN '2003-03-27' AND '2003-03-27'

UNION ALL

SELECT 2,
      ,1,
      ,p.ProductID,
      ,p.ProductSubcategoryID,
      ,soh.SalesOrderID,
      ,soh.OrderDate
FROM Production.Product AS p
JOIN Sales.SalesOrderDetail AS sod
    ON p.ProductID = sod.ProductID
JOIN Sales.SalesOrderHeader AS soh
    ON sod.SalesOrderID = soh.SalesOrderID
WHERE soh.OrderDate BETWEEN '2003-03-27' AND '2003-03-27'
ORDER BY [Product!1!ProductSubcategoryID], [Product!1!ProductID], [Order!2!OrderID]
FOR XML EXPLICIT

```

Be sure to check out the way we dealt with the OrderDate on this one. Even though I needed to fetch that information out of the SalesOrderHeader table, it was easy (since we're using EXPLICIT anyway) to combine that information with the SalesOrderID from the SalesOrderDetail table. As it happens, I could have also just grabbed the SalesOrderID from the SalesOrderHeader table, too, but sometimes you need to mix data from multiple tables in one element, and this query is yet another demonstration of how we can do just that.

We can see from the results that we are indeed getting the sort we expected:

```

<Product ProductID="779" ProductSubcategoryID="1">
    <Order OrderID="49775" OrderDate="2003-03-27T00:00:00"/>
</Product>
<Product ProductID="782" ProductSubcategoryID="1">
    <Order OrderID="49774" OrderDate="2003-03-27T00:00:00"/>
</Product>
<Product ProductID="764" ProductSubcategoryID="2">
    <Order OrderID="49776" OrderDate="2003-03-27T00:00:00"/>
</Product><
Product ProductID="766" ProductSubcategoryID="2">
    <Order OrderID="49777" OrderDate="2003-03-27T00:00:00"/>
</Product>

```

Chapter 16

Now we'll add our `hide` directive and get rid of the category information:

```
SELECT 1          as Tag,
       NULL        as Parent,
       p.ProductID as [Product!1!ProductID],
       p.ProductSubcategoryID as [Product!1!ProductSubcategoryID!hide],
       NULL        as [Order!2!OrderID],
       NULL        as [Order!2!OrderDate]
  FROM Production.Product p
 JOIN Sales.SalesOrderDetail AS sod
   ON p.ProductID = sod.ProductID
 JOIN Sales.SalesOrderHeader AS soh
   ON sod.SalesOrderID = soh.SalesOrderID
 WHERE soh.OrderDate BETWEEN '2003-03-27' AND '2003-03-27'

UNION ALL

SELECT 2,
       1,
       p.ProductID,
       p.ProductSubcategoryID,
       soh.SalesOrderID,
       soh.OrderDate
  FROM Production.Product AS p
 JOIN Sales.SalesOrderDetail AS sod
   ON p.ProductID = sod.ProductID
 JOIN Sales.SalesOrderHeader AS soh
   ON sod.SalesOrderID = soh.SalesOrderID
 WHERE soh.OrderDate BETWEEN '2003-03-27' AND '2003-03-27'
 ORDER BY [Product!1!ProductSubcategoryID!hide], [Product!1!ProductID],
          [Order!2!OrderID]
FOR XML EXPLICIT
```

And we get the same results; only this time, our `Category` information is indeed hidden:

```
<Product ProductID="779">
  <Order OrderID="49775" OrderDate="2003-03-27T00:00:00"/>
</Product>
<Product ProductID="782">
  <Order OrderID="49774" OrderDate="2003-03-27T00:00:00"/>
</Product>
<Product ProductID="764">
  <Order OrderID="49776" OrderDate="2003-03-27T00:00:00"/>
</Product>
<Product ProductID="766">
  <Order OrderID="49777" OrderDate="2003-03-27T00:00:00"/>
</Product>
```

id, idref, and idrefs

None of these three has any affect whatsoever unless you also make use of the `XMLDATA` option (it goes after the `EXPLICIT` in the `FOR` clause) or validate against some other schema that has the appropriate declarations. This makes perfect sense when you think about what they do—they add things to the schema to enforce behavior, but, without a schema, what do you modify?

You see, XML has the concept of an `id`. An `id` in XML works much the same as a primary key does in relational data—it designates a unique identifier for that element name in your XML document. For any element name, there can be no more than one attribute specified in the `id`. What attribute is to serve as the `id` is defined in the schema for the XML. Once you have one element with a given value for your `id` attribute, no other element with the same element name is allowed to have the same attribute.

Unlike primary keys in SQL, you cannot have multiple attributes make up your `id` in XML (there is no concept of a composite key).

Since XML has a concept that is similar to a primary key, it probably comes as no surprise that XML also has a concept that is similar to a foreign key—that's where `idref` and `idrefs` come in. Both are used to create a reference from an attribute in one element to an `id` attribute in another element.

What does this do for us? Well, if we didn't have these, there would only be one way to create a relationship between two elements—nest them. By giving a certain element an `id` and then making reference to it from an attribute declared as being an `idref` or `idrefs` attribute, we gain the ability to link the two elements, regardless of their position in the document.

This should bring on the question, “OK—so why are there two of them?” The answer is implied in their names: `idref` provides for a single value that must match an existing element’s `id` value. `idrefs` provides a multivalued, whitespace separated list—again, the values must *each* match an existing element’s `id` value. The result is that you use `idref` if you are trying to establish a one-to-many relationship (there will only be one of each `id` value but potentially many elements with that value in an attribute of `idref`). Use `idrefs` when you are trying to establish a many-to-many relationship (each element with an `idrefs` can refer to many `ids`, and those values can be referred to by many `ids`).

To illustrate this one, we'll go with a slight modification of our last query. We'll start with the `idref` directive:

```
SELECT 1 as Tag,
       NULL as Parent,
       p.ProductID as [Product!1!ProductID!ID],
       p.ProductSubcategoryID as [Product!1!ProductSubCategoryID!hide],
       NULL as [Order!2!OrderID],
       NULL as [Order!2!ProductID!idref],
       NULL as [Order!2!OrderDate]
FROM Production.Product AS p
JOIN Sales.SalesOrderDetail AS sod
    ON p.ProductID = sod.ProductID
JOIN Sales.SalesOrderHeader AS soh
    ON sod.SalesOrderID = soh.SalesOrderID
WHERE soh.OrderDate BETWEEN '2003-03-27' AND '2003-03-27'

UNION ALL

SELECT 2,
       1,
       p.ProductID,
       p.ProductSubcategoryID,
       sod.SalesOrderID,
```

Chapter 16

```
sod.ProductID,  
soh.OrderDate  
FROM Production.Product AS p  
JOIN Sales.SalesOrderDetail AS sod  
    ON p.ProductID = sod.ProductID  
JOIN Sales.SalesOrderHeader AS soh  
    ON sod.SalesOrderID = soh.SalesOrderID  
WHERE soh.OrderDate BETWEEN '2003-03-27' AND '2003-03-27'  
ORDER BY [Product!1!ProductSubCategoryID!hide],[Product!1!ProductID!ID],  
[Order!2!OrderID]  
FOR XML EXPLICIT, XMLDATA
```

When we look at the results, there are really just two pieces that we are interested in—the schema and our product element:

```
<Schema name="Schema1" xmlns="urn:schemas-microsoft-com:xml-data"  
        xmlns:dt="urn:schemas-microsoft-com:datatypes">  
    <ElementType name="Product" content="mixed" model="open">  
        <AttributeType name="ProductID" dt:type="id"/>  
        <attribute type="ProductID"/>  
    </ElementType>  
    <ElementType name="Order" content="mixed" model="open">  
        <AttributeType name="OrderID" dt:type="i4"/>  
        <AttributeType name="ProductID" dt:type="idref"/>  
        <AttributeType name="OrderDate" dt:type="dateTime"/>  
        <attribute type="OrderID"/>  
        <attribute type="ProductID"/>  
        <attribute type="OrderDate"/>  
    </ElementType>  
</Schema>
```

In the schema, you can see some fairly specific type information. Our `Product` is declared as a type of element, and you can also see that `ProductID` has been declared as being the `id` for this element type. Likewise, we have an `Order` element with the `ProductID` declared as an `idref`.

The next piece that we're interested in is a `Product` element:

```
<Product xmlns="x-schema:#Schema1" ProductID="779">  
    <Order OrderID="49775" ProductID="779" OrderDate="2003-03-27T00:00:00"/>  
</Product>
```

In this case, notice that SQL Server has referenced our inline schema in the `Product` element. This declares that the `Product` element and everything within it must comply with our schema—thus ensuring that our `id` and `idrefs` will be enforced.

When we try to use the `idrefs` directive, we have to get a little trickier. SQL Server requires that the query that we use to build our `idrefs` list be separate from the query that builds the elements with the `ids`. This means we must add another query to our `UNION` to supply the `idrefs` (the list of possible `ids` has to be known before we can build the `idrefs` list—but the actual `ids` will come after the `id` list). The query to generate the `idrefs` must immediately precede the query that generates the `ids`. This makes the query look pretty convoluted:

```

SELECT 1,
       NULL as Tag,
       NULL as Parent,
       p.ProductID as [Product!1!ProductID],
       NULL as [Product!1!OrderList!idrefs],
       NULL as [Order!2!OrderID!id],
       NULL as [Order!2!OrderDate]
FROM Production.Product p
JOIN Sales.SalesOrderDetail AS sod
  ON p.ProductID = sod.ProductID
JOIN Sales.SalesOrderHeader AS soh
  ON sod.SalesOrderID = soh.SalesOrderID
WHERE soh.OrderDate BETWEEN '2003-03-27' AND '2003-03-31'

UNION ALL

SELECT 1,
       NULL,
       p.ProductID,
       soh.SalesOrderID,
       NULL,
       NULL
FROM Production.Product AS p
JOIN Sales.SalesOrderDetail AS sod
  ON p.ProductID = sod.ProductID
JOIN Sales.SalesOrderHeader AS soh
  ON sod.SalesOrderID = soh.SalesOrderID
WHERE soh.OrderDate BETWEEN '2003-03-27' AND '2003-03-31'

UNION ALL

SELECT 2,
       1,
       p.ProductID,
       soh.SalesOrderID,
       soh.SalesOrderID,
       soh.OrderDate
FROM Production.Product p
JOIN Sales.SalesOrderDetail AS sod
  ON p.ProductID = sod.ProductID
JOIN Sales.SalesOrderHeader AS soh
  ON sod.SalesOrderID = soh.SalesOrderID
WHERE soh.OrderDate BETWEEN '2003-03-27' AND '2003-03-31'
ORDER BY [Product!1!ProductID], [Order!2!OrderID!id], [Product!1!OrderList!idrefs]
FOR XML EXPLICIT, XMLDATA

```

Note that I've expanded the date range a bit to make sure that there are multiple product IDs for a given range so you see the proper many to many relationship.

The schema winds up looking an awful lot like the one we got for `idref`:

```
<Schema name="Schema4" xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:dt="urn:schemas-microsoft-com:datatypes">
    <ElementType name="Product" content="mixed" model="open">
        <AttributeType name="ProductID" dt:type="i4"/>
```

```
<AttributeType name="OrderList" dt:type="idrefs"/>
<attribute type="ProductID"/>
<attribute type="OrderList"/>
</ElementType>
<ElementType name="Order" content="mixed" model="open">
    <AttributeType name="OrderID" dt:type="id"/>
    <AttributeType name="OrderDate" dt:type="dateTime"/>
    <attribute type="OrderID"/>
    <attribute type="OrderDate"/>
</ElementType>
</Schema>
```

But the elements couldn't be much more different:

```
<Product xmlns="x-schema:#Schema4" ProductID="763" OrderList="49790 49797">
    <Order OrderID="49790" OrderDate="2003-03-28T00:00:00"/>
    <Order OrderID="49797" OrderDate="2003-03-29T00:00:00"/>
</Product>
```

Using `id`, `idref`, and `idrefs` is very complex. Still, they allow you to make your output strongly typed. For most situations, this level of control and the hassles that go with it simply aren't necessary but, when they are, these three can be lifesavers.

xmltext

`xmltext` expects the content of the column to be XML and attempts to insert it as an integral part of the XML document you are creating.

While, on the surface, that may sound simple enough ("Okay, so they're inserting some text in the middle—big deal!"), the rules of where, when, and how it inserts the data are a little strange:

- ❑ As long as the XML you're trying to insert is well formed, the root element will be stripped out—but the attributes of that element will be retained and applied depending on the following few rules.
- ❑ If you did not specify an attribute name when using the `xmltext` directive, then the retained attributes from the stripped element will be added to the element that contains the `xmltext` directive. The names of the retained attributes will be used in the combined element. If any attribute names from the retained attribute data conflict with other attribute information in the combine element, then the conflicting attribute is left out from the retained data.
- ❑ Any elements nested inside the stripped element will become nested elements of the combined element.
- ❑ If an attribute name is provided with the `xmldata` directive, then the retained data is placed in an element of the supplied name. The new element becomes a child of the element that made the directive.
- ❑ If any of the resulting XML is not well formed, there is no defined behavior. Basically, the behavior will depend on how the end result looks, but I would figure that you're going to get an error (I haven't seen an instance where you can refer to data that is not well formed and escape without an error).

cdata

The term `cdata` is a holdover from DTDs and SGML. (SGML is an old markup language, used in the graphics industry that is the ancestor of both HTML and XML. DTDs are type definition documents that outline rules that your SGML (and later, HTML and XML) documents had to live up to.) Basically, `cdata` stands for character data. XML acknowledges a `cdata` section as something of a no man's land—it completely and in all ways ignores whatever is included inside a properly marked `cdata` section. Since there is no validation on the data in a `cdata` section, no encoding of the data is necessary. You would use `cdata` anytime you need your data completely untouched (you can't have encoding altering the data) or, frankly, when you want to move the data but have no idea what the data is (so you can't know if it's going to cause you problems or not).

For this one, we'll just take a simple example—the `AdventureWorks Production.Document` table. This table has a field that has a `nvarchar(max)` data type. The contents are basically unknown. A query to generate the notes on employees into XML might look something like this:

```
SELECT 1                               as Tag,
       NULL                             as Parent,
       DocumentID                      as [Document!1!DocumentID],
       DocumentSummary                 as [Document!1!!cdata]
  FROM Production.Document Document
 WHERE DocumentID BETWEEN 6 AND 8
 ORDER BY [Document!1!DocumentID]
 FOR XML EXPLICIT
```

The output is pretty straightforward:

```
<Document DocumentID="6">
<![CDATA[Guidelines and recommendations for lubricating the required components of
your Adventure Works Cycles bicycle. Component lubrication is vital to ensuring a
smooth and safe ride and should be part of your standard maintenance routine.
Details instructions are provided for each bicycle component requiring regular
lubrication including the frequency at which oil or grease should be applied.
]]>
</Document><Document DocumentID="7"><![CDATA[It is important that you maintain your
bicycle and keep it in good repair. Detailed repair and service guidelines are
provided along with instructions for adjusting the tightness of the suspension fork.

]]>
</Document>
<Document DocumentID="8">
<![CDATA[Worn or damaged seats can be easily replaced following these simple
instructions. Instructions are applicable to these Adventure Works Cycles models:
Mountain 100 through Mountain 500. Use only Adventure Works Cycles parts when
replacing worn or broken components.

]]>
</Document>
```

Basically, this was a pretty easy one.

PATH

Now let's switch gears just a little bit and get down to a more "real" XML approach to getting data.

While `EXPLICIT` has not been deprecated as yet, make no mistake — `PATH` is really *meant* to be a better way of doing what `EXPLICIT` originally was the only way of doing. `PATH` makes a lot of sense in a lot of ways, and it is how I recommend that you do complex XML output in most casts.

This is a more complex recommendation than it might seem. The Microsoft party line on this is that PATH is easier. Well, PATH is easier in many ways, but, as we're going to see, it has its own set of "except for this, and except for that, and except for this other thing" that can twist your brain into knots trying to understand exactly what to do. In short, in some cases, EXPLICIT is actually easier if you don't know XPath. The thing is, if you're dealing with XML, then XPath should be on your learn list anyway, so, if you're going to know it, you should find the XPath-based approach more usable.

Note, however, that if you need backward compatibility to SQL Server 2000, then you're going to need to stick with EXPLICIT.

In its most straightforward sense, the `PATH` option isn't that bad at all. So, let's start by getting our feet wet by focusing in on just the basics of using `PATH`. From there, we'll get a bit more complex and show off some of what `PATH` has to offer.

PATH 101

With `PATH`, you have a model that molds an existing standard to get at your data — XPath. XPath has an accepted standard, and provides a way of pointing at specific points in your XML schema. For `PATH`, we're just utilizing a lot of the same rules and ideas in order to say how data should be treated in a native XML sort of way.

How `PATH` treats the data you refer to depends on a number of rules, including whether the column is named or unnamed (like `EXPLICIT`, the alias is the name if you use an alias). If the column does have a name, then a number of additional rules are applied as appropriate.

Let's look at some of the possibilities.

Unnamed Columns

Data from a column that is not named will be treated as raw text within the row's element. To demonstrate this, let's take a modified version of the example we used for `XML RAW`. What we're doing here is listing the two customers we're interested in and the number of orders they have placed:

```
SELECT c.ContactID,
       COUNT(soh.SalesOrderID)
  FROM Person.Contact c
 JOIN Sales.SalesOrderHeader soh
    ON c.ContactID = soh.ContactID
 WHERE c.ContactID = 1 OR c.ContactID = 2
 GROUP BY c.ContactID
 FOR XML PATH
```

Check the output from this:

```
<row><ContactID>1</ContactID>7</row>
<row><ContactID>2</ContactID>4</row>
```

What it created is a row element for each row in the query—much as you had with RAW—but notice the difference in how it treated our column data.

Since the ContactID column was named, it was placed in its own element (we'll explore this more in the next section)—notice, however, the number 7 in the results. This is just loose embedded text for the row element—it isn't even associated directly with the ContactID since it is outside the ContactID element.

I feel like I'm repeating myself for the 5,000th time on saying this, but, again, remember that the exact counts (7s in this case) that come back may vary on your system depending on how much you have been playing with the data. The key thing is to see how the counts are not associated with the ContactID but are instead just raw text associated with the row.

My personal slant on this is that the number of situations where loose text at the level of the top element is a valid way of doing things is pretty limited. The rules do say you can do it, but I believe it makes for data that is not very clear. Still, this is how it works—use it as it seems to fit the needs of your particular system.

Named Columns

This is where things get considerably more complex rather quickly. In its most simple form, named columns are just as easy as unnamed were—indeed, we saw one of them in our previous example. If a column is a simple named column using PATH, then it is merely added as an additional element to the row.

```
<row><ContactID>1</ContactID>7</row>
```

Our ContactID column was a simple named column.

We can, however, add special characters into our column name to indicate that we want special behaviors for this column. Let's look at a few of the most important.

@

No, that's not a typo—the “@” symbol is really the heading to this section. If we add an @ sign to our column name, then SQL Server will treat that column as an attribute of the previous column. Let's move the ContactID to be an attribute of the top element for the row:

```
SELECT c.ContactID AS '@ContactID',
       COUNT(soh.SalesOrderID)
  FROM Person.Contact c
 JOIN Sales.SalesOrderHeader soh
    ON c.ContactID = soh.ContactID
 WHERE c.ContactID = 1 OR c.ContactID = 2
 GROUP BY c.ContactID
 FOR XML PATH
```

Chapter 16

Yields:

```
<row ContactID="1">7</row>
<row ContactID="2">4</row>
```

Notice that our order count remained a text element of the row — only the column that we identified as an attribute moved in. We could take this to the next step by naming our count and prefixing it to make it an attribute also:

```
SELECT c.ContactID AS '@ContactID',
       COUNT(soh.SalesOrderID) AS '@OrderCount'
  FROM Person.Contact c
 JOIN Sales.SalesOrderHeader soh
   ON c.ContactID = soh.ContactID
 WHERE c.ContactID = 1 OR c.ContactID = 2
 GROUP BY c.ContactID
 FOR XML PATH
```

With this, we no longer have our loose text for the element:

```
<row ContactID="1" OrderCount="7"/>
<row ContactID="2" OrderCount="4"/>
```

Also notice that SQL Server was smart enough to realize that everything was contained in attributes — with no lower-level elements or simple text, it chose to make it a self-closing tag (see the “/” at the end of the element).

So, why did I indicate that this stuff was tricky? Well, there are a lot of different “it only works if . . .” kind of rules here. To demonstrate this, let’s make a simple modification to our original query. This one seems like it should work, but SQL Server will throw a hissy fit if you try to run it:

```
SELECT c.ContactID,
       COUNT(soh.SalesOrderID) AS '@OrderCount'
  FROM Person.Contact c
 JOIN Sales.SalesOrderHeader soh
   ON c.ContactID = soh.ContactID
 WHERE c.ContactID = 1 OR c.ContactID = 2
 GROUP BY c.ContactID
 FOR XML PATH
```

What I’ve done here is to go back to `ContactID` as its own element. What, at first glance, you would expect to happen is to get a `ContactID` element with `OrderCount` as an attribute, but it doesn’t quite work that way:

```
Msg 6852, Level 16, State 1, Line 1
Attribute-centric column '@OrderCount' must not come after a non-attribute-centric
sibling in XML hierarchy in FOR XML PATH.
```

The short rendition of the “What’s wrong?” answer is that it doesn’t really know what it’s supposed to be an attribute of — is it an attribute of the row, or an attribute of the `ContactID`?

/

Yes, a forward slash. Much like @, this special character indicates special things you want done. Essentially, you use it to define something of a path—a hierarchy that relates an element to those things that belong to it. It can exist anywhere in the column name except the first character. To demonstrate this, we’re going to utilize our last (failed) example and build into what we were looking for when we got the error.

First, we need to alter the OrderID to have information on what element it belongs to:

```
SELECT c.ContactID,
       COUNT(soh.SalesOrderID) AS 'ContactID/OrderCount'
  FROM Person.Contact c
  JOIN Sales.SalesOrderHeader soh
    ON c.ContactID = soh.ContactID
 WHERE c.ContactID = 1 OR c.ContactID = 2
 GROUP BY c.ContactID
 FOR XML PATH
```

By adding the “/” and then placing ContactID before the slash, we are telling SQL Server that OrderCount is below ContactID in a hierarchy. Now, there are many ways XML hierarchy can be structured, so let’s see what SQL Server does with this:

```
<row><ContactID>1<OrderCount>7</OrderCount></ContactID></row>
<row><ContactID>2<OrderCount>4</OrderCount></ContactID></row>
```

Now, if you recall, we wanted to make OrderCount an attribute of ContactID, so, while we have OrderCount below ContactID in the hierarchy, it’s still not quite in the place we wanted it. To do that, we can combine / and @, but we need to fully define all the hierarchy. Now, since I suspect this is a bit confusing, let’s take it in two steps—first, the way we might be tempted to do it, but that will yield a similar error to the earlier example:

```
SELECT c.ContactID,
       COUNT(soh.SalesOrderID) AS 'ContactID/@OrderCount'
  FROM Person.Contact c
  JOIN Sales.SalesOrderHeader soh
    ON c.ContactID = soh.ContactID
 WHERE c.ContactID = 1 OR c.ContactID = 2
 GROUP BY c.ContactID
 FOR XML PATH
```

Error time:

```
Msg 6852, Level 16, State 1, Line 1
Attribute-centric column 'ContactID/@OrderCount' must not come after a non-attribute-
centric sibling in XML hierarchy in FOR XML PATH.
```

To fix this, we need to understand a bit about how things are constructed when building the XML tags. The key is that the tags are essentially built in the order you list them. So, if you want to add attributes to an element, you need to keep in mind that they are part of the element tag—that means you need to define any attributes before you define any other content of that element (subelements or raw text).

Chapter 16

In our case, we are putting the `ContactID` as being raw text, but the `OrderCount` as being an attribute (okay, backward from what would be likely in real life, but hang with me here). This means we are telling SQL Server things backward. By the time it sees the `OrderCount` information, it is already done with attributes for `ContactID` and can't go back.

So, to fix things, we simply need to tell it about the attributes before we tell it about any more elements or raw text:

```
SELECT COUNT(soh.SalesOrderID) AS 'ContactID/@OrderCount',
       c.ContactID
  FROM Person.Contact c
 JOIN Sales.SalesOrderHeader soh
    ON c.ContactID = soh.ContactID
 WHERE c.ContactID = 1 OR c.ContactID = 2
 GROUP BY c.ContactID
 FOR XML PATH
```

This probably seems counterintuitive, but, again, think of the order things are being written in. The attributes are written first and then, and only then, can we write the lower-level information for the `CustomerID` element. Run it, and you'll see we get what we were after:

```
<row><ContactID OrderCount="7">1</ContactID></row>
<row><ContactID OrderCount="4">2</ContactID></row>
```

The `OrderCount` has now been moved into the attribute position, just as we desired, and the actual `CustomerID` is still raw text embedded in the element.

Follow the logic of the ordering of what you ask for a bit, because it works for most everything. So, if we wanted `CustomerID` to also be an attribute rather than raw text, but wanted it to be after `OrderCount`, we could do that—we just need to make sure that it comes after the `OrderCount` definition.

But Wait, There's More . . .

As I said earlier, XPath has its own complexity and is a book's worth to itself, but I don't want to leave you with just what I said in the preceding sections and say that's all there is.

`@` and `/` will give you a great deal of flexibility in building the XML output just the way you want it, and probably meet the need well for most simple applications. If, however, you need something more, then there is still more out there waiting for you. For example, you can:

- “Wildcard” data such that it’s all run together as text data without being treated as separate columns
- Embed native XML data from XML data type columns
- Use XPath node tests—these are special XPath directives that change the behavior of your data
- Use the `data()` directive to allow multiple values to be run together as one data point in the XML
- Utilize namespaces

OPENXML

We've spent pages and pages dealing with how to turn our relational data into XML. It seems reasonably intuitive then that SQL Server must also allow you to open a string of XML and represent it in the tabular format that is expected in SQL.

OPENXML is a rowset function that opens your string much as other rowset functions (such as OPENQUERY and OPENROWSET) work. This means that you can join to an XML document, or even use it as the source of input data by using an `INSERT .. SELECT` or a `SELECT INTO`. The major difference is that it requires you to use a couple of system stored procedures to prepare your document and clear the memory after you're done using it.

To set up your document, you use `sp_xml_preparedocument`. This moves the string into memory and pre-parses it for optimal query performance. The XML document will stay in memory until you explicitly say to remove it or you terminate the connection that `sp_xml_preparedocument` was called on.

Let me digress a moment and say that I'm not at all a fan of letting a system clean up for you. If you instantiate something, then you should proactively clean it up when you're done (if only I could teach my youngest child this when she pulls out her toys!).

Much like Visual Basic, C#, and most other languages are supposed to clean up your objects when they go out of scope for you, SQL Server is supposed to clean up your prepared documents. Please do not take the lazy approach of relying on this—clean up after yourself! By explicitly deallocating it (using `sp_xml_removedocument`), you are making certain the clean up happens, clearing it from memory slightly sooner, and also making it very clear in your code that you're done with it.

The syntax is pretty simple:

```
sp_xml_preparedocument @hdoc = <integer variable> OUTPUT,
[, @xmltext = <xml>]
[, @xpath_namespaces = <url to a namespace>]
```

Note that, if you are going to provide a namespace URL, you need to wrap it in the "<" and ">" symbols at both ends (for example, "<root xmlns:sql = "run: schemas-microsoft-com:xml-sql">").

The parameters of this sproc are fairly self-describing:

- ❑ `@hdoc`—If you've ever programmed to the Windows API (and to tons of other things, but this is a common one), then you've seen the “h” before—it's Hungarian notation for a handle. A handle is effectively a pointer to a block of memory where something (could be about anything) resides. In our case, this is the handle to the XML document that we've asked SQL Server to parse and hold onto for us. This is an output variable—the variable you reference here will, after the sproc returns, contain the handle to your XML. Be sure to store it away, as you will need it when you make use of OPENXML.
- ❑ `@xmltext`—Is what it says it is—the actual XML that you want to parse and work with.
- ❑ `@xpath_namespaces`—Any namespace reference(s) your XML needs to operate correctly.

Chapter 16

After calling this sproc and saving away the handle to your document, you're ready to make use of OPENXML. The syntax for it is slightly more complex:

```
OPENXML(<handle>,
    <XPath to base node>
    [, <mapping flags>])
[WITH (<schema Declaration>|<table Name>)]
```

We have pretty much already discussed the handle — this is going to be an integer value that you received as an output parameter for your `sp_xml_preparedocument` call.

When you make your call to OPENXML, you must supply the XPath to a node that will serve as a starting point for all your queries. The schema declaration can refer to all parts of the XML document by navigating relative to the base node you set here.

Next up are the mapping flags. These assist us in deciding whether you want to favor elements or attributes in your OPENXML results. The options are:

Byte Value	Description
0	Same as 1 except that you can't combine it with 2 or 8 (2 + 0 is still 2). This is the default.
1	Unless combined with 2 (described next), only attributes will be used. If there is no attribute with the name specified, then a <code>NULL</code> is returned. This can also be added to either 2 or 8 (or both) to combine behavior, but this option takes precedence over option 2. If <code>XPath</code> finds both an attribute and an element with the same name, the attribute wins.
2	Unless combined with 1 (described previously), only elements will be used. If there is no element with the name specified, then a <code>NULL</code> is returned. This can also be added to either 1 or 8 (or both) to combine behavior. If combined with 1, then the attribute will be mapped if it exists. If no attribute exists, then the element will be used. If no element exists, then a <code>NULL</code> is returned.
8	Can be combined with 1 or 2 (described previously). Consumed data should not be copied to the overflow property <code>@mp:xmltext</code> (you would have to use the metaproPERTY schema item to retrieve this). If you're not going to use the metaproPERTIES — and most of the time you won't be — I recommend this option. It cuts a small (okay, <i>very</i> small) amount of overhead out of the operation.

Finally comes the schema or table. If you're defining a schema and are not familiar with XPath, this part can be a bit tricky. Fortunately, this particular XPath use isn't very complex and should become second nature fairly quickly (it works a lot like directories do in Windows, only with a lot more power).

The schema can vary somewhat in the way you declare it. The definition is declared as:

```
WITH (
    <column name> <data type> [{<column XPath>}|<metaproPERTY>]
    [,<column name> <data type> [{<column XPath>}|<metaproPERTY>]]
```

- ❑ The column name is just that—the name of the attribute or element you are retrieving. This will also serve as the name you refer to when you build your `SELECT` list, perform `JOINS`, and the like.
- ❑ The data type is any valid SQL Server data type. Since XML can have data types that are not equivalents of those in SQL Server, an automatic coercion will take place if necessary, but this is usually predictable.
- ❑ The column `XPath` is the `XPath` pattern (relative to the node you established as the starting point for your `OPENXML` function) that gets you to the node you want for your column—whether an element or attribute gets used is dependent on the `flags` parameter as described above. If this is left off, then SQL Server assumes you want the current node as defined as the starting point for your `OPENXML` statement.
- ❑ Metaproperties are a set of special variables that you can refer to in your `OPENXML` queries. They describe various aspects of whatever part of the XML DOM you're interested in. To use them, just enclose them in single quotes and put them in the place of the column `XPath`. Available metaproperties include:
 - ❑ `@mp:id`—Don't confuse this with the XML `id` that we looked at with `EXPLICIT`. While this property serves a similar function, it is a unique identifier (within the scope of the document) of the DOM node. The difference is that this value is system generated—as such, you can be sure it is there. It is guaranteed to refer to the same XML node as long as the document remains in memory. If the `id` is zero, it is the root node (its `@mp:parentid` property, as referred to below, will be `NULL`).
 - ❑ `@mp:parentid`—This is the same as the preceding, only for the parent.
 - ❑ `@mp:localname`—Provides the non-fully qualified name of the node. It is used with a prefix and namespace URI (Uniform Resource Identifier—you'll usually see it starting with URN) to name element or attribute nodes.
 - ❑ `@mp:parentlocalname`—This is the same as the preceding, only for the parent.
 - ❑ `@mp:namespaceuri`—Provides the namespace URI of the current element. If the value of this attribute is `NULL`, no namespace is present.
 - ❑ `@mp:parentnamespaceuri`—This is the same as the preceding, only for the parent.
 - ❑ `@mp:prefix`—Stores the namespace prefix of the current element name.
 - ❑ `@mp:parentprefix`—This is the same as the preceding, only for the parent.
 - ❑ `@mp:prev`—Stores the `mp:id` of the previous sibling relative to a node. Using this, you can tell something about the ordering of the elements at the current level of the hierarchy. For example, if the value of `@mp:prev` is `NULL`, then you are at the first node for this level of the tree.
 - ❑ `@mp:xmltext`—This metaproperty is used for processing purposes, and contains the actual XML for the current element.

Of course, you can always save yourself a ton of work by bypassing all these parameters. You get to do this if you have a table that directly relates (names and data types) to the `XPath` starting point that you've specified in your XML. If you do have such a table, you can just name it and SQL Server will make the translation for you!

Chapter 16

Okay, that's a lot to handle, but we're not quite finished yet. You see, when you're all done with your XML, you need to call `sp_xml_removedocument` to clean up the memory where your XML document was stored. Thankfully, the syntax is incredibly easy:

```
sp_xml_removedocument [hdoc = ]<handle of XML doc>
```

Again, I can't stress enough how important it is to get in the habit of always cleaning up after yourself. I know that, in saying that, I probably sound like your mother. Well, like your mother, SQL Server will clean up after you some, but, like your mother, you can't count on SQL Server to clean up after you every time. SQL Server will clean things up when you terminate the connection, but what if you are using connection pooling? Some connections may never go away if your system is under load. It's an easy sproc to implement, so do it—every time!

Okay, I'm sure you've been waiting for me to get to how you really make use of this—so now it's time for the all-important example.

Imagine that you are merging with another company and need to import some of their data into your system. For this example, we'll say that we're working on importing a few shipping providers that they have and our company doesn't. A sample of what our script might look like to import these from an XML document might be:

```
USE AdventureWorks

DECLARE @idoc      int
DECLARE @xmldoc    nvarchar(4000)

-- define the XML document
SET @xmldoc = '
<ROOT>
<Shipper CompanyName="Billy Bob's Pretty Good Shipping" Base="4.50"
Rate="1.05"/>
<Shipper CompanyName="Fred's Freight" Base="3.95" Rate="1.29"/>
</ROOT>
'

PRINT @xmldoc
--Load and parse the XML document in memory
EXEC sp_xml_preparedocument @idoc OUTPUT, @xmldoc

--List out what our shippers table looks like before the insert
SELECT * FROM Purchasing.ShipMethod

--See our XML data in a tabular format

SELECT * FROM OPENXML (@idoc, '/ROOT/Shipper', 0) WITH (
    CompanyName      nvarchar(40),
    Base            decimal(5,2),
    Rate             decimal(5,2))

--Perform and insert based on that data
INSERT INTO Purchasing.ShipMethod
(Name, ShipBase, ShipRate)
```

```

SELECT * FROM OPENXML (@idoc, '/ROOT/Shipper', 0) WITH (
    CompanyName      nvarchar(40),
    Base            decimal(5,2),
    Rate             decimal(5,2))

--Now look at the Shippers table after our insert
SELECT * FROM Purchasing.ShipMethod

--Now clear the XML document from memory
EXEC sp_xml_removedocument @idoc

```

The final result set from this looks just like what we wanted:

ShipMethodID	Name	ShipBase	ShipRate
1	XRQ - TRUCK GROUND	3.95	0.99
2	ZY - EXPRESS	9.95	1.99
3	OVERSEAS - DELUXE	29.95	2.99
4	OVERNIGHT J-FAST	21.95	1.29
5	CARGO TRANSPORT 5	8.99	1.49
6	Billy Bob's Pretty Good Shipping	4.50	1.05
7	Fred's Freight	3.95	1.29

It isn't pretty, but it works — XML turned into relational data.

A Quick (Very Quick) Reminder of XML Indexes

We covered this back in Chapter 8, but we didn't really discuss the XML data type at the time, nor did we discuss all the various options and technologies surrounding XML. With that in mind, I wanted to make sure that I pointed the new XML indexes feature out to you again when it's in the context of XML.

XML indexes are new with SQL Server 2005, and, in simple terms: They ROCK! They are not the answer to everything under the sun, but they give us a way of staying native XML in situations where that's best while maintaining some degree of performance.

XML indexes are fully discussed in Chapter 8.

HTTP Endpoints

This is one of those “should I or shouldn't I” topics to me. The setup, configuration, security, and many other aspects of HTTP endpoints are largely in the administrator's realm. The problem is that the notion of using them as a native Web service is most definitely a developer's realm item. So, I'll take a middle ground here: we'll discuss what they are and get a concept for their importance in design but shy away from some of the administrative aspects.

HTTP endpoints are not supported in SQL Server Express. Also, Windows XP does not support some of the necessary underlying technologies. So, if you or your application will be using SQL Server Express or Windows XP, you will need to stay away from HTTP endpoints.

So then, what are *HTTP endpoints*? Well, it is a concept that comes from the world of Simple Object Access Protocol (SOAP). The idea behind SOAP is to get around many of the firewall issues that exist when you try to get to services over the Web. Prior to SOAP, there would often need to be some special port opened up, and binary calls were often made between objects. The problems with security were huge, and IT departments rebelled at the notion of opening their firewalls up to such access.

SOAP provides for the idea of making a request over normal HTTP. You supply a SOAP packet, and it contains a method request along with any parameters required. Since the SOAP packet is just a bunch of text containing the information needed to understand and execute a request, rather than an active binary making the request, it is entirely up to the receiving connection to decide what to do with that packet.

I'm going to digress right here and make clear that the opening of an HTTP endpoint to the Web is still no small thing security-wise. As I indicated at the start of this topic, the actual setup and configuration of HTTP endpoints are implementation specific and very, very much in the administrator's realm for this very reason — there are major firewall and security considerations when setting these up. They are defined at the server, rather than the database level, so when you create them you are putting the server at risk—not just your particular application's databases.

Make sure that you understand the environment(s) these will be used in before you bet your application on them. Don't shy away from them, but make sure you understand how and why you're going to use them.

HTTP endpoints are largely used to create Web services that are native to your SQL Server. Your SQL Server can accept and handle the Web request directly. These will typically be information requests from your database. You can, of course, control how much and in what way the information is made available.

Security

Now, after creating some panic earlier about security risks surrounding HTTP endpoints, I'll calm things down a bit and say there is a lot that can be done here. Access can be granted to control specific endpoints, and limits can be imposed on an endpoint by endpoint basis. This means that you can expose things through an endpoint, while keeping risk reasonable. Some examples of what level permissions can be granted include:

- Control** — This controls whether a given user can modify a specific endpoint or even drop it.
- View definition** — This controls whether the user can even see information (metadata) about an endpoint.
- Connect** — This controls whether the user can even see the data on the endpoint. It's worth noting that you can grant a user the right to manage an endpoint without giving them the right to view the data on that endpoint (very nice for applications where you need to allow someone some administrative rights but do not want them to have access to the sensitive information the endpoint exposes).

HTTP Endpoint Methods

I had to debate whether to even make this its own section, because these really are nothing by themselves. All you need to have a method available on a HTTP endpoint is have some code that can be referenced.

I'll set up a specific example for use when we create our actual endpoint shortly. In this case, we'll just use a stored procedure to provide information that might be common for a Web service model—a list of bicycles we have available:

```
USE AdventureWorks
GO
CREATE PROCEDURE spListBikes
AS
SELECT p.ProductID, p.ProductNumber, p.Name, p.ListPrice
FROM Production.Product AS p
JOIN Production.ProductInventory AS i
    ON p.ProductID = i.ProductID
JOIN Production.ProductSubcategory psc
    ON p.ProductSubcategoryID = psc.ProductSubcategoryID
JOIN Production.ProductCategory pc
    ON psc.ProductCategoryID = pc.ProductCategoryID
WHERE p.ListPrice > 0
    AND p.DiscontinuedDate IS NULL
    AND pc.Name = 'Bikes'
ORDER BY p.Name
```

And, for the sake of variety, we'll add an additional stored procedure to support inventory inquiries.

```
CREATE PROCEDURE spGetProductInventory
    @ProductID int
AS
BEGIN
    SELECT p.ProductID, p.ProductNumber,
        p.Name,
        i.Quantity
    FROM Production.Product AS p
        INNER JOIN Production.ProductInventory AS i
            ON p.ProductID = i.ProductID
    WHERE p.ProductID = @ProductID
END
```

So, all we've done is created a couple of stored procedures, but, as we'll see shortly, we've actually also created the basis for some Web service methods.

Creating and Managing a HTTP Endpoint

Endpoints go well beyond just HTTP endpoints, but, as I've said before, fall largely in the administrator's realm. That said, they follow the basic CREATE <object type> <object name> syntax we've seen so often, but with a couple of twists. We'll stick to a basic HTTP endpoint example (I'm not even going to list the full syntax, as most of it is off topic for this book, and it is very convoluted) by exposing the procedures we created in the previous section:

Chapter 16

```
CREATE ENDPOINT EP_AW
    STATE = STARTED
AS HTTP
(
    PATH = '/AW',
    AUTHENTICATION = (INTEGRATED),
    PORTS = (CLEAR),
    SITE = 'localhost'
)
FOR SOAP
(
    WEBMETHOD 'ListBikes'
        (NAME='AdventureWorks.dbo.spListBikes'),
    WEBMETHOD 'ProductStockInfo'
        (NAME='AdventureWorks.dbo.spGetProductInventory'),
    BATCHES = DISABLED,
    WSDL = DEFAULT,
    DATABASE = 'AdventureWorks',
    NAMESPACE = 'http://Adventure-Works/'
)
```

This creates a HTTP endpoint that exposes our two stored procedures. Congratulations, you've just created your first Web service.

Just to get the most basic indication that our Web service really is there, try navigating to `http://localhost/aw?wsdl`. You should get a bunch of XML spewed back at you—this is essentially a document defining what is available on this Web service. If you scan toward the bottom of it, you should be able to find references to the methods we defined, for example:

```
- <xsd:element name="ListBikes">
  - <xsd:complexType>
    <xsd:sequence />
  </xsd:complexType>
</xsd:element>
```

This node is defining our `ListBikes` method, which, if invoked, would call our `spListBikes` stored procedure and return the resulting data.

If you are familiar with utilizing Web services in Dev Studio, you would now be able to make use of our two methods—try them out!

Closing Thoughts

The power of this seems obvious, but I will caution you about getting carried away. Some things to think about:

- I'm not generally a big fan of directly exposing SQL Servers to the Internet. If these Web services are going to be for direct Internet access, consider using a proxy of some sort to sit between the Web and your SQL Server Web services. This doesn't need to be a big deal—it could, for example, be a straight pass-through, but it's one more level of indirection between your data (which is usually quite valuable) and the outside world.

- ❑ For internal applications, you are probably fine with direct access (after all, it's not that much different from what a direct ADO connection would be—indeed, it's potentially more secure), but keep in mind that applications that start off as “internal use only” have a nasty habit of having “expect for <xxxxx>” added to them later. Inventory systems get exposed to affiliates and partners, customers and vendors. Other applications often wind up being important to other outside sources.
- ❑ Consider uptime and redirection requirements. This again leads me to the proxy notion, but if you want to be able to switch back and forth seamlessly between back-end servers, you may want to virtualize what server your applications are talking to via a proxy. That way, if you want to take your SQL Server offline for maintenance, you can redirect at the proxy to a backup database for example (assuming the application isn't doing updates).

Summary

The size of the XML portion of SQL Server has grown considerably since its original introduction as a “Web release” prior to SQL Server 2000, and it continues to grow. XML is one of the most important technologies to hit the industry in the last 20 or more years. It provides a flexible, very transportable way of describing data, and SQL Server now has more and more ways of meeting your XML needs.

In this chapter, we've taken a look at how to get relational data into XML format, and how to get XML data into a relational structure. We've also seen how SQL Server can supply Web service data directly using XML-based methods.

17

Reporting for Duty, Sir!

After all the queries have been written, after all the stored procedures have been run, there remains a rather important thing we need to do in order to make our data useful—make it available to end users.

Reporting is one of those things that seem incredibly simple but turn out to be rather tricky. You see, you can't simply start sticking numbers in front of people's faces—the numbers must make sense and, if at all possible, capture the attention of the person you're reporting for. In order to produce reports that actually get used and, therefore, are useful, there are a couple of things to keep in mind:

- Use just the right amount of data**—Do not try to do too much in one report; nor should you do too little. A report that is a jumble of numbers is going to quickly lose a reader's attention, and you'll find that it doesn't get utilized after the first few times it is generated. Likewise, a barren report will get just a glance and get tossed without any real thought. Find a balance, mixing the right amount of data with the right data.
- Make it appealing**—Sad as it is to say, another important element in reporting is what one of my daughters would call making it "prettiful"—which is to say, making it look nice and pleasing to the eye. An ugly report is a dead report.

In this chapter, we're going to be taking a look at the Reporting Services tools that are new with SQL Server 2005. As with all the "add-on" features of SQL Server that we cover in this book, you'll find the coverage here to be largely something of "a taste of what's possible"—there is simply too much to cover to get it all in one chapter of a much larger book. If you find that this "taste" whets your appetite, consider reading a book dedicated specifically to Reporting Services.

Reporting Services 101

Odds are that you've already generated some reports in your day. They may have been paper reports off a printer (perhaps in something as rudimentary as Access's reporting area—which is actually one of the best parts of Access to me). They may have been off a rather robust reporting engine such as Crystal Reports. Even if you haven't used tools that fancy, one can argue that handing your boss the printout from a stored procedure is essentially a very simple (and not necessarily nice looking) report—I would tend to agree with that argument.

The reality, however, is that our managers and coworkers today expect something more. This is where Reporting Services comes in. Reporting Services really has two different varieties of operation:

- **Report models**—This is making use of a relatively simple, Web-driven interface that is meant to allow end users to create their own simple reports.
- **Reports generated in Visual Studio**—While this doesn't necessarily mean you have to write code (you can actually create simple reports using drag and drop functionality—something we'll do in this chapter as an example—you can design fairly robust reports.

Note that, while your users can eventually access these reports from the same Reporting Services Web host, they are based on completely different architectures (and, as you will see, are created in much different fashions).

In addition, Reporting Services provides features for pre-generating reports (handy if the queries that underlie the report take a while to run) as well as distributing the report via e-mail.

Building Simple Report Models

To create a simple model, start by opening Business Intelligence Studio.

Note that this is entirely different from SQL Server Management Studio, which we've been working with thus far. Business Intelligence Studio is a different work area that is highly developer (rather than administrator) focused, and is oriented around many of the "extra" services that SQL Server has beyond the base relational engine that has been our focus until now. In addition to the work we'll do with Business Intelligence Studio in this chapter, we will also visit it some to work with Integration Services in Chapter 19.

Choose File→New Project, and you should bring up a dialog that allows you to choose between several types of Business Intelligence projects. Choose "Report Model Project" and the name of your choice (I'll be going with the rather descriptive ReportModelProject), and click OK. This will bring you to what should look like an everyday, run-of-the-mill Visual Studio development environment.

Note that the exact appearance of this dialog may vary somewhat depending on whether you have Visual Studio installed and, if so, which specific languages and templates you've installed. The preceding image is of the most generic SQL Server-only installation.

If your Visual Studio environment is still in its default configuration, you should see the Solution Explorer on the top right-hand side. Right-click Data Sources and choose Add New Data Source, as shown in Figure 17-1.

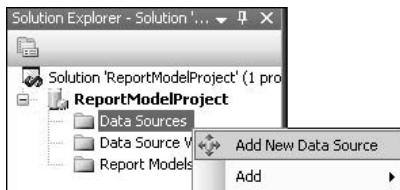


Figure 17-1

This will probably (unless you've already been here before and checked the box saying "Don't show me this page again") bring you up to a Welcome dialog box for the Data Source Wizard. Click Next to get to the start of the meaty stuff—the datasource selection page of the Data Source Wizard. There is, however, one problem—we don't have any datasources as yet (as we see in Figure 17-2).

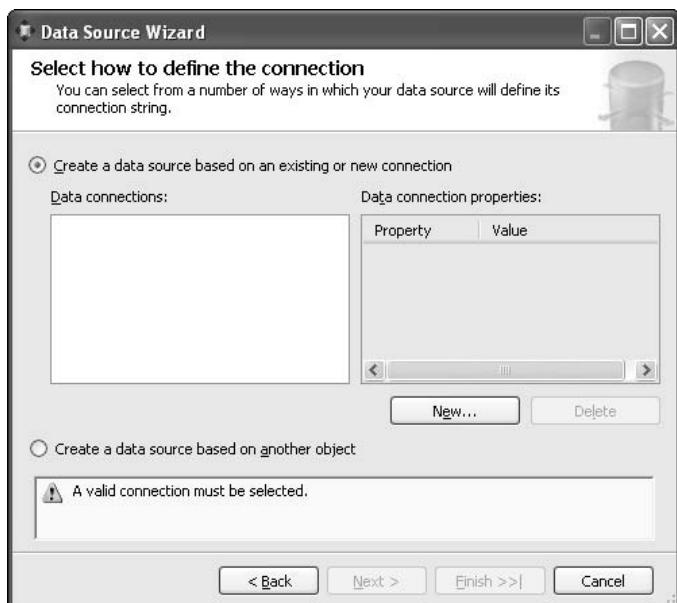


Figure 17-2

It probably goes without saying that without any existing Data Connections to choose from, we really have no choice but to click New.

The first time I saw this next dialog, I was mildly surprised to see that it was a different new connection dialog than had been used repeatedly in Management Studio; nonetheless, it does contain the same basic elements, just in a slightly different visual package (in short, don't worry if it looks a little different).

Chapter 17

This brings us up to the connection manager shown in Figure 17-3.

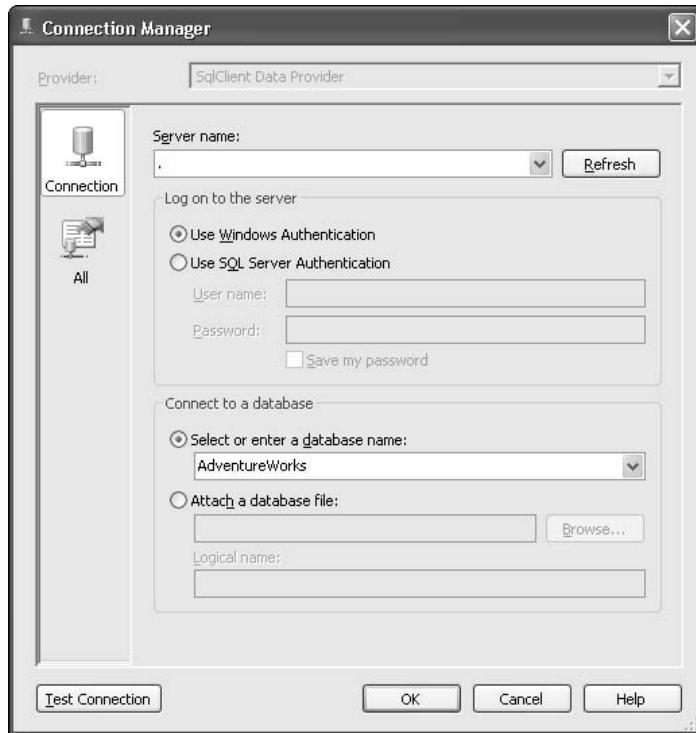


Figure 17-3

While the concepts are the same as we've seen in a few other places in the book, there are one or two new things here, so let's take a look at several key elements of this dialog.

- ❑ **Server name**—This one is what it sounds like and is the same as we've seen before. Name the server you want to connect to for this connection, or, if you're wanting to connect to the default instance of SQL Server for the local server, you can also use the aliases of "(local)" or "." (a simple period).
- ❑ **Windows/SQL Server Authentication**—Choose the authentication type to connect to your server. Microsoft (and, I must admit, me too) would rather you used Windows Authentication, but if your server is not in your domain or if your administrator is not granting rights directly to your Windows login, then you can use an administrator supplied SQL Server-specific username and password (we've used MyLogin and MyPassword on several occasions elsewhere in the book).
- ❑ **Connect to a database**—Now things get even more interesting. Here you can either continue down the logical path of selecting a database on the server you have chosen or you can choose to connect directly to an .mdf file (in which case the SQL Server Express engine takes care of translating for you).

In my case, I've selected the local server (by using a single period as the server name) and our old friend, the AdventureWorks database.

Go ahead and click OK, and we see a different Data Source Wizard dialog than we did the first time—we now have the connection we just created, but it's worth noting that we could, if desired, create several connections and then choose between them.

Click Next, and it brings you to the final dialog of the wizard. Note that the default name happens to be the database we chose, but this is truly just a name for the datasource—we can call this datasource whatever we want, and it would still be connecting to the AdventureWorks database on the local server.

Also take note of how it has built a connection string for us. Connection strings are a fundamental concept in all modern forms of database connectivity—virtually every connectivity model (.NET managed providers, OLEDB, ODBC, for example) uses a connection string at some level. If we had chosen some different options—for example, used a SQL Server username and password instead of Windows security—then our connection string would use a few different parameters and, of course, pass some different values.

Go ahead and click Finish, and we have our datasource for our project, as shown in Figure 17-4.



Figure 17-4

If you ponder Figure 17-4 for a moment, you'll notice that there is a bit more to our Report Model project than just datasources—indeed, we now need to take the next step and create a *Data Source View*.

Data Source Views

A Data Source View has a slight similarity to the views we learned about in Chapter 9. In particular, it can be used to provide users access to data but filter what data they actually see. If we give them access to the entire datasource, then they are able to see all data available to that datasource. Data Source Views allow us to filter the datasource down to just a subset of its original list of available objects.

To add our Data Source View, simply right-click Data Source View and select Add New Data Source View (tricky, isn't it?), as shown in Figure 17-5.

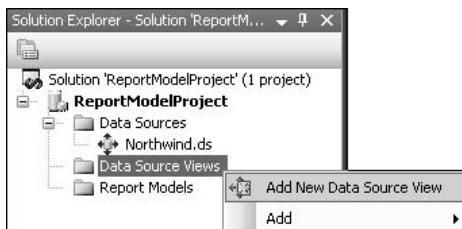


Figure 17-5

Chapter 17

We get another of those silly “Welcome to the wizard” dialogs, so just click next and you’ll arrive at a dialog that you’ve essentially already seen before—one that is basically identical to one we had partway through creating our datasource.

Accept the default, and then click through to the Select Tables and Views dialog shown in Figure 17-6. This one, as the name suggests, allows us to specify which tables and views are going to be available to end users of the report model we’re building.

Start by selecting Person.Contact, and then use the > button to move that into the Included Objects column (you can also just double-click a table to move it to the other side). Now add the Sales.SalesOrderHeader, Sales.SalesOrderDetail, Sales.SpecialOfferProduct, and the Production.Product tables to finish catching up with the tables I have selected in Figure 17-6.

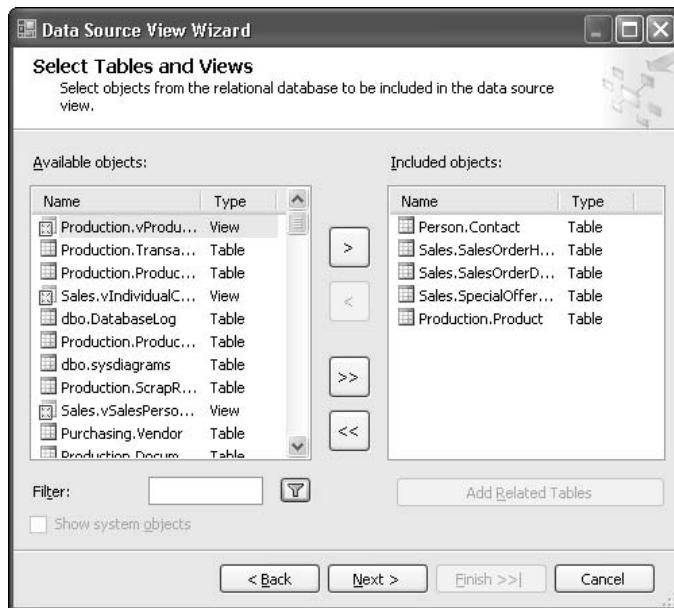


Figure 17-6

Don’t worry about the Filter or Show system objects options in this table—both are pretty harmless. The short rendition is that the filter just filters down the object list to make it easy to deal with databases that have a very large number of objects to show. The Show system objects option is just that—it allows you to include system objects in the report model (which seems pretty silly to me for the vast, vast majority of applications out there).

Click Next, and we’re finally at a dialog that lets us confirm we are done. This dialog is pretty much just synopsizing what’s to be included in this Data Source View before you finalize it and essentially tell SQL Server, “Yeah, that’s really what I mean!” Now click Finish, and the Data Source View is actually constructed and added to our Report Model Project.

Building Your Report Model

At this point, we're ready to create the actual Report Model. As we did with datasource and Data Source View, simply right-click the Report Model node of the Solution Explorer tree and select Add New. At the start of defining a Report Model, we get yet another one of those "welcome" dialogs followed by a dialog that lets us select our datasource view, as shown in Figure 17-7. Not surprisingly, the only one that shows is the one just created (still called AdventureWorks).

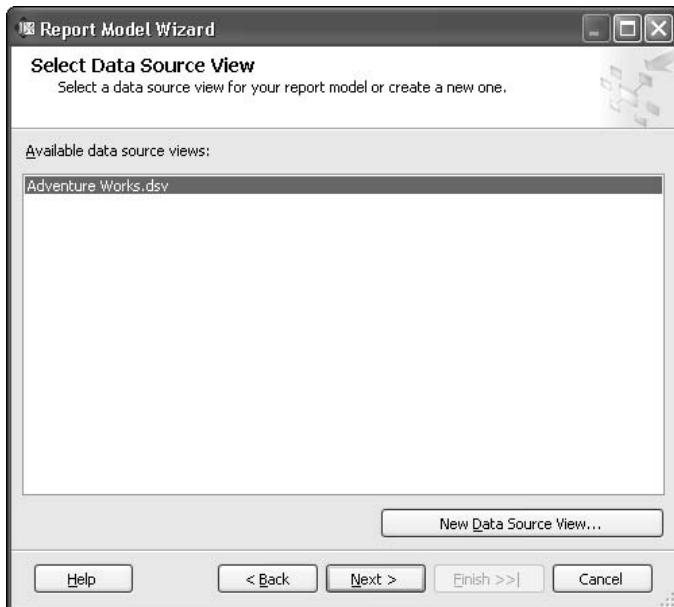


Figure 17-7

Go ahead and select our one datasource view, and click Next to move on to the next dialog (shown in Figure 17-8), the report model generation rules.

The report model generation rules determine things like what rollups are available in your report model, as well as other guides to assist the generation of end user reports. Also make note of your ability to control the default language for your report model (handy for multinational companies or companies where you want to generate reporting for languages beyond the base install for your server).

Go ahead and accept the defaults here by clicking Next, and we move on to the Update Statistics dialog. The Update Statistics dialog gives us two options:

- Update statistics before generating** — This will cause all statistics on all of the tables and indexes referenced in this report model to be updated (see Chapter 8 for more information on table and index statistics) prior to the report model actually being built.
- Use current statistics in the data source view — This just makes use of what's already there. Depending on how many tables and indexes are involved, this can save a lot of time in getting the report generated (since you don't have to wait on all the statistics to be updated), but it runs the risk of the report model making assumptions that are out of date regarding the make-up of your data.

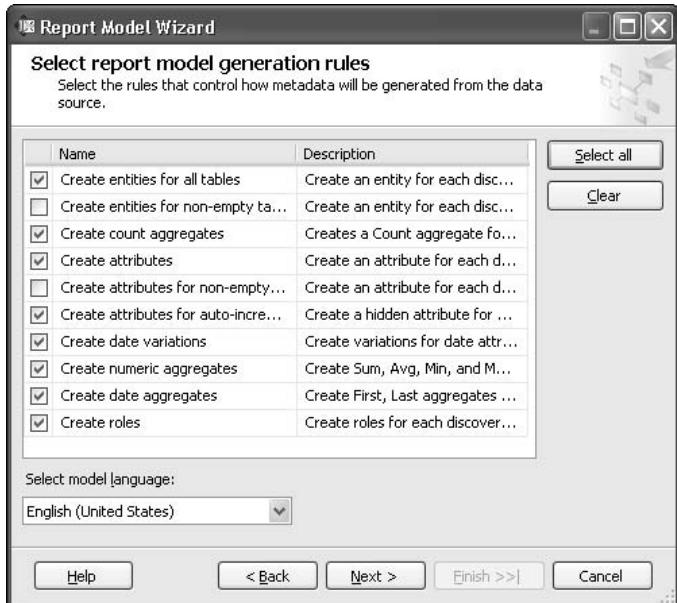


Figure 17-8

I'm going to recommend that you go with the default here and always update your statistics unless you know exactly why you are skipping that and believe you understand the ramifications of doing so. For most installations, not updating the statistics in this scenario is not going to be a big deal, but it is one of those things that can very subtlety reach out and bite you, so better safe than sorry.

The next (and final) dialog is the Completing the Wizard dialog. Name your report model (I've chosen AdventureWorks Report Model—original, eh?) and click Run, and your report model will be generated as shown in Figure 17-9.

Click Finish, and we can begin to see the results of what we just created, which should look something like Figure 17-10.

Be sure to take the time to explore the report model. SQL Server will make assumptions about what you do and do not want to have shown, and it will not always make the right choice (as we'll see in a moment). Look at what it has included, what it has added, and what it has generated in the way of derived attributes (rollups and such).

One could spend an entire chapter just going over the nuances of each entity of the model and all the attributes within it. Take particular note to how SQL Server automatically breaks down well-understood attributes (such as dates) to smaller attributes (say, month, day, and year) based on well understood common uses of the underlying data type.

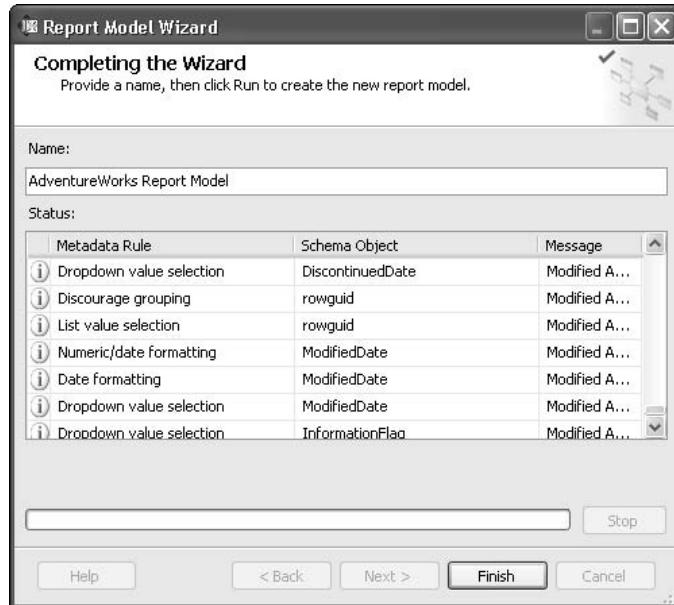


Figure 17-9

Report Model			
	Name	Type	Description
Model	Contact	Entity	
Contact	Product	Entity	
Product	Sales Order Detail	Entity	
Sales Order Detail	Sales Order Header	Entity	
Sales Order Header	Special Offer Product	Entity	
Special Offer Product			

Figure 17-10

Our report model doesn't quite include everything we want in it quite yet, so we're going to need to edit it some.

Start by navigating to the `SalesOrderHeader` table. Notice how the `SalesOrderID` column is grayed out. If you click on it and check the property window shown in Figure 17-11, you should notice that this rather important field has been defined as being hidden.

Change the hidden property to False.

This happens to be one of those instances where the design of the system is utilizing an automatically generated value as something the end user sees. The identity column is system generated but is actually utilized as the SalesOrderID a customer would see on an invoice. Many identity values are artificial, "under the covers" constructs, and thus why SQL Server assumes you wouldn't want to show these in the report.

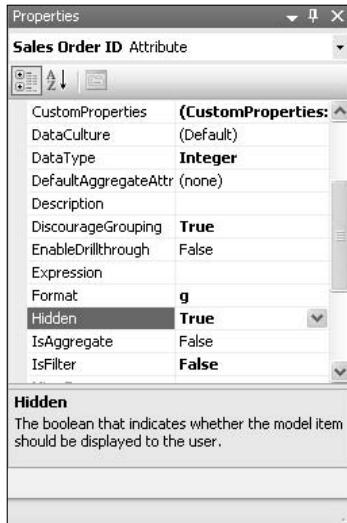


Figure 17-11

Well, building a report model is all well and good, but if no one can get at it, it serves little purpose (much like designing a stored procedure but never actually adding it to your database). To make our report usable to end users, we must *deploy* it.

Deploying Our Model

Fortunately deploying our model couldn't get much easier. Indeed, the hard part is knowing that you need to and then finding the command to do so. To deploy, simply right-click the project in the solution explorer and choose Deploy, or choose Deploy on the Build menu. You can watch the progress of the deploy in the output window of Visual Studio.

```
Build complete -- 0 errors, 0 warnings
----- Build started: Project: ReportModelProject, Configuration: Production -----
Build complete -- 0 errors, 0 warnings
----- Deploy started: Project: ReportModelProject, Configuration: Production -----
Deploying to http://localhost/ReportServer?%2f
Deploying data source '/Data Sources/Adventure Works'.
Deploying model 'AdventureWorks Report Model'.
Deploy complete -- 0 errors, 0 warnings
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
===== Deploy: 1 succeeded, 0 failed, 0 skipped =====
```

Report Creation

Our Report Model is, of course, not itself a report. Instead, it merely facilitates reports (and, considering there are lots of business type people who understand reports, but not databases, facilitating reports can be a powerful thing). Now that it is deployed, we can generate many reports from just the one model.

To see and generate reports, you need to leave Business Intelligence Studio and actually visit the reporting user interface—which is basically just a Web site. Navigate to `http://<your reporting server host>/reports`—in my case, I have it right on my local system, so I can get there by navigating to `http://localhost/reports`, as shown in Figure 17-12.

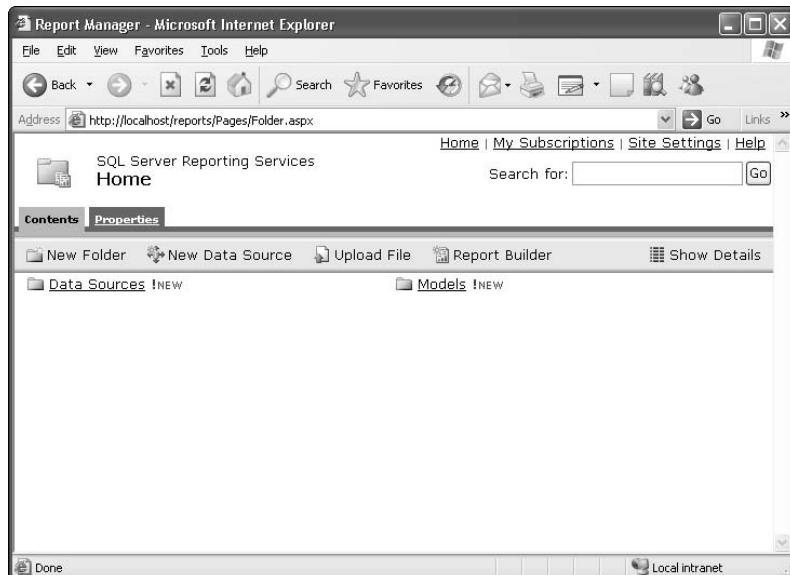


Figure 17-12

This is essentially the home page for user reports and such. Notice right away the “New!” icons next to our datasources and models (a strong hint that our deploy actually worked—amazing!). Go ahead and click on Report Builder.

Report Builder relies on a small applet that it will try to install on your system the first time you navigate to the Report Builder. You must accept the installation of the applet if you want to use Report Builder.

Report Builder will prompt you with a dialog to select the source of your report data. Go ahead and select the model you just created, and click OK to get the default report template. This is a fairly robust drag and drop design environment that will let you explore among the tables you made available in your report model and choose columns you can then drag into your report. Also take note on the far right of the report layout pane. This allows you to select between a few common layouts to start your report from.

We’re going to stick with a table report (the default as it happens) and build a relatively simple report listing all outstanding orders (where the customer has ordered something, but we haven’t shipped it yet).

Start by navigating to the `Sales.SalesOrderHeader` table, and drag `SalesOrderID` into the report. This creates a change in the entity list, as shown in Figure 17-13.

Chapter 17

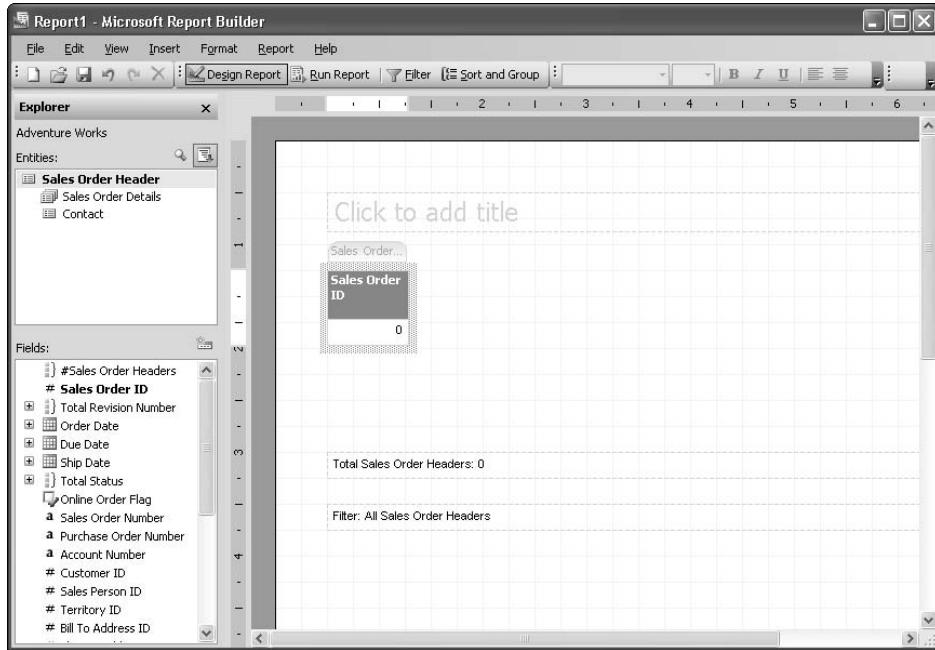


Figure 17-13

Notice how some of our tables appear to have disappeared! Fear not, my friend — our tables are still there. SQL Server has merely started showing our tables in a relative position to each other based on their relationships. So, I can click on Contact and select LastName, and everything looks fine, but what about the ProductName? Since that isn't directly related to the SalesOrderHeader table, I need to drill down to it by selecting SalesOrderDetail, SpecialOfferProduct, and then Product. From there, I can finally add the Name field, as shown in Figure 17-14:



Figure 17-14

Go ahead and fill out the remaining fields we're interested in:

- SalesOrderDetail.UnitQty
- SalesOrderHeader.OrderDate
- SalesOrderHeader.ShipDate

Also go ahead and click in the title area and add the "Orders Shipped August 2004" title. When you're done, you should wind up with something that looks like Figure 17-15.

Sales Order ID	Last Name	Product	Order Qty	Order Date	Ship Date
0	xxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	0	1/1/2006	1/1/2006

Total rows: 0

Filter: Sales Order Details with: Ship Date from 8/1/2004 to 8/31/2004

Figure 17-15

Notice also how the fields you have used are bolded in the Report Data Explorer windows.

The last thing we need to address is making sure the volume of data we get back does not go out of control. Indeed, we titled our report to be just orders shipped in August of 2004, so we had best filter our data to just that date. Report Builder makes this easy as cake to deal with. Simply click Filter in the toolbar and we can add what amounts to a WHERE clause to our report using the Filter Data dialog shown in Figure 17-16.

To create this, I started by dragging the Shipped date into the "filter with" area. I then clicked the default comparison of "equals" and changed it to the "From...To" option (which is equivalent to a BETWEEN in standard SQL). Click OK, rerun the report, and you should back around 12 pages or so (a total of just over 500 rows).

Play around with the report some, and you'll find that you can easily resort the result. Note that we also could have defined a default sort order—for example, sorting those with the soonest required date—by setting it in the Sort and Group dialog (next to Filter in the toolbar).

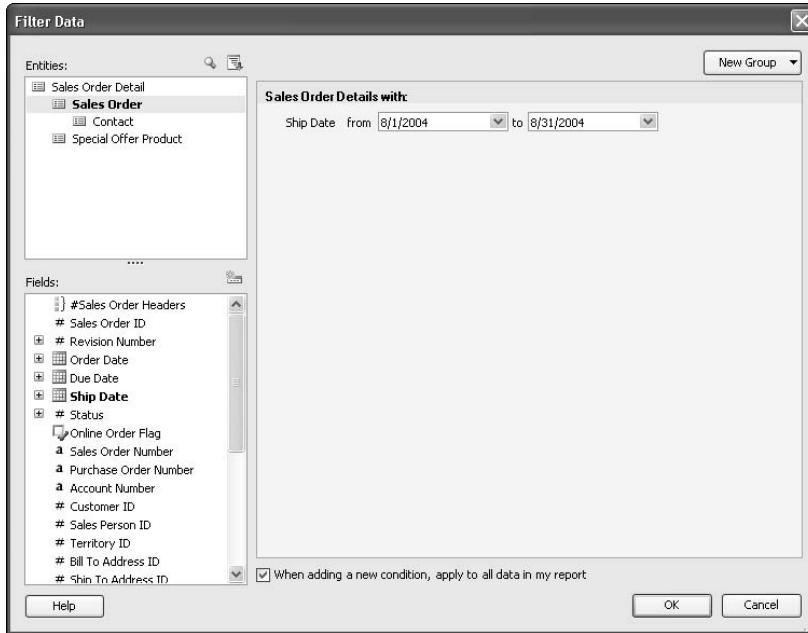


Figure 17-16

Making the Model Available

When you're done designing a report, you can click on Save in the designer, and save the report off for general use. At this point, the report is available for repeated use. Note that the format file for the report is stored on the Report Server. In Figure 17-17, I've saved our August of 2004 shipments to the report server, and it is not available on the Report Server home page:

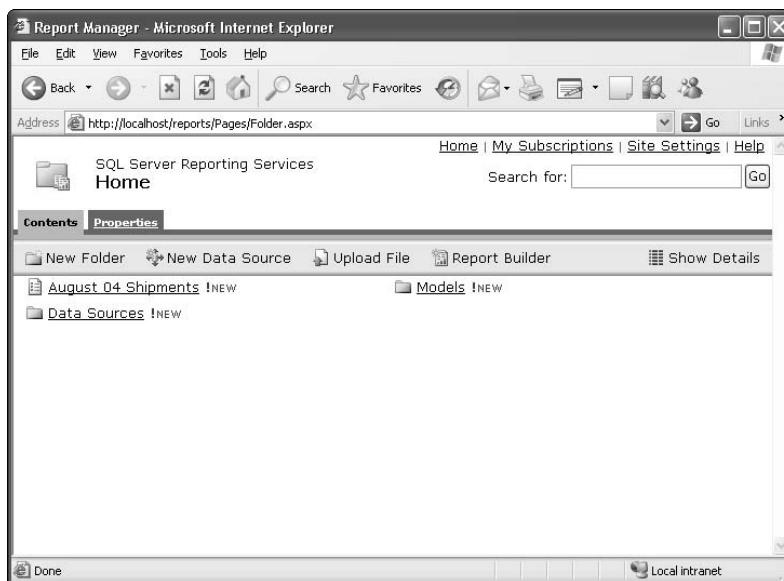


Figure 17-17

Some Notes on Programmability of Report Models

While everything we've done thus far in report models has been using tools, the production report models are actually just metadata stored on the server in what is known as Report Definition Language (RDL). RDL is essentially just XML that adheres to a schema defining the rules of RDL. The RDL defines the report, and the report services engine reads that format information and generates the report.

The significance of this is that you can manually edit the files if you so choose.

A Few Last Words on Report Models

Report models are certainly not the catch all, end all of reporting. It is, however, a very cool new feature in SQL Server in the sense that you can expose data to your end users in a somewhat controlled fashion (they don't see any more than you put in the datasource, and access to the datasource is secured such that you can control which users see which datasources) but still allow them to create specific reports of their own. Report models offer a very nice option for "quick-and-dirty" reports.

Finally, keep in mind that we generated only one very simple report for just the most simplistic of layouts. Report models also allow for basic graphing and more matrixed reporting as well.

Be very careful with who you give report design and execution rights, too. It really does not take that big of a mistake to general a query that will add a very substantial amount of load to your system. Likewise, several people all running reports at once have the potential to devastate the performance of your system. As with all things, think about the consequences of such access before granting it.

Report Server Projects

Report models can be considered to be "scratching the surface" of things—Reporting Services has much more flexibility than that (indeed, I'm very sure there are entire books around just Reporting Services—there is that much to it). In addition to what you can do with the full Visual Studio environment, Business Intelligence Development Studio will allow you to create Report Server Projects.

As I mentioned before, there are entire books around just this subject, so the approach we're going to take here is again to give you something of a little taste of the possibilities through another simple example (indeed, we're just going to do the same example using the project method).

At this point, you should be fairly comfortable with several of the concepts we're going to use here, so I'll spare you the copious screenshots and get to the nitty gritty of what needs to be done to get us up to the point of new stuff:

1. Open a new project using the Report Server Project template in Business Intelligence Development Studio
2. Create a new datasource against the AdventureWorks database (right-click the Shared Data Source folder and fill in the dialog—use the Edit button if you want the helpful dialog to build your connection string, or you can just copy the connection string from earlier in the chapter).

Chapter 17

3. Right-click the Reports folder and choose Add New Report—click past the now almost annoying “Welcome” dialog in the report wizard.
4. Select the datasource you created in Step 2, and click Next.

This should bring you to the query dialog shown in Figure 17-18.

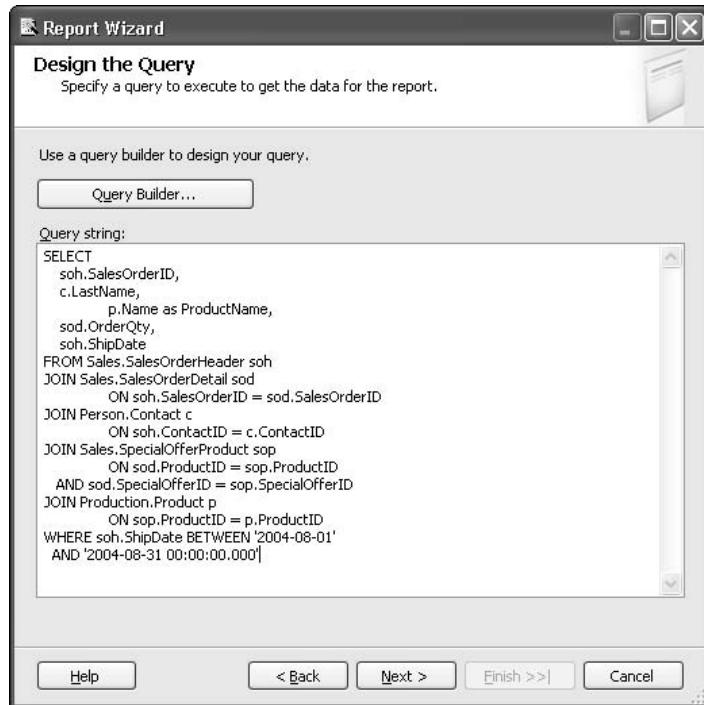


Figure 17-18

I have, of course, added the query in myself—this one should roughly duplicate the report we built in the report model section including the filter to just August 2004 orders. Note that we also could have executed a stored procedure at this point.

Click next and accept the Tabular report type, and we come to the Table Design Wizard shown in Figure 17-19.

Because the query has already trimmed things down to just the columns we need (and, as it happens, even grabbed them in proper order, but we would reorder things if we wanted to), I just selected everything and moved everything into the details table.

The Page and Group fields here would allow us to set up sort hierarchies. For example, if we wanted everything to go onto separate pages based on individual customers (say, for the sales person to understand the status of their particular customers), we could move Last Name up to the Page level. Likewise, we might instead do groupings (instead of pages) based on product name so that our people pulling the orders from inventory can grab all the product needed to fill all outstanding orders in one trip.

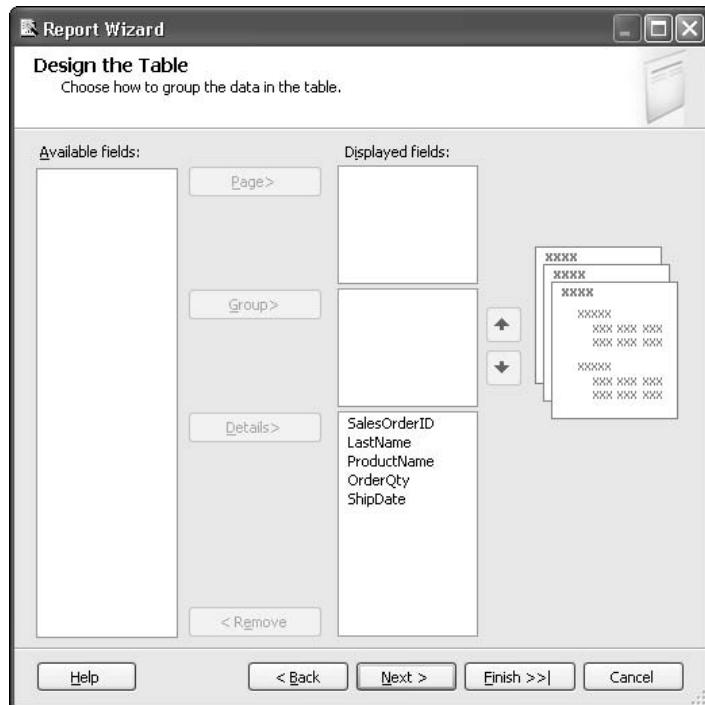


Figure 17-19

Again, click Next, and we are presented with a Table Style dialog. Choose whatever is of your liking (I'm just going to stick with Bold) and again click Next to be greeted with the summary of what your report selections were. Click finish to create the report definition, as shown in Figure 17-20 (yours may look a tad different if you chose a different style than I did).

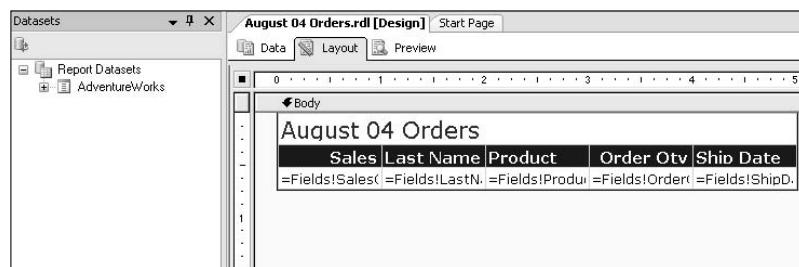


Figure 17-20

If you're familiar with other report designers, this will seem mildly familiar, as the WYSIWYG editor we're presented with here is somewhat standard fare for report designers (certainly some are more robust, but the presentation of the basics is pretty typical).

If you go ahead and preview the report, you'll find that it is largely the same as the report we generated using the Report Model notion. There are, however, some formatting areas where the defaults of the report modeler are perhaps a tad better than we have here—in particular, it makes no sense for us to report the time as part of the date if the time is always going to be midnight. With that in mind, let's make some alterations to the wizard-generated report.

We have a really great start to our report generated in seconds by the wizard. It is not, however, perfect. We want to format our dates more appropriately.

1. Right-click the ShipDate field and select Properties.
2. Click on the fx next to the Value field to indicate that we want to change the output to be that of a function result, bringing up our function dialog, as shown in Figure 17-21.

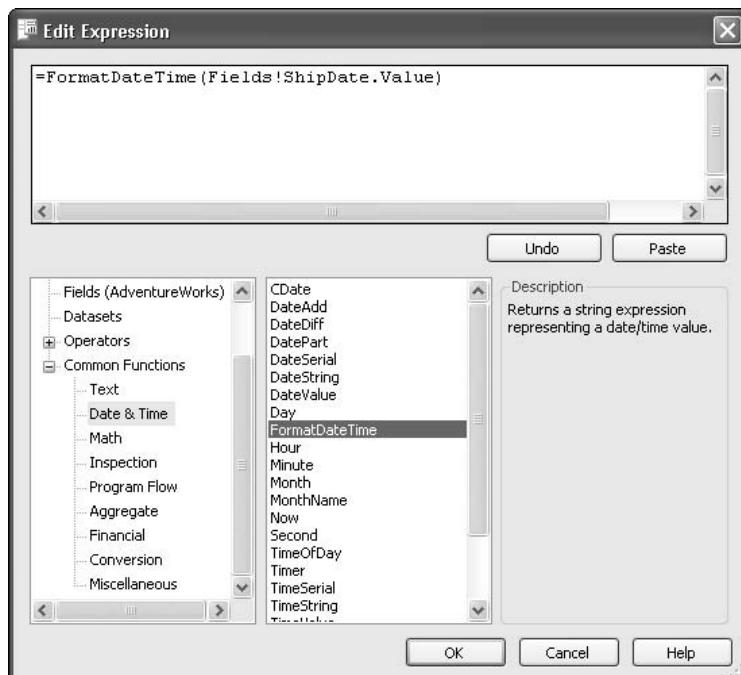
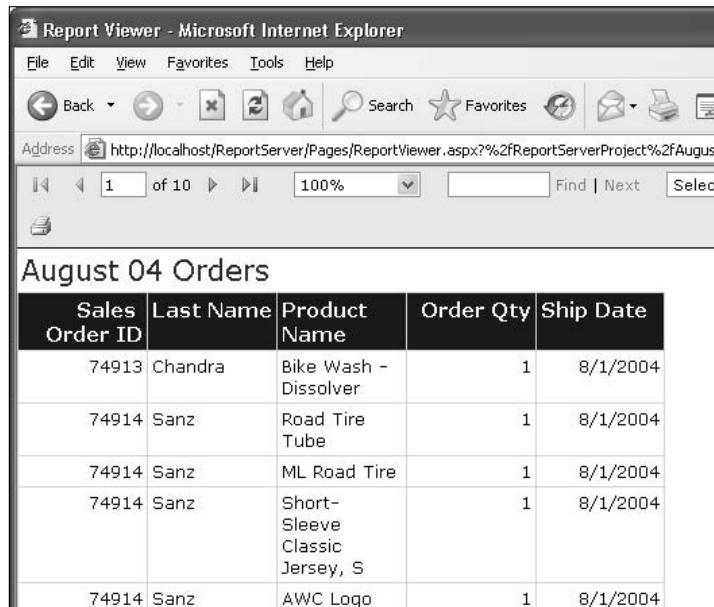


Figure 17-21

3. Add in the use of the `FormatDateTime` function. Note that I've expanded the function helper down below. You can double-click functions and it will insert the base function name into the window above. Also, it will provide context-sensitive tooltips similar to other parts of Visual Studio as you are filling in the function. Also note that this particular function does have an optional parameter that would allow us to specify a particular date representation style (say, European or Japanese), but we're going to allow the function to format it based on whatever the localized settings are on the server.
4. Click OK, and move to the Format tab. Change the "Direction" option to RTL (right to left) to cause the field to be right justified (typical for most reports).

5. Click OK, and repeat the process for RequiredDate.
6. Preview the report again, and it should look something like Figure 17-22.



The screenshot shows a Microsoft Internet Explorer window displaying a report titled "August 04 Orders". The report is presented in a table format with the following data:

Sales Order ID	Last Name	Product Name	Order Qty	Ship Date
74913	Chandra	Bike Wash - Dissolver	1	8/1/2004
74914	Sanz	Road Tire Tube	1	8/1/2004
74914	Sanz	ML Road Tire	1	8/1/2004
74914	Sanz	Short-Sleeve Classic Jersey, S	1	8/1/2004
74914	Sanz	AWC Logo	1	8/1/2004

Figure 17-22

Note that I've played around with column sizes a bit (try it!) to use the page efficiently.

Just like the report modeler, the actual report is stored in what is called Report Definition Language. As we make our changes, the RDL is changed behind the scenes so the report generator will know what to do.

To see what the RDL looks like, right-click the report in Solution Explorer and choose View Code. It's wordy stuff, but here's an excerpt from one of the fields we just edited in our report:

```
<TableCell>
  <ReportItems>
    <Textbox Name="OrderQty">
      <rd:DefaultValue>OrderQty</rd:DefaultValue>
      <ZIndex>1</ZIndex>
      <Style>
        <BorderStyle>
          <Default>Solid</Default>
        </BorderStyle>
        <PaddingLeft>2pt</PaddingLeft>
        <PaddingBottom>2pt</PaddingBottom>
        <FontFamily>Verdana</FontFamily>
        <BorderColor>
          <Default>LightGrey</Default>
        </BorderColor>
```

```
<PaddingRight>2pt</PaddingRight>
<PaddingTop>2pt</PaddingTop>
</Style>
<CanGrow>true</CanGrow>
<Value>=Fields!OrderQty.Value</Value>
</Textbox>
</ReportItems>
</TableCell>
```

Were you to become an expert, you could, if desired, edit the RPL directly.

Deploying the Report

The thing left to do is deploy the report. As with the Report Model approach, you can right-click the report in Solution Explorer and choose Deploy. There is, however, a minor catch—you need to define the target to deploy to in the project definition.

1. Right-click the Report Server Project and choose properties.
2. In the TargetServerURL field, enter the URL to your ReportServer. In my case, this may be as simple as `http://localhost/ReportServer`, but the server name could be any server you have appropriate rights to deploy to (the Virtual Directory may also be something other than `ReportServer` if you defined it that way at install).

After you've deployed, you'll want to view the report. Navigate to your report server (if on the local host and using the default directory, it would be `http://localhost/Reports`, just as it was for the Report Model examples earlier). You should immediately notice your `ReportServer` project has been added to the folder list. Mine was named `ReportProject`, and can be seen in Figure 17-23.

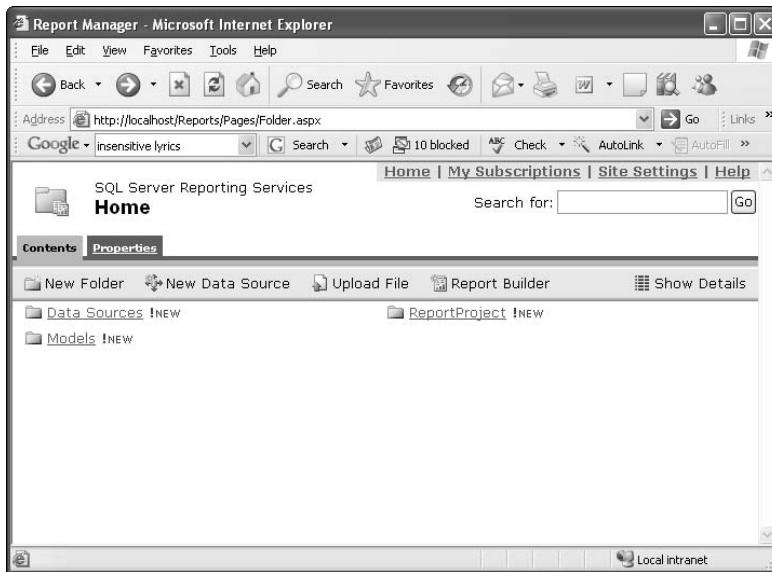


Figure 17-23

Click on your report project, and choose your outstanding orders report. It will take a bit to come up the first time you load it (if you navigate back to it again, the report definition will be cached and thus come up fairly quickly), but you should see your report just as we defined it in our project.

Summary

Reporting Services is new with SQL Server 2005. Just how big of an impact it is going to have in the average SQL Server installation and the average project is yet to be seen. There is definitely a very strong base set of functionality, and using report projects, the sky is the limit in terms of the possibilities. That said, experience tells me that IT departments will resist having a Web server deployed on their SQL Servers (report server doesn't have to be on the relational engine server, but it does add minor install complexity if it isn't). I find they also often resist installations where new functionality (in this case, new reports) can be deployed without their control. Now, lest I sound like an anti-IT bigot, they have some sound reasons to resist such "on-the-fly deployments"—a single poorly structured query or one that is new and may not have sufficient index support can drag a server to its knees in a hurry if it is abused. Your IT department is responsible for keeping that server up and responsive to everyone, so they have a reasonable basis to be skeptical of engines that allow backdoor queries.

All IT concerns aside, Reporting Services is a fine addition that provides developers a new and flexible way to empower end users without requiring a programming class to do it.

18

Buying in Bulk: the Bulk Copy Program (BCP) and Other Basic Bulk Operations

If your system is going to be operating in something of a bubble, then you can probably skip this chapter and move on. Unfortunately, the real world doesn't work that way, so you probably ought to hang around for a while.

There will be times when you need to move around large blocks of data. You need to bring in data that's in the wrong format or that's sitting in another application's data files. Sometimes, reality just gets in the way. The good thing is SQL Server has two tools to help you move data fast—the *Bulk Copy Program (BCP)* and *SQL Server Integration Services (SSIS)*. In this chapter, we'll be looking primarily at the first of these. In addition, we'll take a look at BCP's close cousins—the `BULK INSERT` command and `OPENROWSET (BULK)`. We will examine SSIS in the next chapter.

BCP is something of an old friend. You know the one—where you hardly ever see them anymore, but, when you do, you reminisce on all the crazy things you used to do together. It was, for a very long time, *the* way we moved around large blocks of data—and it did so (still does as far as that goes) amazingly fast. What, however, it lacks is sex appeal—well, frankly, since version 7.0, it has lacked appeal in a whole lot of areas.

So, why then am I even spending a chapter on it? Well, because BCP still definitely has its uses. Among its advantages are:

- It's very compact.
- It can move a lot of data very quickly.

Chapter 18

- ❑ It is legacy—that is, there may be code already running that is making effective use of it, so why change it?
- ❑ It uses a cryptic, yet very traditional scripting style (which will probably appeal to some).
- ❑ It is very consistent.

BCP is used for transferring text and SQL Server native format data to and from SQL Server tables. It has changed very little in the last several versions, and other bulk features have continued to erode the usefulness of BCP, but it still holds its own. You can think of BCP as a data pump, with little functionality other than moving data from one place to the other as efficiently as possible. The various other bulk operations we'll look at in this chapter are often easier to use, but usually come at the price of less flexibility.

In this chapter, we will look at some of the ins and outs of BCP and then use what we learn about BCP to form the foundations of many of the other features that serve a similar purpose—get data in and out of your system as quickly as possible.

BCP Utility

BCP runs from an operating system command prompt to import or export native data (specific to SQL Server), ASCII text, or Unicode text. This means that you can execute BCP from an operating system batch file or user-defined stored procedure, as well as from other places. BCP can also be run as part of a scheduled job, or executed from a .NET object through the use of a shell command.

Like most command-line utilities, options can be specified using a hyphen (-) or forward slash (/); however, unlike most DOS or Windows family utilities, option switches are case sensitive.

BCP Syntax

```
bcp {[<database name>.][<owner>.]{<table name>|<view name>} | "<query>"}
      {in | out | queryout | format} <data file>
      [-m <maximum no. of errors>] [-f <format file>] [-x] [-e <error file>]
      [-F <first row>] [-L <last row>] [-b <batch size>]
      [-n] [-c] [-w] [-N] [-V (60 | 65 | 70)] [-6]
      [-q] [-C <code page>] [-t <field term>] [-r <row term>]
      [-i <input file>] [-o <output file>] [-a <packet size>]
      [-S <server name>[\<instance name>]] [-U <login id>] [-P <password>]
      [-T] [-v] [-R] [-k] [-E] [-h "<hint> [,...n]" ]
```

Geez—that's a lot to take in, so let's go through these switches one by one (thankfully, most of them are optional, so you will usually only include just a fraction of them).

Note that many of the switches for the BCP utility are case sensitive—often, a given letter has an entirely different meaning between cases.

Parameter	Description
<i>Database name</i>	Exactly what it sounds like. Basically, this is a standard part of the four-part naming scheme. If not specified, the user's default database is assumed.
<i>owner</i>	More of the four-part naming scheme stuff. Again, exactly what it sounds like.
<i>Table or View name</i> "query"	Can only be one — table, view, or query. This is the input destination or output source table or view.
<i>in data file</i> <i>out data file</i> <i>queryout data file</i> <i>format data file</i>	A SQL Server query can be used only as a BCP output destination, and only when <i>queryout</i> is specified. If the query returns multiple result sets, only the first result set is used by BCP. Again, can only be one. If using any of these, you must also supply a source or destination file.
	Establishes the direction of the BCP action. <i>in</i> indicates that you are importing data from a source file into a table or view. <i>out</i> indicates that you are exporting data from table or view into the destination file. Use <i>queryout</i> only for output to the destination file using a query as its source. Use <i>format</i> to create a format file based on the format option you've selected. You must also specify <i>-f</i> , as well as format options (<i>-n</i> , <i>-c</i> , <i>-w</i> , <i>-6</i> , <i>-C</i> or <i>-N</i>) or answer prompts from interactive BCP.
<i>-m <maximum errors></i>	The source or destination path and filename is specified as <i><data file></i> and cannot include more than 255 characters.
<i>-f <format file></i>	You can specify a maximum number of errors that you will allow before SQL Server cancels the bulk copy operation, defaulting to 10 errors. Each row that cannot be copied by BCP is counted as one error.
<i>-x</i>	A format file contains responses saved from a previous BCP operation on the same table or view. This parameter should include the full path and filename to the format file. This option is used primarily with the <i>in</i> and <i>format</i> option to specify the path and filename when making use of or creating a format file.
<i>-e <error file></i>	Generates a XML-based format file instead of the straight text version that is default (the non-XML version is legacy support, but remains default for now). It <i>must</i> be used with both the <i>format</i> and <i>-f</i> options.
<i>-F first row</i>	You can specify the full path and filename for an error file to store any rows that BCP is not able to transfer. Otherwise, no error file is created. Any error messages will be displayed at the client station.
	Use this option if you want to specify the first row to be copied by the bulk copy operation. If not specified, BCP defaults to a value of 1 and begins copying with the first row in the source data file. This option can be handy if you want to handle your loading in chunks, and can be used to pick back up where you left off in a previous loading run.

Table continued on following page

Chapter 18

Parameter	Description
<code>-L last row</code>	This option is the complement of <code>-F</code> . It provides for determining the last row you want loaded as part of this BCP execution. If not specified, BCP defaults to a value of 0, the last row in the source file. When used in conjunction with <code>-F</code> , this option can allow you to load your data one chunk at a time—loading small blocks of data and then picking up next time where the previous load left off.
<code>-b batch size</code>	You can specify the number of rows copied as a batch. A batch is copied as a single transaction. Like all transactions, the rows of the batch are committed in an “all or nothing” fashion—either every row is committed or the transaction is rolled back and it is as if the batch never happened. The <code>-h</code> (hint) switch has a similar option (<code>ROWS_PER_BATCH</code>), which should be considered to be mutually exclusive with <code>-b</code> (use neither or one of them, but not both).
<code>-n</code>	Native data types (SQL Server data types) are used for the copy operation. Using this option avoids the need to answer the questions regarding the data types to be used in the transfer (it just picks up the native type and goes with it).
<code>-c</code>	This specifies that the operation uses character data (text) for all fields, and, as such, does not require a separate data type question for each field. A tab character is assumed as field delimiter unless you use the <code>-t</code> option and a new line character as row separator unless you specify different terminator using <code>-r</code> .
<code>-w</code>	The <code>-w</code> option is similar to <code>-c</code> but specifies Unicode data type instead of ASCII for all fields. Again, unless you override with <code>-t</code> and <code>-r</code> , the tab character and row separator are assumed to be the field delimiter and new line character respectively. This option cannot be used with SQL Server version 6.5 or earlier.
<code>-N</code>	This is basically the same as <code>-w</code> —using Unicode for character data but uses native data types (database data types) for non-character data. This option offers higher performance when going from SQL Server to SQL Server. As with <code>-w</code> , this option cannot be used with SQL Server version 6.5 or earlier.
<code>-v (60 65 70 80)</code>	Causes BCP to utilize data type formats that were available only in previous versions of SQL Server. 60 uses 6.0 data types, 65 uses 6.5 data types, 70 uses 7.0 data types, and 80 uses 2000 data types. This replaces the <code>-6</code> option.
<code>-6</code>	Use this option to force BCP to use SQL Server 6.0 or 6.5 data types. This option is used in conjunction with the <code>-c</code> or <code>-n</code> format options for backward compatibility reasons only—use <code>-v</code> whenever possible.
<code>-q</code>	Use <code>-q</code> to specify that a table or view name includes non-ANSI characters. This effectively executes a <code>SET QUOTED_IDENTIFIER ON</code> statement for the connection used by BCP. The fully qualified name, database, owner, and table or view must be enclosed in double quotation marks, in the format “ <i>database name.owner.table</i> ”.

Parameter	Description
<code>-C <code page></code>	This option is used to specify the code page for the data file data. It is only necessary to use this option with <code>char</code> , <code>varchar</code> , or <code>text</code> data having ASCII character values of less than 32 or greater than 127. A code page value of <code>ACP</code> specifies ANSI/Microsoft Windows (ISO 1252). <code>OEM</code> specifies the default client code page. If <code>RAW</code> is specified, there will be no code page conversion. You also have the option of providing a specific code page value. Avoid this option where possible—instead, use a specific collation in the format file or when asked by BCP.
<code>-t <field terminator></code>	This option allows you to override the default field terminator. The default terminator is the tab character. You can specify the terminator as <code>tab (\t)</code> , <code>new line (\n)</code> , <code>carriage return (\r)</code> , <code>backslash (\\\)</code> , <code>null terminator (\0)</code> , any printable character, or a string of up to 10 printable characters. For example, you would use <code>"-t ,"</code> for a comma-delimited text file.
<code>-r <row terminator></code>	This option works just like <code>-t</code> except that it allows you to override the default row terminator (as opposed to the field terminator). The default terminator is <code>\n</code> , the newline character. The rules are otherwise the same as <code>-t</code> .
<code>-i <input file></code>	You have the option of specifying a response file, as <code>input file</code> , containing the responses to be used when running BCP in interactive mode (this can save answering a ton of questions!).
<code>-o <output file></code>	You can redirect BCP output from the command prompt to an output file. This gives you a way to capture command output and results when executing BCP from an unattended batch or stored procedure.
<code>-a <packet size></code>	You have the option of overriding the default packet size for data transfers across the network. Larger packet sizes tend to be more efficient when you have good line quality (few CRC errors). The specified value must be between 4096 and 65535, inclusive and overrides whatever default has been set up for the server. At installation, the default packet size is 4096 bytes. This can be overridden using the SQL Server Management Studio or the <code>sp_configure</code> system stored procedure.
<code>-S <server name></code>	If running BCP from a server, the default is the local SQL Server. This option lets you specify a different server and is required in a network environment when running BCP from a remote system.
<code>-U <login name></code>	Unless connecting to SQL Server through a trusted connection, you must provide a valid username for login.
<code>-P password</code>	When you supply a username, you must also supply a password. Otherwise, you will be prompted for a password. Include <code>-P</code> as your last option with no password to specify a null password.
<code>-T</code>	You have the option of connecting to the server using network user credentials through a trusted connection. If a trusted connection is specified, there is no need to provide a <code>login name</code> or <code>password</code> for the connection.

Table continued on following page

Parameter	Description
-v	When this option is used, BCP returns version number and copyright information.
-R	Use this option to specify that the regional format for clients' local settings is used when copying currency, date, and time data. The default is that regional settings are ignored.
-k	Use this option to override the use of column default values during bulk copy, ignoring any default constraints. Empty columns will retain a null value rather than the column default.
-E	This option is used during import when the import source file contains identity column values and is essentially equivalent to <code>SET IDENTITY_INSERT ON</code> . If not specified, SQL Server will ignore the values supplied in the source file and automatically generate identity column values. You can use the format file to skip the identity column when importing data from a source that does not include identity values and have SQL Server generate the values.
-h "hint[, ...]"	The hint option lets you specify one or more hints to be used by the bulk copy operation. Option -h is not supported for SQL Server version 6.5 or earlier.
ORDER <i>column</i> [ASC DESC]	You can use this hint to improve performance when the sort order of the source data file matches the clustered index in the destination table. If the destination table does not have a clustered index or if the data is sorted in a different order the ORDER hint is ignored.
ROWS_PER_BATCH= <i>nn</i>	This can be used in place of the -b option to specify the number of rows to be transferred as a batch. Do not use this hint with the -b option.
KILOBYTES_PER_BATCH= <i>nn</i>	You can optionally specify batch size as the approximate number of kilobytes of data to be transferred in a batch.
TABLOCK	This will cause a table-level lock to be acquired for the duration of the operation. Default locking behavior is set by the table lock on bulk load table option.
CHECK_CONSTRAINTS	By default, check constraints are ignored during an import operation. This hint forces check constraints to be checked during import.
FIRE_TRIGGERS	Added back in SQL Server 2000 and similar to CHECK_CONSTRAINTS, this option causes any triggers on the destination table to fire for the transaction. By default, triggers are not fired on bulk operations. This option is not supported in versions of SQL Server prior to 2000.

BCP runs in interactive mode, prompting for format information, unless -f, -c, -n, -w, -6, or -N is specified when the command is executed. When running in interactive mode, BCP will also prompt to create a format file after receiving the format information.

BCP Import

Okay, so up to this point we've been stuck in the preliminaries. Well, it's time to get down to the business of what BCP is all about.

Probably the most common use of BCP is to import bulk data into existing SQL Server tables and views. To import data, you must have access permissions to the server, either through a login ID or a trusted connection, and you must have `INSERT` and `SELECT` permissions on the destination table or view.

The source file can contain native code, ASCII characters, Unicode, or mixed native and Unicode data. Remember to use the appropriate option to describe the source data. Also, for the data file to be usable, you must be able to describe the field and row terminators (using `-t` and `-r`) or the fields and rows must be terminated with the default tab and new line characters, respectively.

Be sure you know your destination before you start. BCP has a few quirks that can affect data import. Values supplied for timestamp or computed columns are ignored. If you have values for those columns in the source file, they'll be ignored. If the source file doesn't have values for these columns, you'll need a format file (which we'll see later in this chapter), so you can skip over them.

This is one of those really bizarre behaviors that you run across from time to time in about any piece of software you might use. In this case, if your destination table contains them, you're required to have columns to represent timestamp or computed data even though SQL Server will just ignore that data—silly, isn't it? Again, the way around this is to use a format file that explicitly says to skip the columns in question.

For BCP operations, rules are ignored. Any triggers and constraints are ignored unless the `FIRE_TRIGGERS` and/or `CHECK_CONSTRAINTS` hints are specified. Unique constraints, indexes, and primary/foreign key constraints are enforced. Default constraints are enforced unless the `-k` option is specified.

Data Import Example

The easiest way to see how BCP import works is to look at an example. Let's start with a simple example, a tab-delimited file containing shipper information for the AdventureWorks database. Here's how the data looks:

1 Smart Guys	Research and Development	2006-04-01 00:00:00.000
2 Product Test	Research and Development	2006-04-01 00:00:00.000

To import this into the `Department` table using a trusted connection at the local server, you run:

```
BCP AdventureWorks.HumanResources.Department in c:\DepartmentIn.txt -c -T
```

Two things are important here: First, up to this point, everything we've run has been done in Management Studio. For BCP however, you type your command into a command prompt box. Second, you'll need to change the preceding command line to match wherever you've downloaded the sample files/data for this book.

Because the first column in the `Department` table is an identity column and the `-E` option wasn't specified, SQL Server will ignore the identity values in the file and generate new values. The `-c` option identifies the source data as character data, and `-T` specifies to use a trusted connection.

Note that, if you have not been using Windows authentication and haven't set your network login up with appropriate rights in SQL Server, then you may need to modify the above example to utilize the `-S` and `-P` options.

Chapter 18

When we execute it, SQL Server quickly tells us some basic information about how our bulk copy operation went:

```
2 rows copied.  
Network packet size (bytes): 4096  
Clock Time (ms.) Total      : 375      Average : (3.88 rows per sec.)
```

We can go back into Management Studio and verify that the data went into the `Department` table as expected:

```
USE AdventureWorks  
  
SELECT * FROM HumanResources.Department
```

Which gets us back several rows—most importantly, the two we expect from our BCP operation:

DepartmentID	Name	GroupName	ModifiedDate
1	Engineering	Research and Development	1998-06-01...
2	Tool Design	Research and Development	1998-06-01...
...			
...			
16	Executive	Executive General and Administration	1998-06-01...
17	Smart Guys	Research and Development	2006-04-01...
18	Product Test	Research and Development	2006-04-01...

As always, note that, other than the two rows we just imported, your data may look a bit different depending on what parts of this book you've ran the examples on, which you haven't, and how much playing around of your own you've done. For this example, you just want to see that Smart Guys and Product Test made it into the table with the appropriate information—the identity values will have been reassigned to whatever was next for your particular server.

Now let's look at a more involved example. Let's say we have a table called `CustomerList`. A CREATE statement to make our `CustomerList` table looks like this:

```
CREATE TABLE dbo.CustomerList  
(  
    CustomerID      nchar(5)      NOT NULL  
        PRIMARY KEY,  
    CompanyName     nvarchar(40)   NOT NULL,  
    ContactName     nvarchar(30)   NULL,  
    ContactTitle    nvarchar(30)   NULL,  
    Address         nvarchar(60)   NULL,  
    City            nvarchar(15)   NULL,  
    Region          nvarchar(15)   NULL,  
    PostalCode      nvarchar(10)   NULL,  
    Country         nvarchar(15)   NULL,  
    Phone           nvarchar(24)   NULL,  
    Fax             nvarchar(24)   NULL  
)
```

We have a comma-delimited file (in the same format as a .csv file) with new customer information. This time, the file looks like:

```
XWALL,Wally's World,Wally Smith,Owner,,,,,(503)555-8448,,  
XGENE,Generic Sales and Services,Al Smith,,,,,(503)555-9339,,  
XMORE,More for You,Paul Johnston,President,,,,,(573)555-3227,,
```

What's with all the commas in the source file? Those are placeholders for columns in the CustomerList table. The source file doesn't provide values for all of the columns, so commas are used to skip over those columns. This isn't the only way to handle a source file that doesn't provide values for all of the columns. You can use a format file to map the source data to the destination. We'll be covering format files a little later in the chapter.

Imagine for a moment that are going to run BCP to import the data to a remote system. The command is:

```
BCP AdventureWorks.dbo.CustomerList in c:\newcust.txt -c -t, -r\n -Ssocrates -Usa -Pbubbagump
```

The line wrapping shown here was added to make the command string easier to read. Do not press Enter to wrap if you try this example yourself. Type the command as a single string and allow it to wrap itself inside the command prompt.

Once again, the data is being identified as character data. The -t, option identifies the file as comma-delimited (terminated) data, and -r\n identifies the new line character as the row delimiter. Server connection information was also provided for a little variety this time, using sa as your login and bubbagump as the password.

Again, BCP confirms the transfer along with basic statistics:

```
Starting copy...  
  
3 rows copied.  
Network packet size (bytes): 4096  
Clock Time (ms.) Total : 15 Average : (200.00 rows per sec.)
```

And again we'll also go verify that the data got there as expected:

```
USE AdventureWorks  
  
SELECT CustomerID, CompanyName, ContactName  
FROM dbo.CustomerList  
WHERE CustomerID LIKE 'X%'
```

And, sure enough, all our data is there...

CustomerID	CompanyName	ContactName
XWALL	Wally's World	Wally Smith
XMORE	More for You	Paul Johnston
XGENE	Generic Sales and Services	Al Smith

Logged vs. Nonlogged

BCP can run in either fast mode (not logged) or slow mode (logged operation). Each has its advantages. Fast mode gives you the best performance, but slow mode provides maximum recoverability. Since slow mode is logged, you can run a quick transaction log backup immediately after the import and be able to recover the database should there be a failure.

Fast mode is usually your best option when you need to transfer large amounts of data. Not only does the transfer run faster but since the operation isn't logged you don't have to worry about running out of space in the transaction log. What's the catch? There are several conditions that must be met for BCP to run as nonlogged:

- The target table cannot be replicated.
- If the target table is indexed, it must not currently have any rows.
- If the target table already has rows, it must not have any indexes.
- The TABLOCK hint is specified.
- The target table must have no triggers.
- For versions prior to SQL Server 2000, the `select into/bulkcopy` option must be set to true.

Obviously, if you want to do a fast mode copy into an indexed table with data, you will need to:

- Drop the indexes
- Drop any triggers
- Run BCP
- Reindex the target table
- Re-create any triggers

You need to immediately back up the destination database after a nonlogged BCP operation.

If the target table doesn't meet the requirements for fast BCP, then the operation will be logged. This means that you can run the risk of filling the transaction log when transferring large amounts of data. You can run `BACKUP LOG` using the `WITH TRUNCATE_ONLY` option to clear the transaction log. The `TRUNCATE_ONLY` option truncates the inactive portion of the log without backing up any data.

I can't stress enough how deadly BCP operations can be to the size of your log. If you can't achieve a minimally logged operation, then consider adjusting your batch size down and turning `TRUNCATE ON CHECKPOINT` on for the duration of the operation. Another solution is to use the `-F` and `-L` options to pull things in a block at a time and truncate the log in between each block of data.

BCP Export

If you're going to be accepting data in via bulk operations, then it follows that you probably want to be able to pump data out, too.

BCP allows you to export data from a table, view, or query. You must specify a destination filename—if the file already exists, it will be overwritten. Unlike import operations, you are not allowed to skip columns during export. Timestamp, rowguid, and computed columns are exported in the same manner (just like they were “real” data) as any other SQL Server columns. To run an export, you must have appropriate SELECT authority to the source table or tables.

Look at a couple of quick examples using the HumanResources.Department table in the AdventureWorks database.

To export to a data file using the default format, you could run:

```
BCP AdventureWorks.HumanResources.Department out c:\DepartmentIn.txt -c -T
```

This would create a file that looks like:

```
1   Engineering      Research and Development    1998-06-01 00:00:00.000
2   Tool Design      Research and Development    1998-06-01 00:00:00.000
...
...
17  Smart Guys       Research and Development    2006-04-01 00:00:00.000
18  Product Test     Research and Development    2006-04-01 00:00:00.000
```

In this case, we didn’t have to use a format file, nor were we prompted for any field lengths or similar information—the use of the -c option indicated that we just wanted everything—regardless of type—exported as basic ASCII text in a default format. The default calls for tabs as field separators and the new line character to separate rows.

Keep in mind that the destination file will be overwritten if it already exists—this will happen without any kind of prompt or warning.

To modify the separator to something custom, we could run something like:

```
BCP AdventureWorks.HumanResources.Department out DepartmentOut.txt -c -T -t,
```

Notice the comma at the end—that is not a typo. The next character after the t is the field separator—in this case, a comma.

This would give us:

```
1,Engineering,Research and Development,1998-06-01 00:00:00.000
2,Tool Design,Research and Development,1998-06-01 00:00:00.000
...
...
17,Smart Guys,Research and Development,2006-04-01 00:00:00.000
18,Product Test,Research and Development,2006-04-01 00:00:00.000
```

We used a comma separator instead of a tab, and got what amounts to a .csv file.

Format Files

The use of format files was mentioned in the previous section. This is actually something of an exciting area, as a lot has happened in to our options for format files in SQL Server 2005.

Format files can be thought of as import templates and make it easier to support recurring import operations when:

- ❑ Source file and target table structures or collations do not match.
- ❑ You want to skip columns in the target table.
- ❑ Your file contains data that makes the default data typing and collation difficult or unworkable.

Beginning in this release, format files now come in two varieties: non-XML and XML. We will start off by looking at the “old” way of doing things (the non-XML version) and then take a look at the newer XML format files after we’ve checked how each non-XML version looks.

To get a better idea of how each type of format file works, let’s look at some specific examples. First you’ll see how the file is structured when the source and destination match. Next, you can compare this to situations where the number of source file fields doesn’t match the number of table columns or where source fields are ordered differently than the table columns.

You can create a default format file (which is non-XML for backward-compatibility reasons) to use as your source when you run BCP in interactive mode. After prompting for column value information, you’re given the option of saving the file. The default filename is BCP.fmt, but you can give the format file any valid filename.

To create a default format like this for the AdventureWorks database HumanResources.Department table, you could run:

```
BCP AdventureWorks.HumanResources.Department out c:\department.txt -T
```

This is a handy way of creating a quick format file that you can then edit as needed. You can do this with any table, so you can use BCP to get a jump-start on your format file needs.

Accept the default prefix and data length information for each file, and, in this case, a comma as the field terminator. SQL Server will prompt you to save the format file after you’ve entered all of the format information, in my case I’m going to save it off as Department.fmt. You can then edit the format file to meet your particular needs with any text editor, such as Windows Notepad.

Let’s take a look at the format file we just produced:

```
9.0
4
1      SQLSMALLINT    0      2      " , "      1      DepartmentID      "
2      SQLNCHAR        2      100     " , "      2      Name
SQL_Latin1_General_CI_AS
3      SQLNCHAR        2      100     " , "      3      GroupName
SQL_Latin1_General_CI_AS
4      SQLDATETIME     0      8      " , "      4      ModifiedDate      "
```

The first two lines in the file identify the BCP version number and the number of fields in the host file. The remaining lines describe the host data file and how the fields match up with target columns and collations.

The first column is the host file field number. Numbering starts with 1 through the total number of fields. Next is the host file data type. The example file has a mix of a few data types. All text is in Unicode format, so the data type of all fields is SQLNCHAR—given that there are no special characters in this date, we could have just as easily gone with a SQLCHAR (ASCII) format.

The next two columns describe the prefix and data length for the data fields. The prefix is the number of prefix characters in the field. The prefix describes the length of the data in the actual BCP file and allows the data file to be compacted to a smaller size. The data field is the maximum length of the data stored in the field. Next is the field terminator (delimiter). In this case, a comma is used as the field terminator and new line as the row terminator. The next two columns describe the target table columns by providing the server column order and server column name. Since there is a direct match between the server columns and host fields in this example, the column and field numbers are the same, but it didn't necessarily have to work that way. Last, but not least, comes the collation for each column (remember that, with SQL Server 2000 and newer, we can have a different collation for every column in a table).

Now, let's check the XML version. To create this, we run almost the same command, but add the -x switch:

```
BCP AdventureWorks.HumanResources.Department out c:\department.txt -T -x
```

The file we wind up with looks radically different:

```
<?xml version="1.0"?>
<BCPFORMAT xmlns="http://schemas.microsoft.com/sqlserver/2004/bulkload/format"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<RECORD>
  <FIELD ID="1" xsi:type="NativeFixed" LENGTH="2"/>
  <FIELD ID="2" xsi:type="NCharPrefix" PREFIX_LENGTH="2" MAX_LENGTH="100"
COLLATION="SQL_Latin1_General_CI_AS"/>
  <FIELD ID="3" xsi:type="NCharPrefix" PREFIX_LENGTH="2" MAX_LENGTH="100"
COLLATION="SQL_Latin1_General_CI_AS"/>
  <FIELD ID="4" xsi:type="NativeFixed" LENGTH="8"/>
</RECORD>
<ROW>
  <COLUMN SOURCE="1" NAME="DepartmentID" xsi:type="SQLSMALLINT"/>
  <COLUMN SOURCE="2" NAME="Name" xsi:type="SQLNVARCHAR"/>
  <COLUMN SOURCE="3" NAME="GroupName" xsi:type="SQLNVARCHAR"/>
  <COLUMN SOURCE="4" NAME="ModifiedDate" xsi:type="SQLDATETIME"/>
</ROW>
</BCPFORMAT>
```

Notice that everything is explicitly called out. What's more, there is a XML schema document associated with XML format files, which means you can validate the XML in your XML editor of choice.

I'm not going to pick any bones about this—I LOVE the new XML-formatted version. If you don't need to worry about backwards compatibility, this one seems a no brainer to me.

The old format files work, but, every time I work with them extensively, I consider purchasing stock in a pain reliever company—they are that much of a headache if you have to do anything beyond the defaults. Everything about them has to be "just so," and in larger tables, it's easy to miss a typo since fields are not clearly separated. XML tagging fixes all that and makes clear what every little entry is there for—debugging is much, much easier.

When Your Columns Don't Match

If only the world was perfect and the data files we received always looked just like our tables.

Okay, time to come out of dreamland. I'm reasonably happy with the world I live in, but it's hardly a perfect place and the kinds of data files I need to do bulk operations on rarely look like their destination. So, what then are we to do when the source file and destination table do not match up the way we want? Or what about going the other way—from a table to an expected data file format that isn't quite the same?

Fortunately, format files allow us to deal with several different kinds of variations we may have between source and destination data. Let's take a look.

Files with Fewer Columns Than the Table

Let's start with the situation where the data file has fewer fields than the destination table. We need to modify the format file we've already been using to identify which columns do not exist in the data file and, accordingly, which columns in our table should be ignored. This is done by setting the prefix and data length to 0 for each missing field and the table column number to 0 for each column we are going to skip.

For example, if, as one might expect, the data file has only `DepartmentID`, `Name`, and `GroupName`, you would modify the file to:

9.0							
4							
1	SQLSMALLINT	0	2	" , "	1	DepartmentID	" "
2	SQLNCHAR	2	100	" , "	2	Name	
	SQL_Latin1_General_CI_AS						
3	SQLNCHAR	2	100	" , "	3	GroupName	
	SQL_Latin1_General_CI_AS						
4	SQLDATETIME	0	0	" , "	0	ModifiedDate	" "

As you can see, the `ModifiedDate` field and column has been zeroed out. Because `ModifiedDate` is not supplied and the column has a default value (`Getdate()`), that default value will be used for our inserted rows.

The XML version doesn't look all that different:

```
<?xml version="1.0"?>
<BCPFORMAT xmlns="http://schemas.microsoft.com/sqlserver/2004/bulkload/format"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <RECORD>
    <FIELD ID="1" xsi:type="NativeFixed" LENGTH="2" />
    <FIELD ID="2" xsi:type="NCharPrefix" PREFIX_LENGTH="2" MAX_LENGTH="100"
      COLLATION="SQL_Latin1_General_CI_AS"/>
    <FIELD ID="3" xsi:type="NCharPrefix" PREFIX_LENGTH="2" MAX_LENGTH="100"
      COLLATION="SQL_Latin1_General_CI_AS"/>
  </RECORD>
  <ROW>
    <COLUMN SOURCE="1" NAME="DepartmentID" xsi:type="SQLSMALLINT"/>
    <COLUMN SOURCE="2" NAME="Name" xsi:type="SQLNCHAR" />
    <COLUMN SOURCE="3" NAME="GroupName" xsi:type="SQLNCHAR" />
  </ROW>
</BCPFORMAT>
```

There was no column in the file to define, so we didn't. We aren't sticking anything in the `ModifiedDate` column, so we skipped that, too (counting on the default in its case).

More Columns in the File Than in the Table

The scenario for a data file that has more columns than the table does is actually amazingly similar to the short data file scenario we just looked at. The only trick here is that you must add column information for the additional fields, but the prefix length, data length, and column number fields are all set to 0:

9.0							
4							
1	SQLSMALLINT	0	2	","	1	DepartmentID	" "
2	SQLNCHAR	2	100	","	2	Name	
	SQL_Latin1_General_CI_AS						
3	SQLNCHAR	2	100	","	3	GroupName	
	SQL_Latin1_General_CI_AS						
4	SQLDATETIME	0	8	","	4	ModifiedDate	" "
5	SQLDATETIME	0	0	","	0	ModifiedDate	" "

This time, the host file includes fields for a date the department was created on. The target table doesn't have a column to receive this information. The fields are added to the original format file, as well as two dummy columns with a column number of 0. This will force BCP to ignore the fields.

For this one, the XML version does have to deal with the fact that the file has a column that needs to be addressed. The destination, however, we can continue to ignore:

```
<?xml version="1.0"?>
<BCPFORMAT xmlns="http://schemas.microsoft.com/sqlserver/2004/bulkload/format"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <RECORD>
        <FIELD ID="1" xsi:type="NativeFixed" LENGTH="2"/>
        <FIELD ID="2" xsi:type="NCharPrefix" PREFIX_LENGTH="2" MAX_LENGTH="100"
COLLATION="SQL_Latin1_General_CI_AS"/>
        <FIELD ID="3" xsi:type="NCharPrefix" PREFIX_LENGTH="2" MAX_LENGTH="100"
COLLATION="SQL_Latin1_General_CI_AS"/>
        <FIELD ID="4" xsi:type="NativeFixed" LENGTH="8"/>
        <FIELD ID="5" xsi:type="NativeFixed" LENGTH="8"/>

    </RECORD>
    <ROW>
        <COLUMN SOURCE="1" NAME="DepartmentID" xsi:type="SQLSMALLINT"/>
        <COLUMN SOURCE="2" NAME="Name" xsi:type="SQLNVARCHAR"/>
        <COLUMN SOURCE="3" NAME="GroupName" xsi:type="SQLNVARCHAR"/>
        <COLUMN SOURCE="4" NAME="ModifiedDate" xsi:type="SQLDATETIME"/>
    </ROW>
</BCPFORMAT>
```

Mismatched Field Order

Another possibility is that the host and target have the same fields, but the field orders don't match. This is corrected by changing the server column order to match the host file order:

Chapter 18

```
9.0
4
1      SQLSMALLINT    0      2      " , "    1      DepartmentID          " "
2      SQLNCHAR       2      100     " , "    3      GroupName
SQL_Latin1_General_CI_AS
3      SQLNCHAR       2      100     " , "    2      Name
SQL_Latin1_General_CI_AS
4      SQLDATETIME    0      8      " , "    4      ModifiedDate         " "
```

In this case, the group name is listed before the department name in the source file. The server column order has been changed to reflect this. Notice, the order in which the server columns are listed has not changed, but the server column numbers have been swapped.

So, to translate this to XML, we just need to change a field or two versus our original XML file:

```
<?xml version="1.0"?>
<BCPFORMAT xmlns="http://schemas.microsoft.com/sqlserver/2004/bulkload/format"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <RECORD>
    <FIELD ID="1" xsi:type="NativeFixed" LENGTH="2" />
    <FIELD ID="2" xsi:type="NCharPrefix" PREFIX_LENGTH="2" MAX_LENGTH="100"
COLLATION="SQL_Latin1_General_CI_AS"/>
    <FIELD ID="3" xsi:type="NCharPrefix" PREFIX_LENGTH="2" MAX_LENGTH="100"
COLLATION="SQL_Latin1_General_CI_AS"/>
    <FIELD ID="4" xsi:type="NativeFixed" LENGTH="8" />
  </RECORD>
  <ROW>
    <COLUMN SOURCE="1" NAME="DepartmentID" xsi:type="SQLSMALLINT" />
    <COLUMN SOURCE="3" NAME="Name" xsi:type="SQLNVARCHAR" />
    <COLUMN SOURCE="2" NAME="GroupName" xsi:type="SQLNVARCHAR" />
    <COLUMN SOURCE="4" NAME="ModifiedDate" xsi:type="SQLDATETIME" />
  </ROW>
</BCPFORMAT>
```

Using Format Files

As an example, let's use a format file for an import. This command will copy records into the Department table based on a file named shortdept.txt. We'll use ShortDept.fmt as our non-XML format file example, and ShortDeptX.xml as our XML-based format file.

```
BCP AdventureWorks.HumanResources.Department in c:\shortdept.txt -
fc:\shortdept.fmt -Usa -Pbubblegum
```

Just for a change of flavor, the preceding example command line uses SQL Server authentication instead of Windows authentication. If you prefer Windows authentication, just replace the -U and -P parameters above with the -T we've used frequently.

The sample files used in this example, ShortDept.txt, ShortDept.fmt, and ShortDeptX.xml, are available for download from the Wrox Web site or from ProfessionalSQL.com.

Maximizing Import Performance

One obvious way of maximizing BCP performance is to make sure that the target table meets all the requirements for running BCP as a nonlogged operation. This may mean you need to:

- Drop any existing indexes on the target table. While this is actually required only if you want a minimally logged operation, the fact is that leaving indexes off during bulk operation is greatly beneficial performance wise regardless of the logging status. Be sure, however, to rebuild your indexes after the bulk operation is complete.
- Attempt to have your source data files created in the same order that your clustered index (if there is one) is in. During your index rebuild, this will allow you to make use of the `SORTED_DATA_REORG` option, which greatly speeds index creation (and thus the overall time of your BCP operation). Even if you have to leave a clustered index in place, performing the BCP with sorted data will allow the use of the `ORDER` column option (within the `-HINT` option).
- Make sure your maintenance properties are set to simple or nonlogged. If they are set to Full Recovery, then BCP will not be allowed a minimally logged operation.

If you're looking for additional improvement when importing data into a table, you can run *parallel data loads* from multiple clients. To do this, you must:

- Use the `TABLOCK` hint.
- Remove all indexes (you can rebuild them after the operation is complete).
- Set the server recovery option to Bulk-Logged.

How would this work? Rather than importing one very large file, break it up into smaller files. Then you launch BCP from multiple client systems, each client importing one of the smaller files. Obviously, you will be interested in doing this only if the expected performance increase saves more time on the import than you'll spend preparing the source files and copying them to the clients.

Parallel loads were not supported for SQL Server 6.5 or earlier.

With either of these operations, it will be necessary to recreate any indexes on the target table after completing the operation. Recreate the target table clustered index (if any) before any non-clustered indexes.

You can get additional performance improvement by letting SQL Server ignore check constraints and triggers, the default option. Keep in mind that this can result in loading data that violates the table's check constraints and any data integrity rules that are enforced by your triggers.

BULK INSERT

One of the “cousins” that I mentioned at the beginning of the chapter was the `BULK INSERT` command. In order to make use of this command, you must be a member of either the `sysadmin` or `bulkadminserver` role.

Chapter 18

What `BULK INSERT` does is essentially operate like a limited version of BCP that is available directly within T-SQL. The syntax looks like this:

```
BULK INSERT [['<database name>'].] ['<schema name>'].]<table name>' FROM '<data file>'  
[WITH  
(  
    [BATCHSIZE [= <batch size>]]  
    [, CHECK_CONSTRAINTS]  
    [, CODEPAGE [= {'ACP' | 'OEM' | 'RAW' | '<code page>'}]]  
    [, DATAFILETYPE [= {'char' | 'native' | 'widechar' | 'widenative'}]]  
    [, FIELDTERMINATOR [= '<field terminator>']]  
    [, FIRSTROW [= <first row>]]  
    [, FIRE_TRIGGERS]  
    [, FORMATFILE = '<format file path>']  
    [, KEEPIDENTITY]  
    [, KEEPNULLS]  
    [, KILOBYTES_PER_BATCH [= <no. of kilobytes>]]  
    [, LASTROW [= <last row no.>]]  
    [, MAXERRORS [= <max errors>]]  
    [, ORDER ({column [ASC|DESC]} [ ,...n ] )]  
    [, ROWS_PER_BATCH [= <rows per batch>]]  
    [, ROWTERMINATOR [= '<row terminator>']]  
    [, TABLOCK]  
    [, ERRORFILE = '<file name>']  
)  
)
```

Now, if you are getting a sense of *déjà vu*, then you're on top of things for sure — these switches pretty much all have equivalents in the basic BCP import syntax that we started off the chapter with.

The special permission requirements of `BULK INSERT` are something of a hassle (not everyone belongs to `sysadmin` or `bulkinsert`), but `BULK INSERT` does carry with it a couple of distinct advantages:

- ❑ It can be enlisted as part of a user-defined transaction using `BEGIN TRAN` and its associated statements.
- ❑ It runs in-process to SQL Server, so it should pick up some performance benefits there as it avoids marshalling.
- ❑ It's slightly (very slightly) less cryptic than the command-line syntax used by BCP.

The big issue with `BULK INSERT` is just that — it's bulk *insert*. `BULK INSERT` will not help you build format files. It will not export data for you. It's just a simple and performative way to get BCP functionality for moving data into your database from within SQL Server.

OPENROWSET (BULK)

Yet another cousin to BCP, but this one is a far more distant one. You can think of this cousin as being from the side of the family that got most of the money and power (in case you can't tell, I like this one!). `OPENROWSET (BULK)` marries the bulk rowset provider with the `OPENROWSET`'s ability to be used within queries for fast and relatively flexible access to external files without necessarily needing to load them into an intermediate table.

One of the more common uses for BCP is to load external data files for use by some periodic process. For example, you may receive files that contain things like credit reports, vendor catalogs, and other data that is placed in a generic format by a vendor. This is vital information to you, but you're more interested in a one-time interaction with the data than in truly importing it. OPENROWSET (BULK) allows the possibility of treating that file—or just one portion of that file—as a table. What's more, it can utilize a format file to provide a better translation of the file layout than a simple linked table might provide. The syntax looks like this:

```
OPENROWSET
( BULK '<data file>' ,
  { [ FORMATFILE = '<format file>' ] [
    [, CODEPAGE [= {'ACP' | 'OEM' | 'RAW' | '<code page>'}]]
    [, FIRSTROW [= <first row>]]
    [, LASTROW [= <last row no.>]]
    [, MAXERRORS [= <max errors>]]
    [, ROWS_PER_BATCH [= <rows per batch>]]
    [, ERRORFILE = '<file name>']
  ]
  | SINGLE_BLOB | SINGLE_CLOB | SINGLE_NCLOB
) )
```

Keep in mind that OPENROWSET is more of a bulk access method than an insert method. You can most certainly do an `INSERT INTO` where the source of your data is an OPENROWSET (indeed, that's often how it's used), but OPENROWSET has more flexibility than that. Now, with that in mind, let's look at a couple of important bulk option issues when dealing with OPENROWSET.

ROWS_PER_BATCH

This is misleading. The big thing to remember on this one is that, if you use it, you are essentially providing a hint to the query optimizer. SQL Server will always process the entire file, but whatever you put in this value is going to be a hint to the optimizer about how many rows are in your file—try to make it accurate or leave it alone.

SINGLE_BLOB, SINGLE_CLOB, SINGLE_NCLOB

These say to treat the entire file as one thing—one row with just one column. The type will come through as `varbinary(max)`. Windows encoding conventions will be applied if you use `SINGLE_BLOB`. `SINGLE_CLOB` will assume that your data is ASCII, and `SINGLE_NCLOB` will assume it is Unicode.

Summary

In this chapter, we looked at the first of our two major data import/export utilities. BCP is used primarily for importing and exporting data stored as text files to and from our SQL Server. We also took a look at some of BCP's brethren.

As a legacy utility, BCP will be familiar to most people who have worked with SQL Server for any length of time. Microsoft continues to enhance the core technology behind BCP, so I think it's safe to say that BCP is here to stay.

Chapter 18

That said, BCP is quite often not your best option. Be sure to check your options with `BULK INSERT` (and the benefits of running in process to SQL Server) as well as `OPENROWSET (BULK)`.

In our next chapter, we will take a look at BCP's major competition—SQL Server Integration Services (SSIS). SSIS has the glamour and glitz that BCP is missing, but it also has its own quirks that can occasionally make the simplicity of BCP seem downright appealing.

19

Getting Integrated

Out with the old, in with the new — that's going to be what this chapter is about. For context, we need to touch base on what the old was — Data Transformation Services, or DTS. That's what we had before we had Integration Services — the topic of this chapter.

I mention DTS for two reasons. First, it was revolutionary. Never before was a significant tool for moving and transforming large blocks of data included in one of the major Relational Database Management Systems (RDBMSs). All sorts of things that were either very difficult or required very expensive third-party tools were suddenly a relative piece of cake. Second, I mention it because it's gone — or, more accurately, replaced.

DTS was completely rewritten for this release and, as part of that, also got a new name: Integration Services.

If you're running a mixed environment with SQL Server 2000 or migrating from that version, do not fear. SQL Server 2005 Integration Services (SSIS) will run old DTS packages with the installation of Legacy Services in the Installation Wizard when you install SQL Server 2005.

Use the SSIS Package Migration Wizard to help upgrade old DTS packages.

In this chapter, we'll be looking at how to perform basic import and export of data, and we'll briefly discuss some of the other things possible with tools like Integration Services.

Understanding the Problem

The problems being addressed by Integration Services exist in at least some form in a large percentage of systems — how to get data into or out of our system from or to foreign data sources. It can be things like importing data from the old system into the new, or a list of available items from

a vendor—or who knows what. The common thread in all of it, however, is that we need to get data that doesn't necessarily fit our tables into them anyway.

What we need, is a tool that will let us *Extract, Transform, and Load* data into our database—a tool that does this is usually referred to simply as an “ETL” tool. Just how complex of a problem this kind of tool can handle varies, but SQL Server Integration Services—or SSIS—can handle nearly every kind of situation you may have.

This may bring about the question “Well, why doesn’t everybody use it then, since it’s built in?” The answer is one of how intuitive it is in a cross-platform environment. There are third-party packages out there that are much more seamless and have fancier UI environments. These are really meant to allow unsophisticated users move data around relatively easily—they are also outrageously expensive. Under the old DTS product, I actually had customers that were Oracle or other DBMS oriented, but purchased a full license for SQL Server just to make use of DTS—I’m sure that SSIS will be much the same (it’s very, very nice!).

An Overview of Packages

Just as DTS did in SQL Server 7.0 and 2000, SSIS utilizes the notion of a “package” to contain a set of things to do. Each individual action is referred to as a “task.” You can bundle up a series of tasks and even provide control of flow choices to conditionally run different tasks in an order of your choosing (for example, if one task were to fail, then run a different task). Packages can be created programmatically (using a rather robust object model), but most initial package design is done in a designer that is provided in SQL Server.

Let’s go ahead and create a simple package just to get a feel for the environment. To get to SSIS, you need to start the SQL Server Business Intelligence Development Studio from the Programs→Microsoft SQL Server 2005 menu on your system—then select Integration Services as your project type as shown in Figure 19-1.

To be honest, I consider the move of this to the Intelligence Studio to be nothing short of silly, but I do have to admit that one of the most common uses for a tool such as SSIS is extracting data from a Online Transaction Processing (OLTP) database and transforming it for using in a Online Analytical Processing (OLAP) database. Still, it’s a tool used for all sorts of different things and probably should have been part of the core Management Studio (or at least available in both places).

So, to reiterate, the SSIS tool is in Business Intelligence Studio (much like the Reporting Services-related items)—not in Management Studio as most items we’ve looked at have been.

The exact look of the dialog in Figure 19-1 will vary depending on whether you also have Visual Studio installed and, if so, what parts of Visual Studio you included in your installation.

In this case, I’ve named my project an ever so descriptive “SSISProject”—from there, I simply click OK and SQL Server creates the project and brings up the default project window, shown in Figure 19-2, for SSIS-related projects.

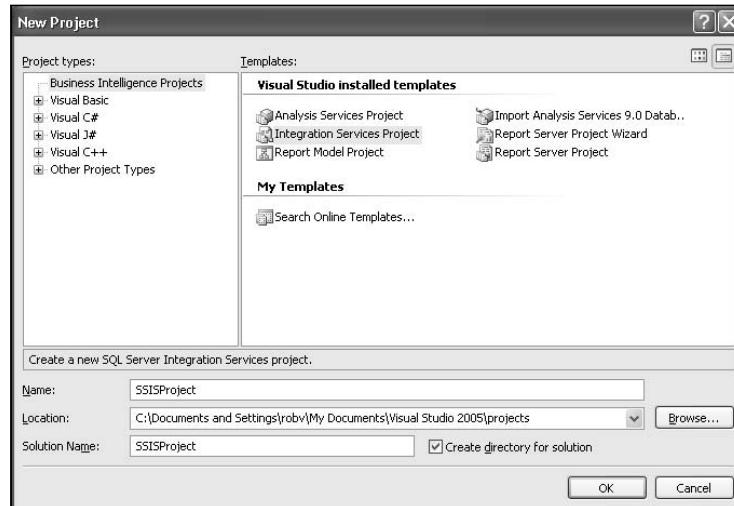


Figure 19-1

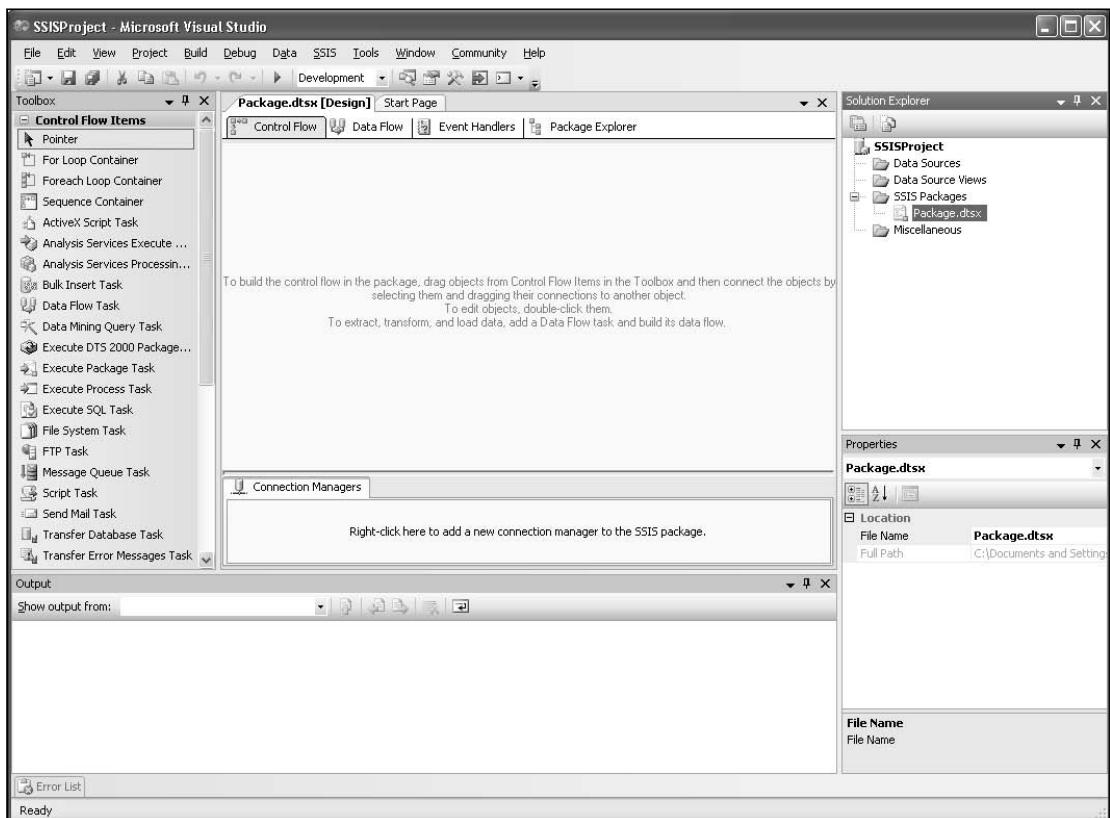


Figure 19-2

Chapter 19

For those of you used to the SQL Server 2000 environment, this screen will likely come as something of a minor shock—it really doesn't look much like the old thing at all. Instead, it is your run-of-the-mill Developer Studio project window. The only significant difference versus most Dev Studio projects is that, as we build the project, the design tab will be graphical in nature rather than code.

Perhaps the most significant difference versus SQL Server 2000 is the way that connections are treated. Under SQL Server 2000, you created new connections by dragging in objects that were grouped with the tasks. With Integration Services in SQL Server 2005, connections have been moved into the data source and Data Source View that we first looked at in our chapter on Reporting Services (see Chapter 17).

There are four key windows in our project, so let's start by looking at these. We will then do a walk-through example later in the chapter.

Tasks

On the left side of our project, we have the task window. It opens to the Control Flow Items list by default, but you should also find a section on Maintenance Plan tasks (these are more in the realm of the administrator, but you should take note of them—they underline my earlier notion that Integration Services is not just about ETL activities but also for a wide array of other actions, including many that you might have expected to find in Management Studio).

A task, much as the word implies, is generally about an action that you want to take. They range from migration tasks (such as moving objects between servers) to data migration and transformation to tasks that manage the execution of other programs or packages. Though they are called "tasks," you will also find some container objects that help organize or wrap the other objects in your package.

It's worth noting that you can reorganize the tasks. You can, for example, drag and drop tasks in the task list to reorder them (perhaps to move those you use the most often up to the top where they are more visible), or create your own tabs to contain those tasks you use the most often. In addition, you can add new tasks to the list much as you can add new controls to other Dev Studio projects. In short, the environment is very customizable.

There are a ton of tasks here, so let's take a quick overview at what the base tasks do.

Task	Description
Pointer	Okay, it's silly to even have to describe this, but just in case: This puts things into a generic drag and drop mode. When pointer is selected, clicking in the designer pane implies that you merely want to select an object that is already there as opposed to adding a new one.
For Loop Container	This is nothing more than a glorified FOR (or FOR/NEXT depending on your language of choice) statement. The FOR loop container allows you to initialize a control counter and set the conditions by which that counter is adjusted as well as under what conditions you exit the loop. Use this task to allow for controlled repetition of other tasks.

Task	Description
For Each Container	Again, this is your run of the mill FOR/EACH statement. Like the FOR loop, it allows for controlled repetition, but this time, rather than using a counter, the loop is based on iterating through a collection of some sort (perhaps a collection of tables or other objects). The object list can come from a wide variety of sources ranging from such things as ADO and ADO.NET rowsets to SMO object lists.
Sequence Container	I think of this one as something of a “subpackage.” The sequence container allows you to group up tasks and treat them as a single unit. This is useful for things like wrapping several tasks into a single transaction (thus allowing your overall package to contain several separate transactions—each potentially having many tasks to perform). Individual sequence containers can be made active or inactive conditionally, so you could, for example, turn off an entire set of tasks by disabling that sequence container (you could even do that programmatically based on conditions found in previous tasks!).
Script Tasks	One of those “what it sounds like” things — these let you run your own custom code using either any ActiveX scripting language (JavaScript or VBScript for example) or any .NET-based language. Use the ActiveX Script task for ActiveX languages, and use the Script task for .NET code.
Analysis Services Tasks	These allow you to construct or alter Analysis Services objects as well as execute them.
Bulk Insert Task	As you might guess, this allows for the bulk importing of data. It uses the same Bulk Insert facilities that you touched on in the BCP chapter, but allows the bulk operation to be part of a larger control flow. The Bulk Insert task is easily the fastest way for an SSIS package to get data into your system. Note, however, that any package containing a Bulk Insert task can be run only by a login that is a member of the sysadmins server role.
Data Flow Task	The Data Flow task wraps the connection between data sources along with any transformations you want to make in moving data between those data sources. The Data Flow task is among the most complex tasks in SSIS in that it operates as both a task and a container. The Data Flow task is a container in the sense that you associate several parts of a given data flow with it. The Data Flow tasks define sources as well as destinations of data as well as the transformations to take place between the source and destination. Editing Data Flow tasks will automatically take you to a different tab within the main editing window.
Data Mining Query Task	This task requires that you have already defined Data Mining Models in Analysis Services. You can utilize this task to run predictive queries and output the results into tables (you could then define additional tasks to make use of those tables).

Table continued on following page

Chapter 19

Task	Description
Execute Tasks	These are somewhat specific to what you want to execute. They can range from running other packages (there are separate tasks for running old DTS packages versus the newer SSIS packages) to executing external programs to running SQL scripts.
File System Tasks	These allow you to create, move, and delete files and directories. In a wide variety of SSIS environments, the ability to transfer files is key to both performance and execution of your package. For example, you may need to copy a file from a remote location to local storage for performance reasons as you perform operations against that file. Likewise, you may only have network access that allows you to read or create a file, but not change it—File System Tasks allow you to get just the right thing done.
FTP Tasks	This is something of a different slant on the File System Tasks notion. Instead, however, this allows you to use the FTP protocol to retrieve files (very handy for doing things like transferring files to or from vendors, customers, or other partners).
Message Queue Task	This allows you to send and receive messages via Microsoft Message Queue. This is actually a very powerful tool that allows for the delivery and/or receipt of files and other messages even when the remote host is not currently online. Instead, you can “queue” the file, and that host can be notified that the file is available the next time it is online. Likewise, files can be left in queue for your process to pick up when you execute the package.
Send Mail	Yup—yet another of those “what it sounds like” things. This allows you to specify a mail including attachments that may have been created earlier in your package execution. The only real trick on this one is that you must specify a SMTP connection (basically the outbound mail server) to use to send the mail. SSL and Windows-based authentication is also supported.
Transfer Tasks	These range from server migration tasks, such as transferring logins, error messages, master database stored procedures, to more straightforward transfers such as transferring a table.
Web Service Task	This allows you to execute a Web service method and retrieve the result into a variable. You can then make use of that result in the remaining tasks in your package.
WMI Tasks	Windows Management Instrumentation (WMI) is an API that allows for system monitoring and control. It is a Windows-specific implementation of Web-Based Enterprise Management (WBEM), which is an industry standard for accessing system information. SSIS includes tasks for monitoring WMI events (so you can tell when certain things have happened on your system) and for requesting data from WMI in the form of a WMI query. You could, for example, ask WMI what the total system memory is on your server.

Task	Description
XML Tasks	XML Tasks allows for a wide variety of XML manipulation. You can apply XSLT transformations, merge documents, filter the XML document using XPath, and the list goes on.
Maintenance Tasks	Much of this is outside the scope of this book, but this set of tasks allows you to perform a wide variety of maintenance tasks on your server. From a developer perspective, a key use here would be things like automatic a backup prior to a major import or another similar activity that is part of your package. Similarly, you may want to do index rebuilds or other maintenance after performing tasks that do major operations against a particular table.

The Main Window

This window makes up the center of your default SSIS package window arrangement in Dev Studio. The thing to note is that it has four tabs available, and each is something of its own realm—take a look at each of them.

It's worth noting that you can change from the default tab style interface to a window-based interface if you so choose. (It's in the options for Visual Studio.)

Control Flow

This is actually where the meat of your package comes together. No, a package isn't just made up of flow alone, but this is where you initially drag all your tasks in and establish the order in which they will execute.

Data Flow

As you place data flow objects into the control flow pane, they become available for further definition in the Data Flow pane. Data flow tasks require addition objects to define such things as data connections, sources, and destinations of data as well as actual transformations.

Event Handlers

SSIS packages create a ton of events as they execute, and this tab allows you to trap certain events and act upon them. Some of the more key events worth trapping include:

Event	Description
OnError	This is a glorious new feature with SSIS. DTS had a quasi-error handler, but it was weak at best. This gives you something far more robust.
OnExecStatusChanged	This event is triggered any time the task going into a different status. The possible statuses are idle, executing, abend (abnormal ending), completed, suspended, and validating. You can set traps for each of these conditions and have code run accordingly.

Table continued on following page

Event	Description
OnPostExecute	This one fires immediately after execution of the task is complete. In theory, this is the same as OnExecStatusChanged firing and having a status of completed, but I have to be honest and say I haven't tested this enough to swear to it.
OnProgress	This event is called regularly when any reasonably measurable progress happens in the package. This one is probably more useful when you're controlling a package programmatically than through one of the other execution methods but is nice from the standpoint of providing a progress bar for your end users if you need one.

There are several other event methods available, but the preceding gives you a flavor of things.

Package Explorer

This one I find the location of to be a little odd. In a nutshell, this one presents a tree control of your package, complete with all the event handlers, connections, and executables (which include any tasks you have defined in the package). The reason I find this one a little odd is because I would have expected something like this to be part of or at least similar to Solution Explorer. Nonetheless, it does give you a way of looking over your project at an overall package level.

Solution Explorer

This is pretty much just like any other explorer window for Dev Studio. You get a listing of all the files that belong to your solution broken down by their nature (packages and data source views, for example).

The Properties Window

This one is pretty much the same as any other property window you've seen throughout SQL Server and Dev Studio. The only real trick here is paying attention to what exactly is selected so you know what you're setting properties for. If you've selected an object within the package, then it should be that particular task or event object. If you have nothing selected, then it should be the properties for the entire package.

Building a Simple Package

Okay, it's time for us to put some application to all this. This is going to be something of a quick-and-dirty example run, but, in the end, we will have shown off several of the key features of SSIS.

Let's start with a little prep work. For this sample, we're going to be making use of a vbScript file that will generate some data for us to import. You can think of this script as simulating any kind of pre-process script you need to run before a major import or export.

Create a text file called `CreateImportText.vbs` with the following code:

```

Dim iCounter
Dim oFS
Dim oMyTextFile

Set oFS = CreateObject("Scripting.FileSystemObject")
Set oMyTextFile = oFS.CreateTextFile("C:\TestImport.txt", True)

For iCounter = 1 to 10
    oMyTextFile.WriteLine(cstr(iCounter) & vbTab & """TestCol" & cstr(iCounter) &
    """
)
Next

oMyTextFile.Close

```

This script, when executed, will create a new text file (or replace the existing file if it's there). It will add 10 rows of text to the file containing two tab-separated columns with a new line row terminator. We will use this in conjunction with a few other tasks to create and populate a table in SQL Server.

This is a pretty simplistic sample, so please bear with me here. What, in the end, I hope you see from this is the concept of running a preprocess of some sort (that's our script that generates the file in this case, but it could have been any kind of script or external process), followed by SQL Server scripting and data pump activities.

With our sample vbScript created, we're ready to start building a package.

Let's start with the "SSISProject" project file we created in the previous section. At this point, our Control Flow should be empty. In order to get the proverbial ball rolling on this, we need to make a call to our vbScript to generate the text file we will be importing. Drag an "Execute Process Task" from the Toolbox into the main Control Flow window. Very little will happen other than SSIS adding the Execute Process Task to the Control Flow window, as shown in Figure 19-3.



Figure 19-3

Chapter 19

To do much with our new task, we need to double-click the task to bring up the Execute Process Task Editor shown in Figure 19-4.

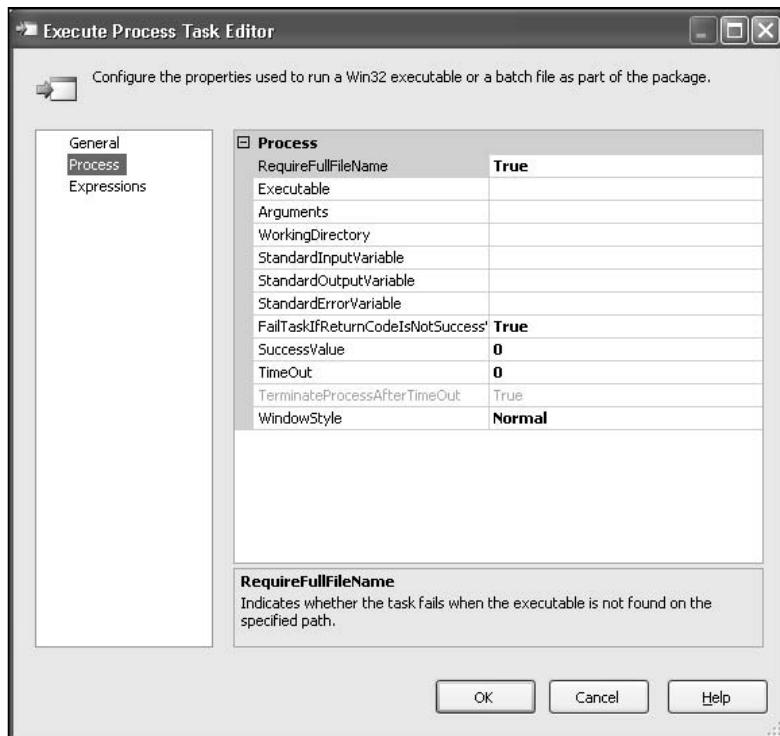


Figure 19-4

Note that I've switched to the Process options because they are a bit meatier to show in a screenshot than the General options are, but here's an overview of how we want to set things up:

Option	Setting
General \Rightarrow Name	GenerateImportFile
General \Rightarrow Description	Generates the text file for import
Process \Rightarrow Executable	CreateImportText.vbs (prefix it with the full path to your script file)
Process \Rightarrow Working Directory	C:\ (or other directory of your choosing—just make sure you're consistent)

When you're done making these changes, click OK, and very little will have changed in your Control Flow window, except that the name of the task will have been updated to `GenerateImportFile`.

Next, drag an Execute SQL Task object into your Control Flow. Now, select the `GenerateImportFile` task, and it should have an arrow hanging from the bottom of the task box, as shown in Figure 19-5.

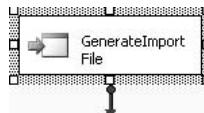


Figure 19-5

Now comes the tricky part: Click on the “output” of our `GenerateImportFile` task—that is, click on the end of the little arrow. Drag the arrow into the top of the Execute SQL Task, and the builder should connect the two tasks (as shown in Figure 19-6)—notice how the arrow indicates the control flow.

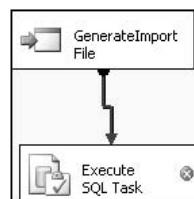


Figure 19-6

For the moment, let’s look at what this arrow represents. Double-click on it, and you’ll get a Precedence Constraint Editor, as shown in Figure 19-7.

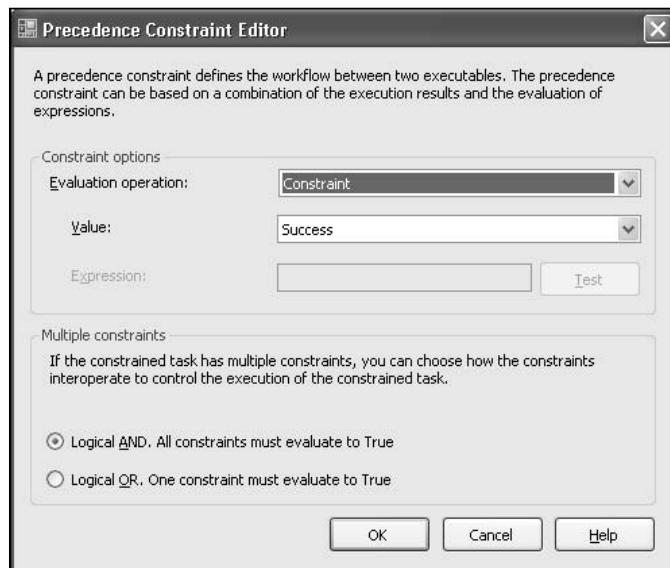


Figure 19-7

Chapter 19

Notice how it defines under what conditions this flow will be allowed to happen. In our case, it will move on to the Execute SQL Task only if our `GenerateImportFile` task completes successfully. We could define additional flows to deal with such things as the task failing or to allow for our second task to run on completion of the first task regardless of whether the first task succeeds or fails (any completion, regardless of success).

Cancel back out of this dialog, and double-click Execute SQL Task to bring up the Execute SQL Task Editor, as shown in Figure 19-8.

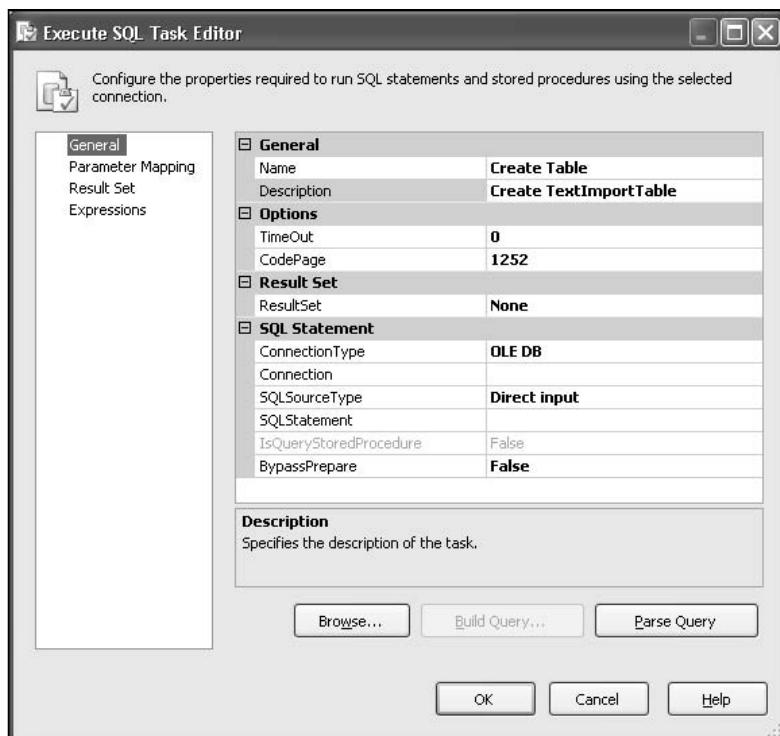


Figure 19-8

Again, I've edited the name a bit. Next, click on the `SQLStatement` option to bring up the Enter SQL Query dialog shown in Figure 19-9.

We're checking to see whether the table already exists, and, if it does drop it. Then, knowing that the table cannot already exist (if it did, we just dropped it), we go ahead and create our destination table.

One last thing we need is to have a connection to work with. Start by clicking in the `Connection` option and selecting "New Connection...." This will bring up a somewhat run-of-the-mill OLEDB connection manager dialog. How I've filled out mine is shown in Figure 19-10, but adjust yours to match your database server name (the simple period "." in mine implies that I mean my local server) and security model.

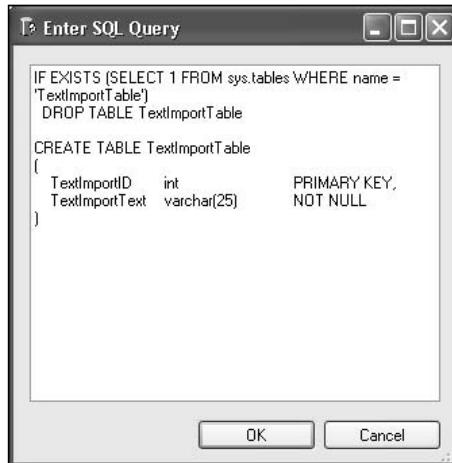


Figure 19-9

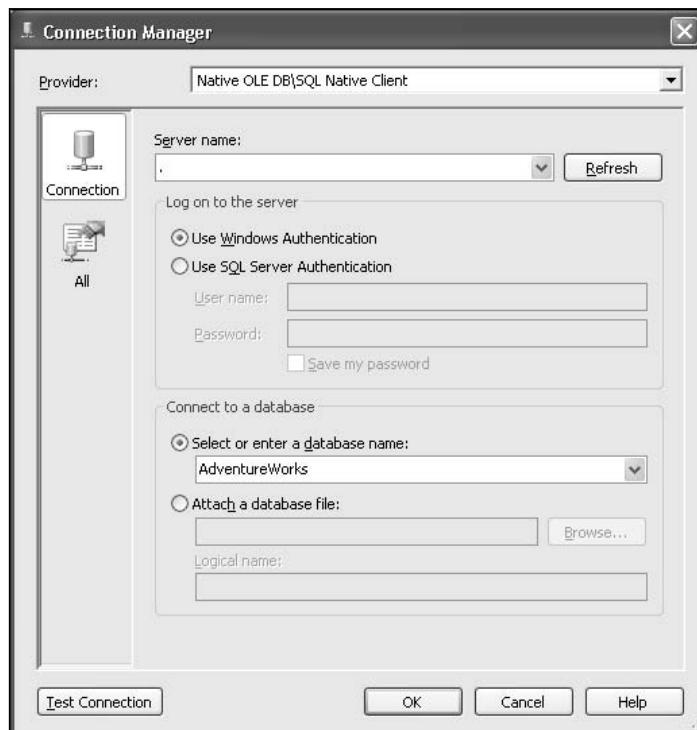


Figure 19-10

We're not able to connect to our database to create the destination table. And, with source data created and a destination table in place, we're ready to start working on actually transferring the data from our source to our destination. For that, we're going to utilize a Bulk Insert task, so go ahead and drag one of those into our model and connect the CreateTable task to the new BulkImport task, as shown in Figure 19-11.

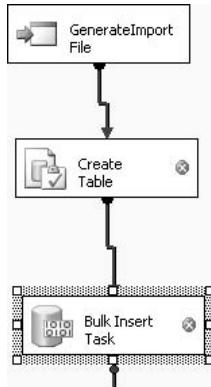


Figure 19-11

Again, double-click on our task (the Bulk Insert Task in this case) to bring up a relevant editor box. Of particular interest is the Connection tab shown in Figure 19-12.

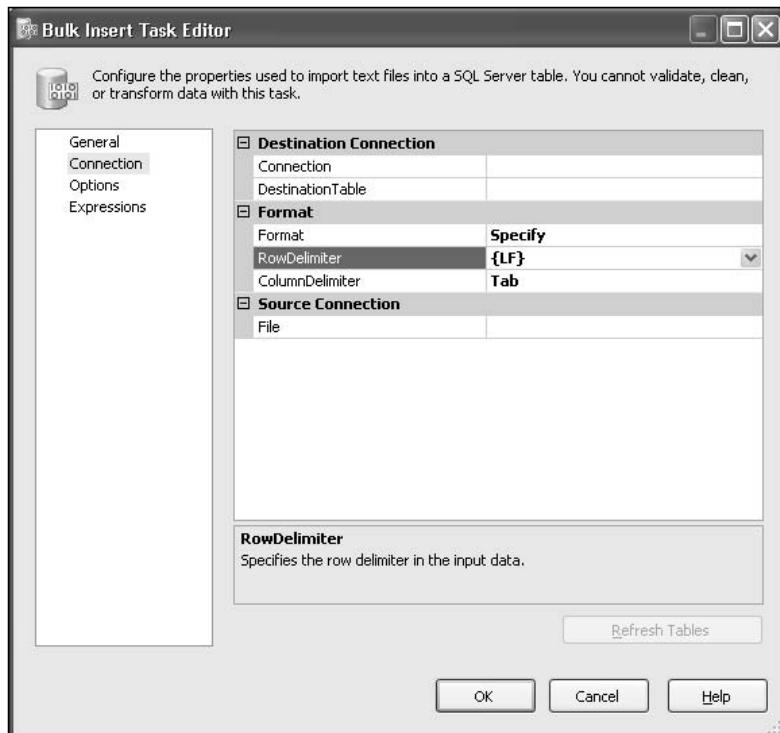


Figure 19-12

We have several things to change here. For example, I've already changed the Row Delimiter to be the line feed that is written by our vbScript's `WriteLine` command. We do, however, need to do even more. Start by selecting the same connection you created to run the `CREATE TABLE` statement against. Then enter in our destination table name (`[AdventureWorks].[dbo].[TextImportTable]`).

Note that the table must already exist for you to reference it in this dialog. I just manually run the `CREATE` statement once to prime the database and make sure anything that needs to reference the table at compile time can do so. This should create no harm since the process will drop the table and create a new one each time anyway.

Finally, click in the File connection box and select new connection to bring up the File Connection Management Editor for text files shown in Figure 19-13.

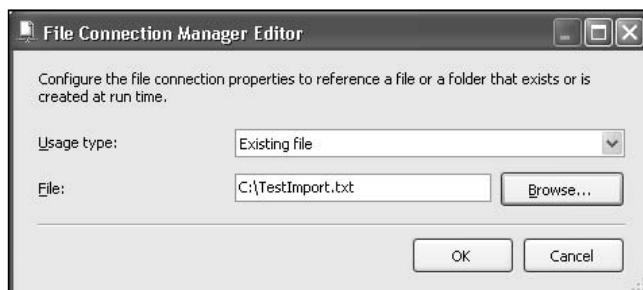


Figure 19-13

Click OK all the way back out to our Control Flow, and we're ready to rock.

To execute our package immediately, click the run (the green arrow on the toolbar). Watch how Dev Studio indicates progress by changing the color of different tasks as they run.

A few more items of note: SSIS is capable of running multiple tasks at one time for you. For example, I made this project entirely linear (one piece at a time) based on the idea that we didn't want to drop the destination data until the last minute (when we are sure there's new data available). We could, however, have placed the link from the file generation directly to the build import just the same as the `CREATE TABLE` dependency is linked directly to the import. If we had, SQL Server would have run both the table `DROP/CREATE` and the file creation at the same time but waited for "both" to complete before allowing the build import to execute.

Go ahead and build your package, as we will be utilizing it in the next section!

Executing Packages

There are a few different ways to execute an SSIS package. We utilized one of these in something of test mode within the Dev Studio, but this is hardly how you are likely to run your packages on a day-to-day basis. The more typical methods of executing a package include:

- ❑ **The Execute Package Utility**—This is essentially an executable where you can specify the package you want to execute, set up any required parameters, and have the utility run it for you on demand.
- ❑ **As a scheduled task using the SQL Server Agent**—I'll talk more about the SQL Server Agent in Chapter 24, but for now, realize that executing an SSIS package is one of the many types of jobs that the agent understands. You can specify a package name and time and frequency with which to run it, and the SQL Server Agent will take care of it.
- ❑ **From within a program**—There is an entire object model supporting the notion of instantiating SSIS objects within your programs, setting properties for the packages, and executing them. This is fairly detailed stuff—so much so that Wrox has an entire book on the subject *Professional SQL Server 2005 Integrations Services* by Knight, et. al (Wiley, 2006). As such, we're going to consider it outside the scope of this book other than letting you know it's there for advanced study when you're ready.

Using the Execute Package Utility

The Execute Package Utility is a little program by the name of `DTExecUI.exe`. You can fire it up to specify settings and parameters for existing packages and then execute them. You can also navigate using Windows Explorer and find a package in the file system (they end in `.DTSX`) and then double-click it to execute it. Do that to our text import package, and you should get the execute dialog shown in Figure 19-14.

As you can see, there are a number of different dialogs that you can select by clicking on the various options to the left. Coverage of this could take up a book all to itself, but let's look at a few of the important things on several key dialogs within this utility.

General

Many fields on this first are fairly self-explanatory, but let's pay particular attention to the Package Source field. We can store SSIS packages in one of three places:

- ❑ **The File System**—This is what you did on your Import/Export Wizard package. This option is really nice for mobility—you can easily save the package off and move it to another system.
- ❑ **SQL Server**—This one stores the package in SQL Server. Under this approach, your package will be backed up whenever you back up your MSDB database (which is a system database in every SQL Server installation).
- ❑ **SSIS Package Store**—This storage model provides the idea of an organized set of “folders” where you can store your package along with other packages of the same general type or purpose. The folders can be stored in either MSDB or the file system.

Configurations

SSIS allows you to define configurations for your packages. These are essentially a collection of settings to be used, and you can actually combine more than one of them into a suite of settings.

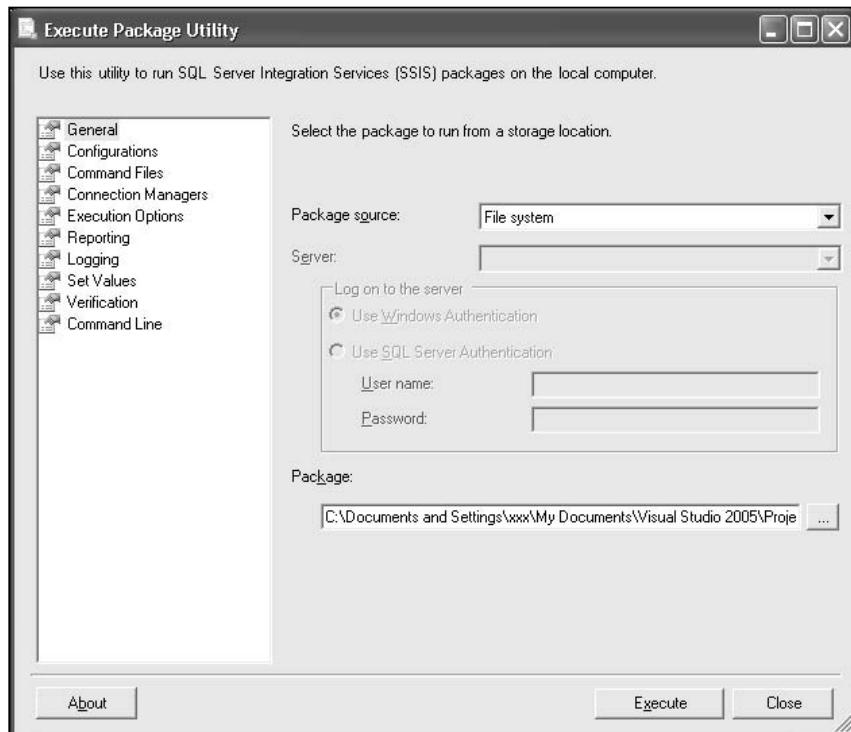


Figure 19-14

Command Files

These are batch files that you wish to run as part of your package. You can use these to do system-level things such as copying files around to places you need them (they will run under whatever account the Integration Services Service is running under, so any required access on your network will need to be granted to that account).

Connection Managers

This is a bit of misnomer — this isn't so much a list of connection managers as it is a list of connections. By taking a look at the Description column, you'll see many of the key properties for each connection your package uses. Notice that in our example package, we have two connections, and if you look closely, you'll see how one relates to file information (for our connection to the flat file we're using), and there is another that specifically relates to SQL Server (the export source connection).

Execution Options

Do not underestimate the importance of this one. Not only does it allow you to specify how, at a high level, you want things to happen if something goes wrong (if there's an error), but it also allows you to establish checkpoint tracking — making it easy to see when and where your package is getting to different execution points. This can be critical in performance tuning and debugging.

Reporting

This one is all about letting you know what is happening. You can set up for feedback: exactly how much feedback is based on which events you decide to track and the level of information you establish.

Logging

This one is fairly complex to set up and get going but has a very high “coolness” factor in terms of giving you a very flexible architecture for tracking even the most complex of packages.

Using this area, you can configure your package to write log information to a number of preconfigured “providers” (essentially, well-understood destinations for your log data). In addition to the preinstalled providers such as text files and even a SQL Server table, you can even create your own custom providers (not for the faint of heart). You can log at the package level, or you can get very detailed levels of granularity and write to different locations for different tasks within your package.

Set Values

This establishes the starting value of any runtime properties your package uses (there are none in our simple package).

Verification

Totally different packages can have the same filename (just be in a different spot in the file system for example. In addition, packages have the ability to retain different versions of themselves within the same file or package store. The Verification dialog is all about filtering or verifying what package/version you want to execute.

Command Line

You can execute SSIS packages from the command line (handy when, for example, you’re trying to run DTS packages out of a batch file). This option within the SSIS Package Execution Utility is about specifying parameters you would have used if you had run the package from the command line.

The utility will establish most of this for you—the option here is just to allow you to perform something of an override on the options used when you tell the utility to Execute.

Executing the Package

If you simply click Execute in the Package Execution Utility, your package will be off and running. After it runs, you should find a text file in whatever location you told your package to store it—open it up, take a look, and verify that it was what you expected.

Executing within Management Studio

While Management Studio doesn’t give you a package editor, it does give you the ability to run your packages.

In the Registered Servers pane of Management Studio, click on the icon for Integration Services, and then double-click the server you want to execute the package on (you may need to register your server

as an Integration Services server within Management Studio). This should create a connection to the Integration Services on that server, and add an Integration Services node in your Object Explorer.

To execute a package in this fashion (using Management Studio), the package must be local to that server (not in the file system). Fortunately, if you right-click the File System node under Stored Packages, SQL Server gives you the ability to import your package. Simply navigate the file system to the package we created, give it a name in the package store, and import it. You can then right-click and execute the package at any time. (It will bring up the execution utility we saw in a previous section, so you should be in familiar territory from here.)

Summary

SQL Server Integration Services is a robust Extract, Transform, and Load tool. You can utilize Integration Services to provide one-off or repeated import and export of data to and from your databases—mixing a variety of data sources while you’re at it.

While becoming expert in all that Integration Services has to offer is a positively huge undertaking, getting basic imports and exports up and running is a relative piece of cake. I encourage you to start out simple and then add to it as you go. As you push yourself further and further with what SSIS can do, take a look at other books that are specific to what SSIS has to offer.

20

Replication

Replication is one of those areas of big change for SQL Server 2005. If you're new to it, well . . . you had some adjusting to do to understand replication anyway, but I want to caution those of you who already know replication from SQL Server 2000 and prior that some fairly core things—particularly in the area of replication security—have changed, so listen up and pay attention!

Okay, okay—enough with the drill sergeant act. I'm going to guess that most of you reading this chapter are actually fairly new to replication. You see, replication is one of those things that everyone loves to ignore—until they need it. Then, it seems, there is a sudden crisis about learning and implementing it instantly (and not necessarily in that order I'm sorry to say).

So, what then, exactly, is replication? I'll shy entirely away from the Webster's definition of it and go to my own definition:

Replication is the process of taking one or more databases and systematically providing a rule based copy mechanism for that data to and potentially also from a different database.

Replication is often a topology and administration question. As such, many developers have a habit of ignoring it—bad idea. Replication has importance to software architects in a rather big way, as it can be a solution to many complex load and data distribution issues such as:

- Making data available to clients that are generally not connected to your main network
- Distributing the load associated with heavy reporting demands
- Addressing latency issues with geographically dispersed database needs
- Supporting geographic redundancy

And those are just a few of the biggies.

Chapter 20

So, with that in mind, we're going to take a long look at replication. I'm going to warn you in advance that this isn't going to have quite as many walkthroughs as I usually do, but patience, my young *padawan* — there is a reason. In simple terms, once you've built one or two of the styles of replication, you have most of the "constructing" part of the learning out of the way. What's more, the actual building up of the replication instance is indeed mostly an administrator's role. Instead, we're going to focus on understanding what's happened, and, from there, save most of the space in this chapter for understanding how different replication methods both create and solve problems for us and how we might use the different replication models to solve different problems.

In this chapter we will look at things like:

- ❑ General replication concepts
- ❑ What replication models are available (we will see an example or two here)
- ❑ Security considerations (an area of big change for SQL Server 2005)
- ❑ Replication Management Objects (RMO) — the programmatic way of managing replication

In the end, while I can't promise to make you a replication expert (to be honest, I'm not really one myself), you will hopefully have a solid understanding of the fundamentals and have a reasonable understanding of the possibilities.

Replication Basics

Replication is like a big puzzle — made up of many pieces in order to form a complete unit. We have topology considerations (publisher, subscriber, and distributor) as well as publication models (merge, transactional, snapshot). Before you get to deciding on those, there are several things to take into account.

Considerations When Planning for Replication

There are a number of things to take into account when thinking about the topology and replication methods available. These should be part of an assessment you make at design time to determine what forms of replication should even be considered for your application. Among these are:

- ❑ Autonomy
- ❑ Latency
- ❑ Data consistency

Let's take a quick look at each of these.

Autonomy

Autonomy is all about how much a replication instance is able to run as its own thing. What data needs to be replicated and at what frequency? For example, you could be supporting a sales application where each site keeps separate customer records. You would want to have these replicated to a central database for reporting and, perhaps, such other things as automatic stock replacement. Each site is highly autonomous (they really don't care whether the central database gets its data or not; they can still

continue to make sales based on the data they have on-site). Indeed, even the central database, while dependent, is probably not in a catastrophic situation if it misses data from a site for a day (depends how you're using the reports that come off it or how much lag you can have before you restock).

Latency

Latency refers to the time delay between updates; in other words, the time taken for a change at the publishing server to be made available at the subscribing server. The higher the autonomy between sites, the greater the latency between updates can be.

Determining an acceptable delay can be tricky and will likely be tied into the aforementioned autonomy question. If our site information is only transmitted to the central server for periodic rollup reporting, then we can probably get away with only daily—or even longer—updates. If, however, the sites are drawing from a central shipping facility for some of the sales, then we need to update the central database in a timelier manner, so a product is not oversold (two sites trying to sell the one remaining piece of stock).

Data Consistency

Data consistency is obviously going to be a key concern of virtually any distributed system. This is, of course, all about making sure that your various replication instances contain the same values from end to end, and this can be accomplished in two ways:

- ❑ **Data convergence**—All sites eventually end up with the same values; however, the values aren't necessarily the same as they would be if all of the changes had taken place on one server. An example might be our overstock situation. Had our two sales happened on the same server, then the second sale would have known about the out of stock situation and perhaps not been completed. Instead, each database thought one item was available, and, depending on the way the inventory adjustment is handled, you may wind up with a negative inventory level. In the same vein, your data may wind up with exactly the same end value, but may have taken a different set of steps to arrive at that value (the actual ordering of the updates may not be the same depending on how many replication clients were involved and at what time they synchronized).
- ❑ **Transactional consistency**—The results at any server are the same as if all transactions were executed on a single server. This is implemented by the mechanism implied in the name—transactions. I'm sure, if you ponder this for a bit, you can recognize the latency impact (both good and bad) of this—before your transaction can complete, it has to complete on every server that is participating in that particular replication set.

Schema Consistency

Many developers who are used to developing in non-replicated environments take the ability to easily change the database schema for granted. Need to add or drop a new column? No problem. Need to add a new table? No big deal. Well, beyond the basic problems of being so cavalier with your database in any environment, you'll quickly find that life gets a bit more complicated in a replicated world.

Replication or not, remember that any time you alter the schema of your table you are essentially altering the foundation of your entire system (or at least the part that the schema object in question serves). Schema changes should always be treated as fairly serious alterations and be carefully considered and methodically planned. Some changes (additions in particular) can usually be made with relatively minor collateral impact. Things that change or remove existing objects, however, can be deadly when dealing with backward-compatibility issues. Also, keep in mind that others may have built “extensions” to your system that are relying on your existing schema, this can mean impacts that are hard to plan for when you change your existing schema.

Chapter 20

The good news is that SQL Server continues to increase its support for schema changes during replication. Fields that are added or dropped on the publisher may be propagated to all subscribers during future replication operations. The bad news is that your change procedures need to be much stricter. The bottom line is that, if you need to make frequent schema changes, you'll want to fully plan what your change strategy is going to be before implementing replication at all.

When the concept of replicating schema changes was first added to SQL Server, it was done through the use of special stored procedures called sp_repladdcolumn and sp_repldropcolumn rather than the more familiar ALTER TABLE command. This has been changed for SQL Server 2005, and sp_repladdcolumn and sp_repldropcolumn should be considered deprecated (avoid using them).

Other Considerations

Some other things to think about include:

- ❑ How reliable is the connection between your servers? If it is a local connection, then you can probably count on it, but what if it is in a different geographic location? What if it's a different country?
- ❑ What kind of connection latency do you have? This falls somewhat into the reliability question, but is really its own issue. Do you really want to enforce transactional replication if it takes even a second or two for a simple ping to return (imagine that with a block of data now)?
- ❑ In the same vein as connection latency, how much bandwidth do you have? How much traffic are you going to be flushing over the wire, and what other processes are going to be using that same wire? Do you need to compress your replication related data?
- ❑ Is the replication method wire at all? That is, what if you don't have connectivity at all with the servers you want to replicate to? SQL Server supports a disconnected model, but what does that do to you between long updates?

Replication Roles

The process of replication is based on three basic roles: The publisher, distributor, and subscriber. Any one server can potentially be serving any one (or any subset) of these roles. Just to paint a picture at how flexible this can be, take a look at Figure 20-1.

As you can see, multiple publishers can be utilizing the same distributor, and any given publication can have multiple subscribers. Let's take a little bit closer look at these roles.

The Publisher

The publisher can be considered to be the source database. Even in situations where the publisher and its various subscribers are sharing data equally, there is one database that can be thought of as something of the control database.

The Distributor

The distributor serves as something of the clearinghouse for changes. It has a special distribution database that keeps track of changes, as well as which subscribers have already received those changes. In addition, it will keep track of the results of any synchronization process and will know what happened in the case of any conflicts that had to be resolved (we'll look more into conflict resolution later).

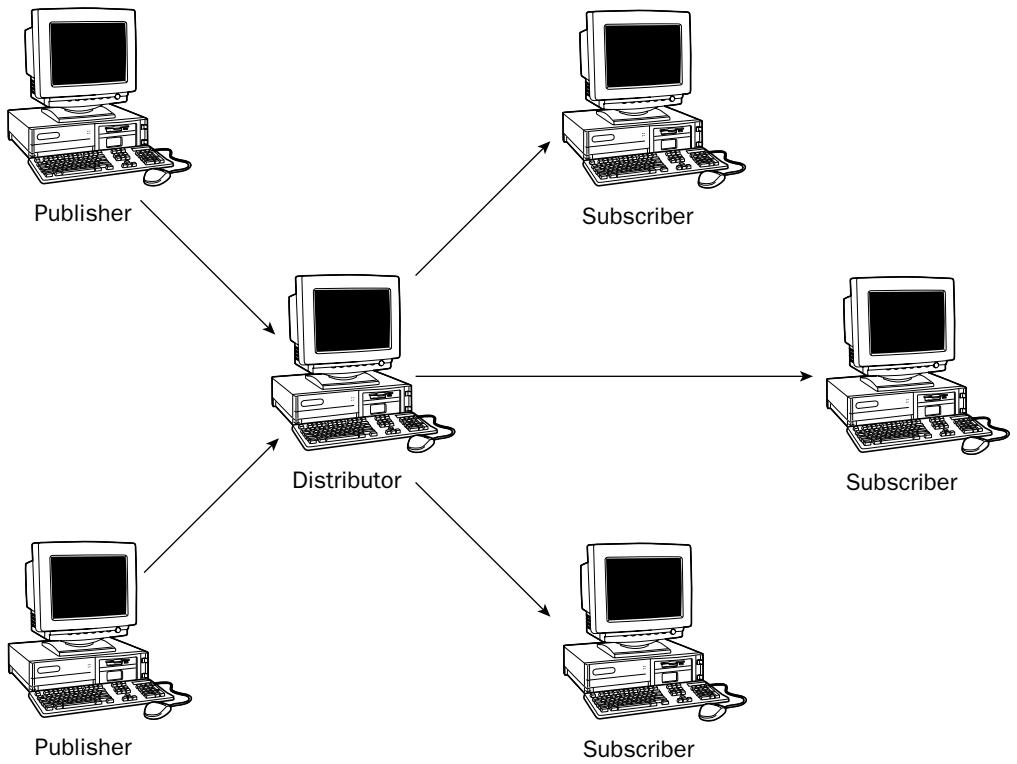


Figure 20-1

The Subscriber

Any database that is participating in the replication publication, but is not the actual publisher, can be considered a subscriber. This does not, however, mean that the subscriber only receives data—indeed, depending on the specific model chosen (again, more on those later), the subscriber may well be both receiving and disseminating data.

Subscriptions

The subscriptions that a subscriber receives are called *publications*. A publication will contain one or more *articles*. An article is usually a table or some subsection of the data from a table, but it can be a stored procedure or a group of stored procedures. By subscribing to a publication, the subscriber is subscribing to all of the articles in the publication. The subscriber cannot subscribe to individual articles alone.

Subscriptions can be set up as *push* subscriptions or *pull* subscriptions.

- ❑ With **push** subscriptions, the publisher determines when updates go out to the subscriber. This is used most frequently when you want to keep latency to a minimum (since the publisher is often the only copy of the database receiving changes, it makes sense that it would be the one to know about changes as they happen and take appropriate action) or you want to keep full control at the publisher for some other reason.

- ❑ With **pull** subscriptions the subscriber requests updates. This allows for a higher level of autonomy since the subscriber decides when updates should occur.

A publication can simultaneously support both push and pull subscriptions; however, any given subscriber is restricted to either a push or pull subscription—it cannot have both push and pull to the same publication.

Types of Subscribers

SQL Server supports three types of subscribers:

- ❑ The default is a **local** subscriber. The publisher is the only server that knows about the subscriber. Local subscribers are often used as a security mechanism or when you want to maximize autonomy between servers.
- ❑ **Global** subscribers occur where all servers participating in the publication (be they the publisher or a subscriber) know about all the other subscribers. Global subscribers are commonly used in a multiserver environment where you want to be able to combine data from different publishers at the subscriber.
- ❑ **Anonymous** subscribers are visible only to the publisher while the subscriber is connected. This is useful when setting up Internet-based applications.

Filtering Data

SQL Server provides for the idea of horizontally or vertically filtering tables. *Horizontal filtering* (you may come across the term *horizontal partitioning* for this as well) identifies rows within the table (by way of WHERE clause) for publication. For example, you could divide inventory information by warehouse as a way of maintaining separate warehouse totals. *Vertical filtering* (also known as *vertical partitioning*) identifies the columns to be replicated. For example, you might want to publish quantity on hand information from an inventory table, but not quantity on order.

Replication Models

We have three different models available to us in replication. They trade off between the notions of latency, autonomy, and some of the other considerations we discussed earlier in the chapter. Deciding which to choose is something of a balancing act between:

- ❑ **Degree of autonomy**—Is there a constant connection available between the servers? If so, what kind of bandwidth is available? How many transactions will be replicating?
- ❑ **Conflict management**—What is the risk that the same data will be edited in multiple locations either at the same time or in between replicated updates? What is the tolerance for data on one or more of the replicated servers disagreeing?

Some replication scenarios don't allow for connectivity except on a sporadic basis—others may never have connectivity at all (save, perhaps, through what is sarcastically referred to as “sneaker net”—where you run, mail, fly, or the like, a disk or other portable storage medium from one site to another). Other replication scenarios have an absolute demand for perfectly consistent data at all sites with zero data loss.

From highest to lowest in autonomy, the three models are:

- Snapshot** replication
- Merge** replication
- Transactional** replication

Let's look at the pros and cons of each replication model, outlining situations where it would be an appropriate solution and any data integrity concerns.

It's important to note that you can mix and match the replication types as necessary to meet your implementation requirements. There are going to be some publications where you want to allow greater autonomy between sites. There will be other publications where minimizing latency is critical.

Let me take a moment here to point out that a publication is just that—a publication. It does not necessarily map out that one publication equals one database. You may have one publication where the articles included in it make up only part of your subscribing database. Other objects in the subscribing database may be served by a different publication—potentially from a completely different publishing server.

Snapshot Replication

With *snapshot replication*, a “picture” is taken at the source of all of the data to be replicated (as shown in Figure 20-2). This is used to replace the data at the destination server.

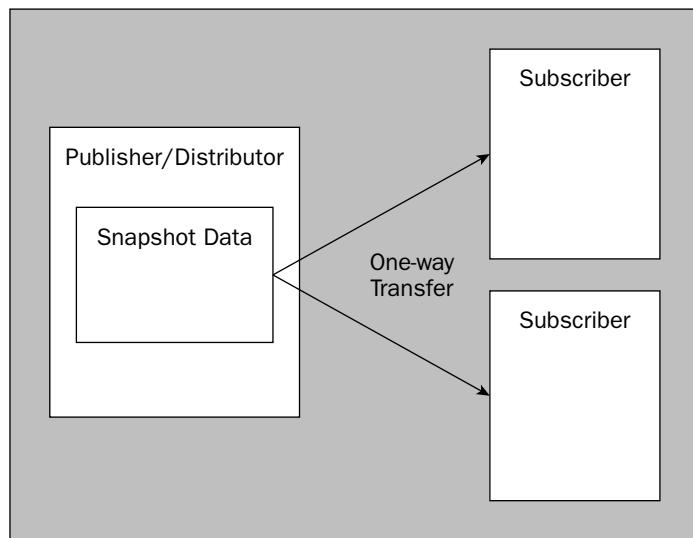


Figure 20-2

Snapshot replication, in its simplest form, is the easiest type of replication to set up and manage. Complete tables or table segments (for partitioned tables) are written to the subscribers during replication. Since updates occur on a periodic basis only, most of the time, there is minimal server or network overhead required to support replication.

Snapshot replication is frequently used to update read-only tables on subscriber systems. It allows for a high level of autonomy at the subscriber, but at the cost of relatively high latency. You are able to keep tight control on when periodic updates occur when using snapshot replication. This means that you can schedule updates to occur when network and server activity is at a lull (or you can even carry the snapshot via disk or other hard medium). There is a potential concern about the time and resources to complete replication during the periodic updates. As source tables grow, the amount of data that has to be transferred during each update increases. Over time, it may become necessary to either change the replication type or partition the table to reduce the amount of data replicated to keep traffic to manageable levels.

A variation of snapshot replication is snapshot replication with immediate-updating subscribers. With this, changes can be made to the data at the subscriber. Those changes are sent to the publishing server on a periodic basis unless immediate updating has been implemented, in which case distributed transactions are executed in real time.

How Snapshot Replication Works

Replication is implemented through *replication agents*. Each agent is essentially its own, small, independent program that takes care of the tasks of monitoring transactions and distributing data as required for that particular type of agent.

Snapshot Agent

The *Snapshot Agent* supports snapshot replication and initial synchronization of data tables for other types of replication (which all also rely on a snapshot for synchronizing data for the first time). All types of replication require that the source and destination tables must be synchronized, either by the replication agents or through manual synchronization, before replication can begin. In either case, the Snapshot Agent has the same responsibility. It takes the “picture” of the published data and stores the files on the distributor.

Distribution Agent

The *Distribution Agent* is used for moving data for initial synchronization and snapshot replication (and, as we'll see later, for transactional replication) from the publisher to the subscriber(s). For push subscriptions, the Distribution Agent typically runs on the distributor. For pull subscriptions, the Distribution Agent typically runs on the subscriber. The actual location of the Distribution Agent is an option that can be configured within Management Studio or via RMO.

The Process of Snapshot Replication

Snapshot replication uses periodic updates (the frequency is up to you, but, in general, you'll schedule a job in the job manager to run your snapshot on a regular basis). During the updates, schemas and data files are created and sent to the subscribers. Let's step through the basic procedure (see Figure 20-3):

1. The Snapshot Agent places a shared lock on all articles in the publication to be replicated, ensuring data consistency.
2. A copy of each article's table schema is written to the distribution working folder on the distributor.

3. A snapshot copy of table data is written to the snapshot folder.
4. The Snapshot Agent releases the shared locks from the publication articles.
5. The Distribution Agent creates the destination tables and database objects, such as indexes, on the subscriber and copies in the snapshot data, overwriting the existing tables, if any.

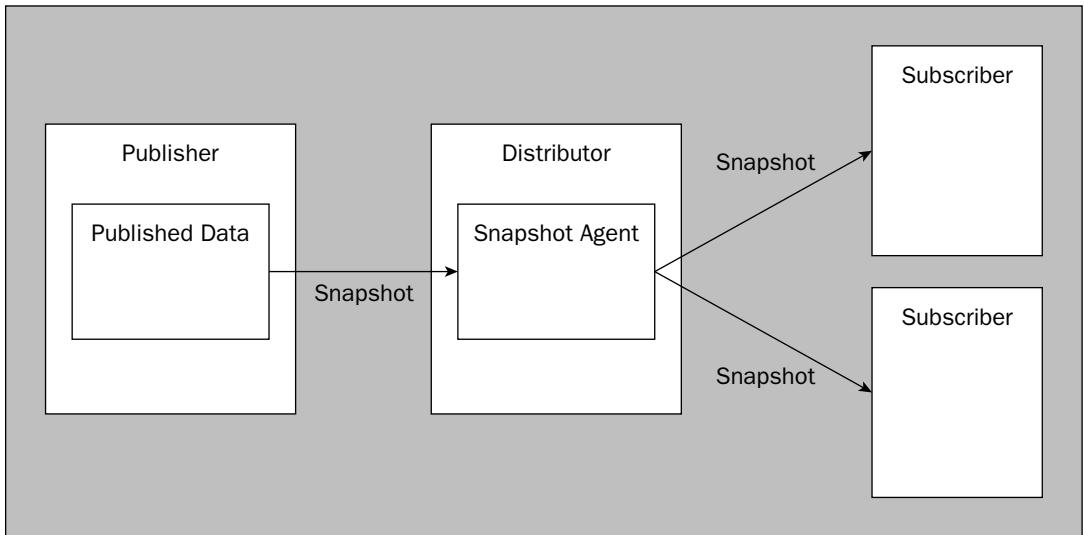


Figure 20-3

Snapshot data is stored as a native BCP file if all of the subscribers are Microsoft SQL Servers. Character mode files, instead of SQL Server BCP files, will be created if you are supporting heterogeneous (non-SQL Server) data sources.

SQL Server supports heterogeneous data sources for replication. Currently, transactional and snapshot replication are supported for all O/S platforms for Oracle as well as most O/S platforms for DB2.

When to Use Snapshot Replication

Use snapshot replication to update lookup data or read-only copies of data on remote servers. You can use snapshot replication when you want (or need) to connect to the publisher only intermittently.

As an example, think of how servers might be managed for a chain of garden supply stores. You have stores in several cities. Some larger cities have multiple stores. What are some good candidates for snapshot replication?

Customer records are an obvious choice. A customer, such as a landscape gardener, may turn up at different locations. In most cases, it won't matter if there's a delay updating customer information. This would also give you a way to make sure that only users who have access to the publishing server can change customer records.

Inventory records could be a little more of a problem. The items you keep in inventory are somewhat constant with most changes taking place by season. Even then, you would probably keep the items in file, but with a zero quantity on hand. The problem is, you may want to replicate more up-to-date inventory records between stores. This would let you search for items you might not have on hand without having to call each of the stores. Timely updates would most likely mean transactional replication (which we will discuss shortly).

Special Planning Requirements

An important issue when setting up snapshot replication is timing. You need to make sure that users are not going to need write access to any published tables when the Snapshot Agent is generating its snapshot (remember that share lock that gets set on every article in the publication? Well, that's going to prevent inserts, updates, and deletes to that data for the duration of that lock—which is to say for the duration of the publishing of the distribution). You also want to be sure that the traffic generated by replication does not interfere with other network operations.

Storage space can also become an issue as published tables grow. You have to verify that you have enough physical disk space available on the destination folder (CD-ROM, DVD, jump drive, tape, etc.) to support the snapshot folder.

Merge Replication

Snapshot is great, but we do not always live in a “read-only” world. Among the choices for dealing with data changes taking place at multiple servers is through the use of *merge replication*. The changes from all of the sites are merged when they are received by the publisher (see Figure 20-4). Updates can take place either periodically (via schedule—this is the typical way of doing things) or on demand.

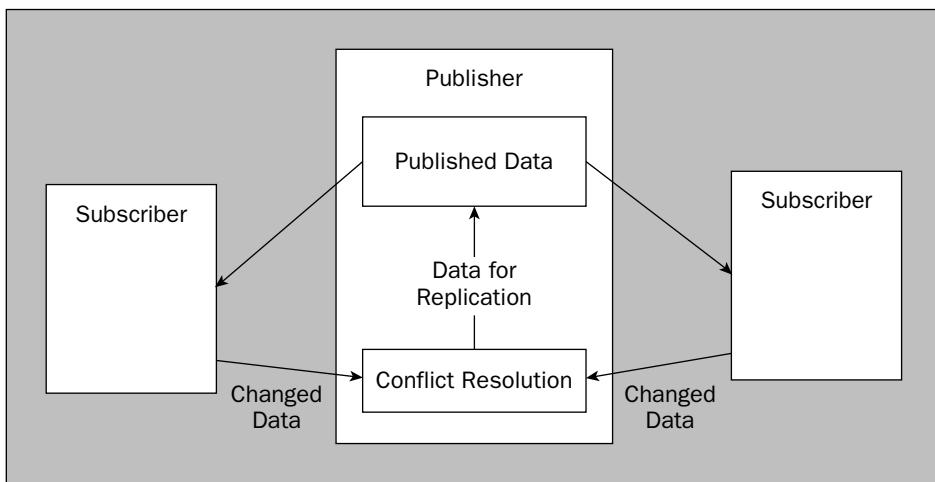


Figure 20-4

Merge replication has a high level of autonomy, but also has high latency and runs a risk of lower transactional consistency. Unlike transactional and snapshot replication, which guarantee consistency, merge replication does not. This is one of the more critical design considerations that you need to make when implementing merge replication—how important is consistency?

In a way, roles tend to get somewhat blurred in merge replication. The publisher is the initial source for the merge data, but changes can be made at the publisher or the subscribers. Changes can be tracked by row or by column. Transactional consistency is not guaranteed because conflicts can occur when different systems make updates to the same row. Data consistency is maintained through conflict resolution based on criteria you establish (you can even write custom resolution algorithms). You can determine whether conflicts are recognized by row or by column.

As with transactional replication, the Snapshot Agent prepares the initial snapshot for synchronization. The synchronization process is different, however, in that the Merge Agent performs synchronization. It will also apply any changes made since the initial snapshot.

Merge Agent

Just as we saw with snapshot replication, merge replication uses an agent—the *Merge Agent*. As shown in Figure 20-5, the agent copies the changes from all subscribers and applies them to the publisher. It then copies all changes at the publisher (including those made by the Merge Agent itself during the resolution process) to the subscribers. The Merge Agent typically runs on the distributor for push subscriptions and on the subscriber for pull subscriptions, but as with the snapshot and transactional replication, this can be configured to run remotely.

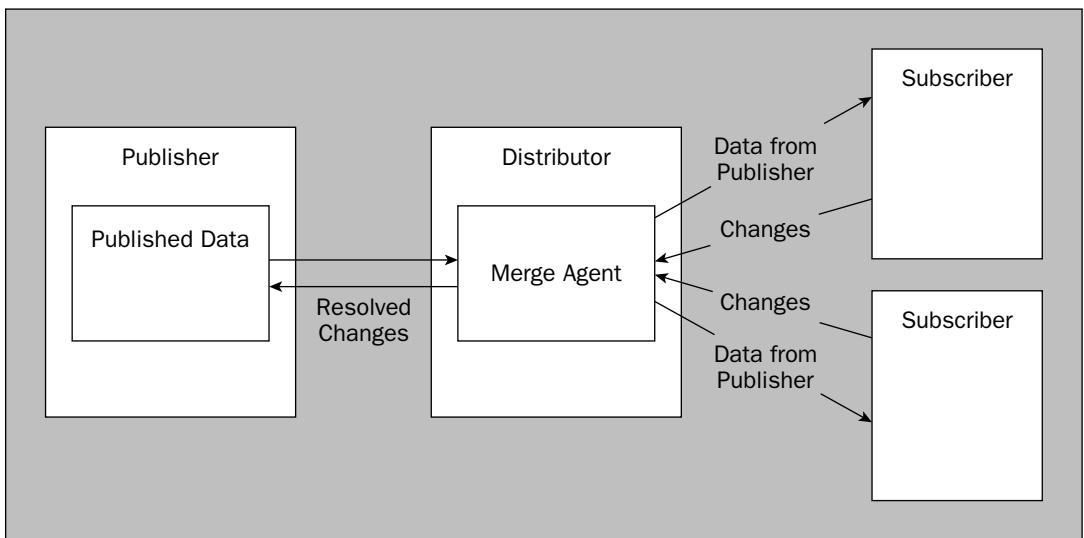


Figure 20-5

The Process of Merge Replication

Assuming that the initial synchronization has already taken place (remember, that will be based on a snapshot), the steps to merge replication are:

1. Triggers installed by SQL Server track changes to published data.
2. Changes from the publisher are applied to subscribers.
3. Changes from subscribers are applied to the publisher, and any conflicts resolved.

Merge triggers do not interfere with the placement or use of user-defined triggers.

Changes, whether occurring at the publisher or subscriber, are applied by the Merge Agent. Conflicts are resolved automatically through the Merge Agent, using a conflict resolver (you can select one and can even build your own). The Merge Agent tracks every row update for conflicts at the row or column level, depending on how you have configured conflict resolution. You will define the priority scheme to be used when conflicts occur between new (arriving) and current data values.

When to Use Merge Replication

One way of using merge replication is to support partitioned tables. Going back to the garden supply business, you could set up filtering (partitioning) so that each store can view inventory information for any store but would only be able to directly update its own inventory. Changes would be propagated through merge replication. Data can be filtered horizontally or vertically. You can exclude rows to be replicated from a table, and you can exclude any table columns. Merge replication watches for changes to any column in a replicated row. In this particular scenario, there is little risk of conflict in inventory since each store can only update its own inventory, but what if you were allowing all stores to update customer data (such as a new address for the customer)? The right answer is situational, but this illustrates how different needs can place a different burden on your replication design.

Special Planning Requirements

When implementing merge replication, there are checks that you need to make to ensure that your data is ready for replication. While setting up merge replication, some changes will be made automatically by SQL Server to your data files. Use care when selecting the tables to be published. Any tables required for data validation (such as lookup tables and other foreign key situations) must be included in the publication if you want that validation to apply on the subscribers.

SQL Server will identify a column as a globally unique identifier for each row in a published table. If the table already has a `uniqueidentifier` column, SQL Server will automatically use that column. Otherwise, it will add a `rowguid` column (which will, as it happens, also be called `rowguid`) to the table and create an index based on the column.

There will be triggers created on the published tables at both the publisher and the subscribers. These are used to track data changes for Merge Agent use based on row or column changes.

There will also be several tables added for tracking purposes. These tables are used by the server to manage:

- Conflict detection and resolution
- Data tracking
- Synchronization
- Reporting

For example, conflicts are detected through a column in the `MSmerge_contents` table, one of the tables created when you set up merge replication.

Transactional Replication

The difference between *transactional replication* and snapshot replication is that incremental changes, rather than full tables, are replicated to the subscribers. Any changes logged to published articles, such as `INSERT`, `UPDATE`, and `DELETE` statements, are tracked and replicated to subscribers. In transactional replication, only changed table data is distributed, maintaining the transaction sequence. In other words, all transactions are applied to the subscriber in the same order that they were applied to the publisher.

Note that only logged actions are properly replicated. Unlogged bulk operations (such as a BCP that has logging turned off) or Binary Large Object (BLOB) operations that do not generate full log entries will not be properly replicated.

In its simplest form, as shown in Figure 20-6, changes can only be made at the publisher. Changes can be replicated to subscribers at set intervals or as near-real-time updates. While you may have less control over when replication occurs, you are typically moving less data with each replication. Updates are occurring much more often and latency is kept to a minimum. Reliable and consistent near real-time subscriber updates (immediate transactional consistency) require a reliable network connection between the publisher and subscriber (make sure you have the bandwidth on your connection to handle the chatter between the publisher and the subscriber if it is a very high update frequency and/or volume).

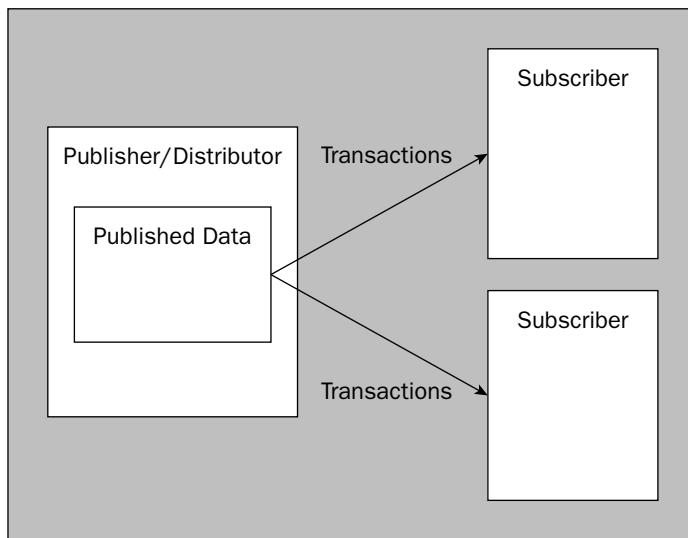


Figure 20-6

Just as with merge replication, the published articles must be initially synchronized between the publisher and the subscriber before transactional replication can take place. This is typically managed through automatic synchronization, using snapshot replication. In situations where automatic synchronization is neither practical nor efficient, manual synchronization can be used to prepare the subscriber. This is a relatively simple process:

1. Run `BACKUP DATABASE` to back up the Publisher database.
2. Deliver the tape backup to the subscriber system.
3. Run `RESTORE DATABASE` to create the database and database objects, and to load the data.

The publisher and subscriber are synchronized as of the point when the backup was run.

Transactional replication can also be used to replicate stored procedures. In its simplest implementation, changes can only be made at the publishing server. This means that you don't have to worry about conflicts.

You can also implement transactional replication as transactional replication with immediate-updating subscribers. This means that changes can be made at the publisher or at the subscriber. Transactions occurring at the subscriber are treated as distributed transactions. Microsoft Distributed Transaction Coordinator (MS DTC) is used to ensure that both the local data and data on the publisher are updated at the same time to avoid update conflicts. Queued updating—where updates are placed in an ordered “to be done” list—can be used as a fallback in the event that there is a network connectivity issue such as a disconnection or if the network is physically offline.

Another option would be to implement distributed transactions directly rather than using transactional replication. This will get you a lower latency than that provided with transactional replication, but you will still have the distribution delay in getting changes posted at the publisher out to all of the subscribers. Assuming a solid connection between the servers involved, distributed transactions could provide near immediate updates to all servers when data is changed at any server. However, depending on the connection speed and reliability between servers, this could result in performance problems, including locking conflicts.

Log Reader Agent

The *Log Reader Agent* is used in transactional replication. After a database is set up for transactional replication, the associated transaction log is monitored by the Log Reader Agent for changes to published tables. The agent then has responsibility for copying those transactions marked for replication from the publisher to the distributor as shown in Figure 20-7. The *Distribution Agent* is also used in transactional replication and is responsible for moving transactions from the distributor to the subscriber(s).

The Process of Transactional Replication

Assuming that initial synchronization has already taken place, transactional replication follows these basic steps:

1. Modifications are posted to the publisher database and recorded in the associated transaction log.
2. The Log Reader Agent reads the transaction log and identifies changes marked for replication.
3. Changes taken from the transaction log are written to the distribution database on the distributor.
4. The Distribution Agent applies the changes to the appropriate database tables.

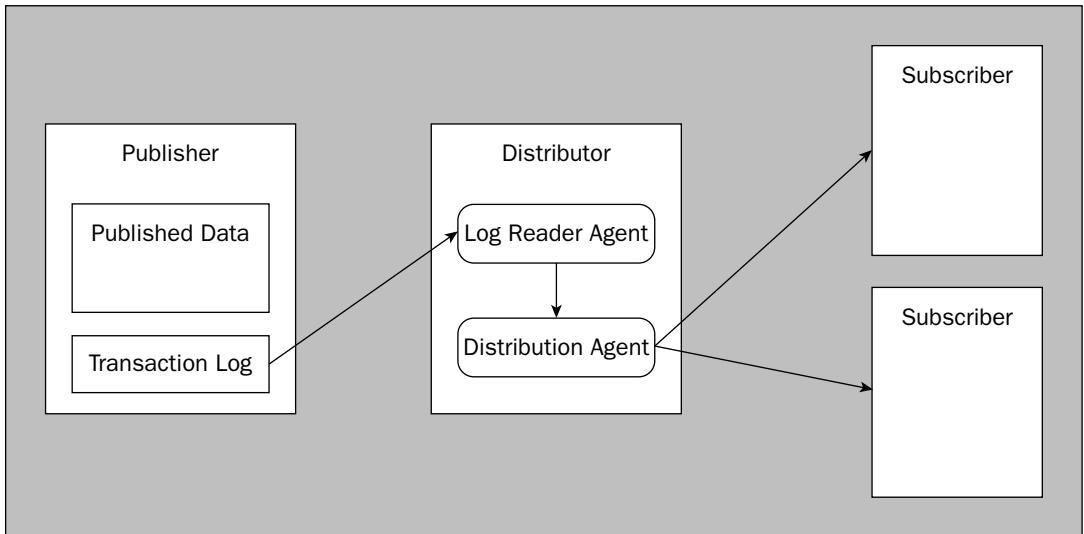


Figure 20-7

You can set up the Log Reader Agent to read the transaction log continuously or on a schedule that you specify. As before, the Distribution Agent typically runs at the publisher for push subscriptions and at the subscriber for pull subscriptions, but this can be changed through Management Studio or RMO to run remotely.

When to Use Transactional Replication

Use transactional replication when you need or just want to reduce latency and provide subscribers with relatively up-to-date information. Near real-time updates usually require a local area network connection, but scheduled replication can often be managed through scheduled updates. If you choose to use scheduled updates, latency increases, but you gain control over when replication occurs.

Let's go back to our garden supply store and the inventory problem discussed earlier. You want each of the stores to have up-to-date, or at the very least, relatively up-to-date, inventory information. You would probably use scheduled replication to pass data to the subscribers.

Now let's see if we can make things a little more difficult. Not only do you have a chain of stores; you also have traveling sales people who visit and take orders from your largest customers. They need to have at least relatively up-to-date inventory information but can spend their days sitting around and waiting for updates from the publisher. For systems of this type, you may want to use pull subscriptions, letting the sales people decide when they connect to the server and download recent transactions.

You've probably noticed a potential problem in both of these scenarios. The remote servers can receive data, but they are not able to make any changes to the data. We'll cover that problem a little later. Transactional replication, when implemented in this manner, is used to support read-only copies of the data at subscriber systems.

Special Planning Requirements

Space is an important issue when planning for transactional replication. You have to make sure that you allow adequate space for the transaction log on the publisher and for the distribution database on the distributor.

Check each of the tables that you are planning to publish. For a table to be published under transactional replication, it must have a primary key. There are also potential concerns if you are supporting text or image data types in any of the tables. `INSERT`, `UPDATE`, and `DELETE` are supported as for any data type, but you must be sure to use an option that utilizes the transaction log when performing BLOB or bulk operations.

You may encounter problems with the `max text repl size` parameter, which sets the maximum size of text or image data that can be replicated. Make sure that this server-level parameter is set to a high enough value to support your replication requirements.

Immediate-Update Subscribers

As indicated earlier in the chapter, you have the option of setting up subscribers to snapshot or transactional publications as immediate-update subscribers. Immediate-updating subscribers have the ability to update subscribed data, as long as the updates can be immediately reflected at the publisher. This is accomplished using the two-phase commit protocol managed by MS DTC. There is effectively no latency in updating the publisher. Updates to other subscribers are made normally (as if the change was initiated at the publisher), so latency when going to other subscribers will depend on the rate at which those subscribers are updated.

You should consider immediate-updating subscribers when you need to post changes to replicated data at one or more subscribers and propagate near-immediate updates. You might be using multiple servers to support an Online Transaction Processing (OLTP) application as a way of improving performance and providing near real-time redundancy. When a transaction is posted to any server, it will be sent to the publisher, and through the publisher, to the remaining servers.

Much as with any form of merge replication, conflicts can arise when using immediate-updating subscribers. In order to assist with conflict identification and management, a `uniqueidentifier` column will be added to any published tables that do not already have one (if your table has one, the column in question will have a column level property of `IsRowGUID` of true—you can only have one `RowGUID` column per table).

A high-speed, *reliable* connection is required between the publisher and any immediate-updating subscribers, such as a local area network connection, unless queued updates are used. If queued updates are configured, then the replication process can tolerate an unreliable connection and will just process any queued transactions as soon as connectivity is restored.

Keep in mind that queued updates increase the opportunities for you to have a conflict. Since the subscriber is making changes that the publisher does not know about, there is the increased prospect for the publisher to be making changes to the same rows that the subscriber is. In such a case, the conflict resolver will identify the existence of the conflict when replication occurs and resolve it according to whatever rules you have established.

Mixing Replication Types

You can mix and match replication types as needed. Indeed, not only can you have different replication types on the same server; you can even have different replication types for the same table.

As an example of why you might want to do this, imagine that a heavy equipment warehouse wants to have up-to-date inventory information and reference copies of invoices available at each of its locations. Each location has its own local SQL Server. Invoices are posted to a central location using an Internet-based application. These are replicated to all local servers through transactional replication so that inventory records are updated. You also want to have invoice and inventory information replication updated to yet another server weekly. This information on this last server is used for business analysis and running weekly reports. This server is updated weekly through a separate snapshot publication referencing the same tables used by the distributed inventory servers that were getting immediate updates.

Replication Topology

Over the years, Microsoft has outlined a number of replication topology models to describe how replication can be physically implemented. Let's look at some of these here as examples of how things are commonly implemented. It's worth noting that it is not only possible to mix and modify these models but actually rather common to do so.

Your decisions about the type of replication you need to use and your replication model topology can be made somewhat independent of each other. That said, there is a chance that restrictions imposed by your physical topology, such as transmission bandwidth, will influence your decisions.

Simple Models

Let's start with a look at the more simple models. Once you've got the basic idea, we can move on to some variations and ways these models are mixed.

Central Publisher/Distributor

This is the default SQL Server model. As shown in Figure 20-8, you have one system acting as publisher and as its own distributor. This publisher/distributor supports any number of subscribers. The publisher owns all replicated data and is the sole data source for replication. The most basic model assumes that all data is being published to the subscribers as read-only data. Read-only access can be enforced at the subscriber by giving users SELECT permission only on the replicated tables.

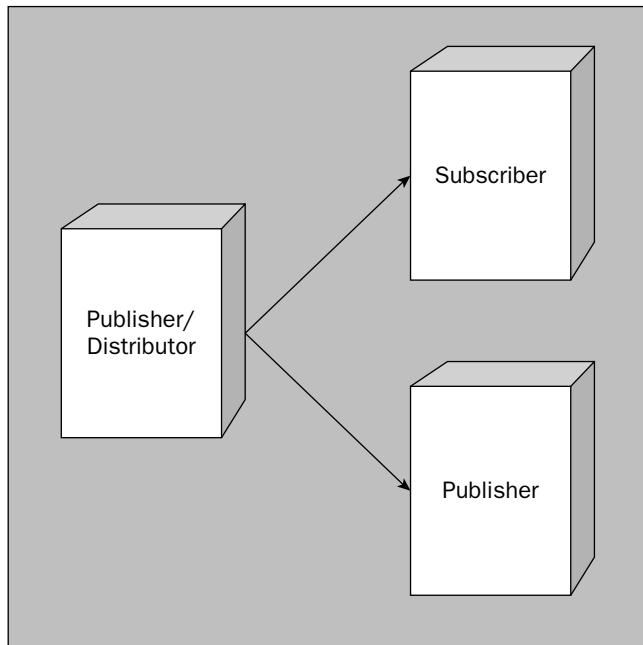


Figure 20-8

Since this is the easiest model to set up and manage, you should consider its use in any situation where it fits. If you have a single publisher, one or more subscribers, and read-only access to data at the subscriber, this is your best choice.

Central Publisher/Remote Distributor

You may find that the volume of replicated data and/or the amount of activity at the publisher may create the need to implement the publisher and distributor as separate systems. As shown in Figure 20-9, this is effectively, from an operational point of view, the same as the publisher/distributor model. The publisher is still the owner of—and only source for—replicated data. Once again, the simple model assumes that the data will be treated as read-only at the subscriber.

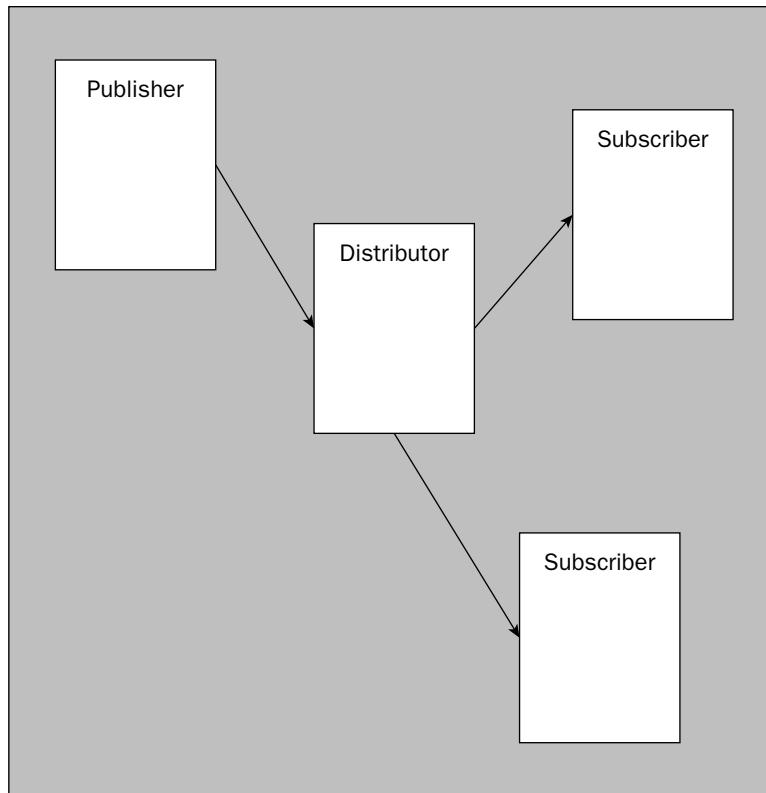


Figure 20-9

Obviously, you usually only use this model when a single publisher/distributor cannot handle both production activity and replication to subscribers.

Central Subscriber

In this model, as shown in Figure 20-10, you have only one subscriber receiving data, but there are multiple publishers. The publishers can be configured as publisher/distributor systems. This model provides a way to keep just local data at the local server but still have a way of consolidating the data at one central location. Horizontal filtering may be necessary to keep publishers from overwriting each other's data at the subscriber.

This is the model to use when you have data consolidation requirements such as gathering distributed data up for use in a data warehouse.

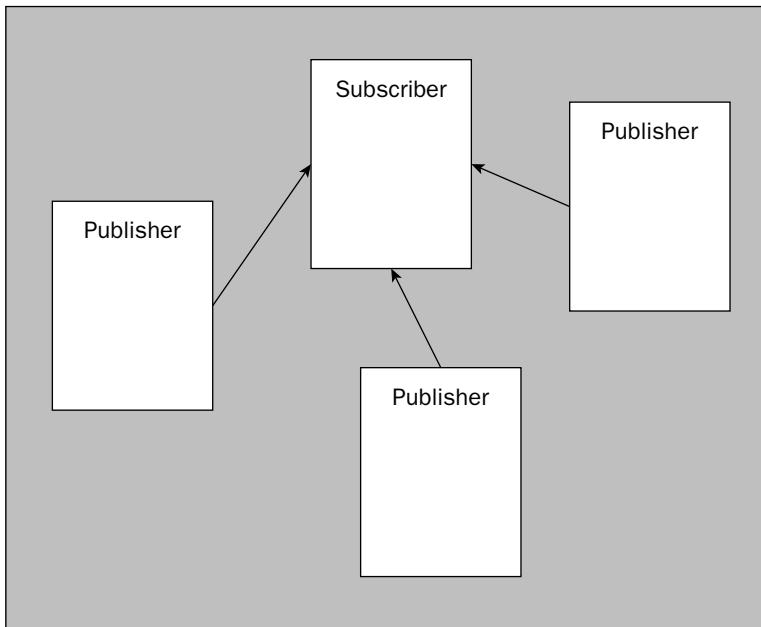


Figure 20-10

Mixed Models

Now let's look at a few variations based on the idea that we will frequently want to mix and match the basic models. Consider these as just a taste of the possibilities—something of “just the beginning.” The possibilities are almost endless.

Publishing Subscriber

Publishing subscribers (that is subscribers that are also configured as publishers) can be added to any of the basic models. This model has two publishers publishing the same data. The original publisher replicates data to its subscribers, one of which is a publishing subscriber. The publishing subscriber can then pass the same data along to its subscribers.

This model, shown in Figure 20-11, is useful when you have pockets of servers or when you have an especially slow or expensive link between servers. Another possibility is that you don't have a direct link between the initial publisher and all of the potential subscribers. The publisher only needs to pass data to one system on the far side of the link, and the publisher subscriber can then pass the data along to the other subscribers.

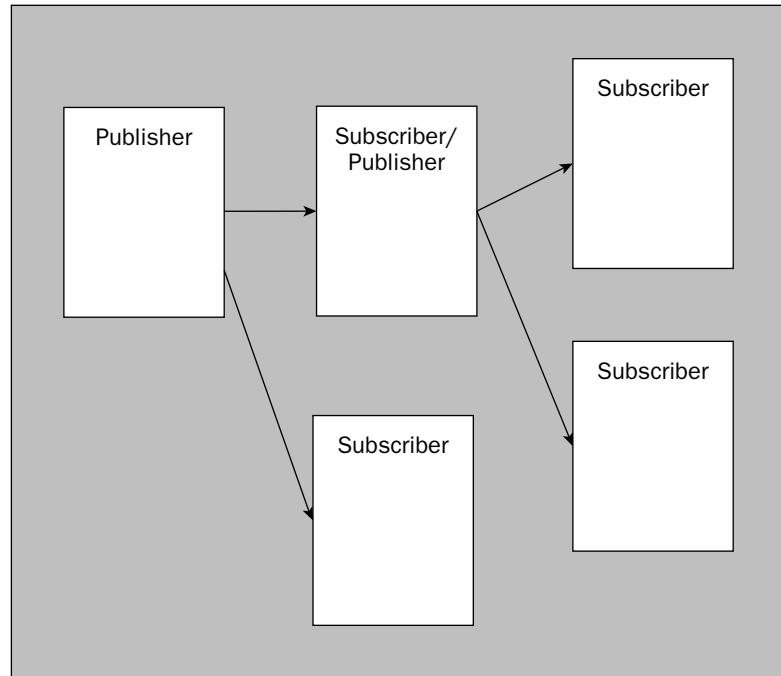


Figure 20-11

Publisher/Subscriber

This is another case where you have SQL Servers acting as both publishers and subscribers (Figure 20-12). Each server has its own set of data for which it is responsible. This model can be used when you have data changes taking place at both locations and you want to keep both servers updated. This is different from publishing subscribers in that each server is generating its own data, not just passing along updates received from another server.

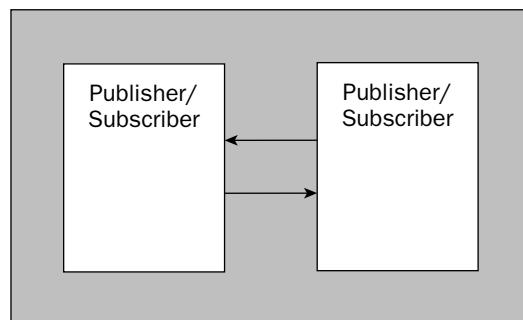


Figure 20-12

Multiple Subscribers/Multiple Publishers

Figure 20-13 shows one of the more complicated scenarios. Under this scenario, you have multiple publishers and multiple subscribers. Systems may or may not act as a publisher/subscriber or publishing subscriber. This model requires very careful planning to provide optimum communications and to ensure data consistency.

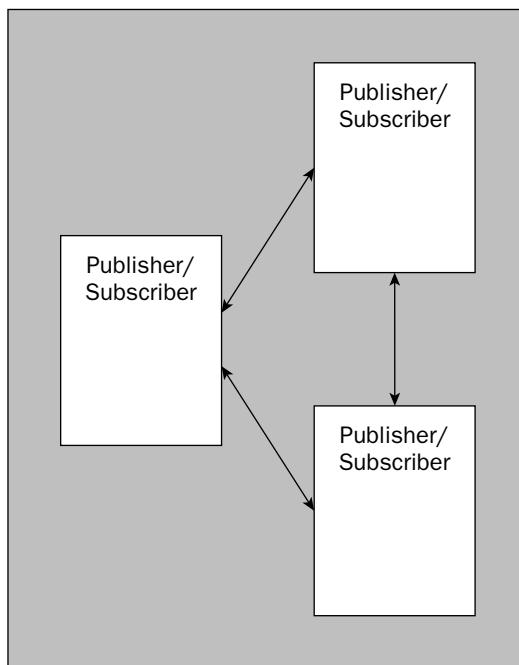


Figure 20-13

Self-Publishing

It is worth specifically calling out that you can have a server subscribe to its own published articles. This is actually fairly common in small installations, where there is a diverse need, but not necessarily enough load to justify more than one physical server. For example, you may want to segregate the data used for online transaction processing from the data used for decision making. You can use replication to make separate read-only copies of your data (updated on any schedule you consider appropriate) to be used as a reference.

Whether to locate your other databases—such as a data warehouse—on the same physical server as your core system is a matter of taste and your particular scenario. An example of where this can be very valid is the scenario where you have relatively low transactional volume but complex analysis needs. In my experience, companies that have enough need for a separate data warehouse usually have a physical or operational need for that to be on a separate server, but that is far from an “always” scenario.

Consider your particular situation—does your server have room to share the load? Can you risk both databases being offline at the same time in the event of a catastrophe?

Planning for Replication

Replication is one of those things where it can be easy to “just toss something together.” It’s also one of those things where it is easy to create a huge mess if you take such a cavalier approach. Keep in mind that SQL Server may automatically make some alterations to your schema to implement replication—do you really want SQL Server adding columns and objects to your database without fully thinking about that first? Of course not.

Any replication installation worth doing is worth taking the time to plan out. Some planning considerations include:

- What data is to be replicated
- Replication type
- Replication model

Along with these are other factors that will influence your decision, such as current network topologies, current server configurations, server growth potential, activity levels, and so forth. Each replication method has its advantages and disadvantages, and there is not a one-size-fits-all approach to replicating data. For instance, if you have a slow network or unreliable connection, then you may not want to implement transactional replication. Instead, you may opt to use merge replication that runs during a scheduled connection time. As has been pointed out repeatedly in this chapter, however, you need also need to balance that against consistency needs.

Data Concerns

First, you have to consider what you are going to publish and to whom. You need to identify your articles (tables and specific columns to be published) and how you plan to organize them into publications. In addition, there are some other data issues of which you need to be aware. Some of these have already been mentioned, but it’s worth our time to review them here.

timestamp

Include a `timestamp` column for transaction publications. That gives you a way of detecting conflicts on updates. By having a `timestamp` column already in place, you’ve already met part of the requirements for adding immediate-updating subscribers.

uniqueidentifier

A unique index and globally unique identifier is required for merge replication. Remember, if a published table doesn’t have a `uniqueidentifier` column, a globally unique identifier column will be added.

User-Defined Data Types

User-defined data types are not supported unless they exist on the subscriber destination database. Alternately, you can have user-defined data types converted to base data types during synchronization.

NOT FOR REPLICATION

The NOT FOR REPLICATION clause lets you disable table actions on subscribers. You can disable:

- The IDENTITY property
- CHECK constraints
- Triggers

These actions are essentially ignored when and only when the replication process changes data on the subscriber. Any other processes would still use them normally. So, for example, an insert into the original receiving database would have an identity value assigned, but as the row was subsequently published (in the form of an INSERT) to subscribers, the existing identity value would be used rather than generating a new value.

Mobile Devices

SQL Server also comes in a “Mobile” version (in SQL Server 2000, this was called SQL Server CE). This is an extremely small footprint version of SQL Server designed to run on Windows Mobile Edition (formerly Windows CE). The Mobile edition supports replication from a subscriber point of view. Snapshot and merge replication is supported — transactional replication is not.

Many of the considerations for mobile devices are just variants of the same theme that we’ve seen already in replication — bandwidth and space for example. Just keep in mind that the constraints for mobile devices may be much more extreme than with a full server class system (or even your salesmen’s laptops for that matter).

Setting Up Replication in Management Studio

Setting up replication takes a few steps. In particular, you need to:

- Configure your publication and distribution server(s) to be ready to perform those tasks
- Configure your actual publications
- Configure subscribers

Let’s take a look at how to do each of these within the Management Studio.

Configuring the Server for Replication

Before you can set up any publication or distribution on your server, your server must be configured for replication.

To get at this in Management Studio, navigate to the Replication node, right-click and select Configure Distribution.

SQL Server greets you with the standard intro dialog—in this case, it points out some of the options you will have as you go through this wizard. Click Next, and you are moved on to a dialog (shown in Figure 20-14) that decides if this publisher is to serve as its own distributor or if it should utilize an existing distributor:



Figure 20-14

If we select the option to use a different server as the distributor and choose “Add...”, then we would get a standard connection dialog box (asking for login security information for the distribution server). For our example run through here, keep the default option (that this box will act as its own distributor) and click Next.

Note that which dialog comes after the Distributor dialog will change depending on whether you have the SQL Server Agent configured to start automatically on system startup or not.

If you do not have the SQL Server Agent configured to start automatically (although you almost certainly want it to be on a production server), SQL Server will pop up a dialog, shown in Figure 20-15, to ask you about this (it will skip this next dialog if your agent is already configured to start automatically when you start your system).

Feel free to leave your system configured however you already have it (SQL Server will, however, default this dialog to changing your SQL Server Agent service to start automatically), but keep in mind that the agent will need to be running for some forms of replication to work.



Figure 20-15

Click Next. We move on to configuring a snapshot folder as shown in Figure 20-16. This will default to a directory in your main SQL Server folder, which for many installations may not be large enough to hold snapshots of large databases. This can be configured as a local volume or as a UNC path. Since I'm not going to assume you have a full server farm to try this stuff out on, we're going to take a "one server does everything" approach for this example, so accepting the default should be fine.



Figure 20-16

From there, it's on to configuring the actual distribution database. SQL Server gives a dialog to get some typical database creation information (what do you want to call it and where to store it), as shown in Figure 20-17.



Figure 20-17

From here, we move on to what, at first, appears to be a rather boring dialog (shown in Figure 20-18) with seemingly nothing new.



Figure 20-18

Chapter 20

Looks can, however, be deceiving. If we click on the little ellipsis (...) on the right, we get yet another dialog (shown in Figure 20-19)—one that does have a key item of note.

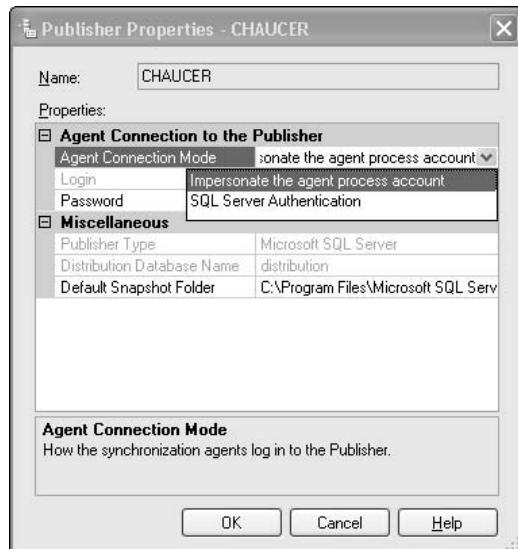


Figure 20-19

As Figure 20-19 shows, we have the ability to specifically set the connection mode we're going to use when connecting the agent to the publisher. In most cases, the default of impersonating the Agent process will be fine, but keep in mind that we can use specific SQL Server security credentials if need be.

Cancel out of this properties dialog, and click Next back in the publishers dialog (the one in Figure 20-18). Figure 20-20 shows the confirmation dialog, with what we want to do, at the end of the wizard. Note how it provides not only the option of immediately configuring the distribution, but also the concept of scripting the configuration for later or potentially remote use.

Go ahead and click Finish (the next dialog is just a summary, so there is no need to dwell there). SQL Server begins processing the configuration request. When the process is complete, go ahead and close the dialog.

And, just that quick, you have a server configured for publication and distribution of replicated data. Obviously, were this a production environment, we might have some other choices to make in terms of specific locations or even whether we wanted the publisher and distributor to be on the same system or not, but the basic foundations of what we are doing remains the same regardless.

If you wonder about the distribution database, you should now be able to find it under the "System Databases" subfolder of the Databases folder.

Configuring a Publication

With our server all nice and configured, we're ready to get down to creating an actual publication.

To do this, navigate to the Replication node in Management Studio, right-click the Local Publications sub-node, and choose New Publication....



Figure 20-20

After the usual intro dialog, we come to the Publication Database dialog shown in Figure 20-21. This allows us to choose what database we want to utilize for our publication. As you can see, I've selected our old friend, AdventureWorks.

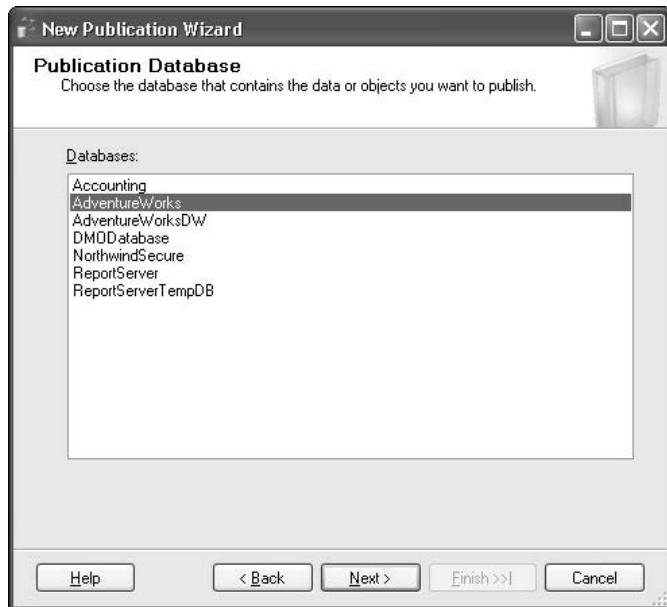


Figure 20-21

Chapter 20

Click Next, and you're ready to move on to the Publication Type dialog shown in Figure 20-22.

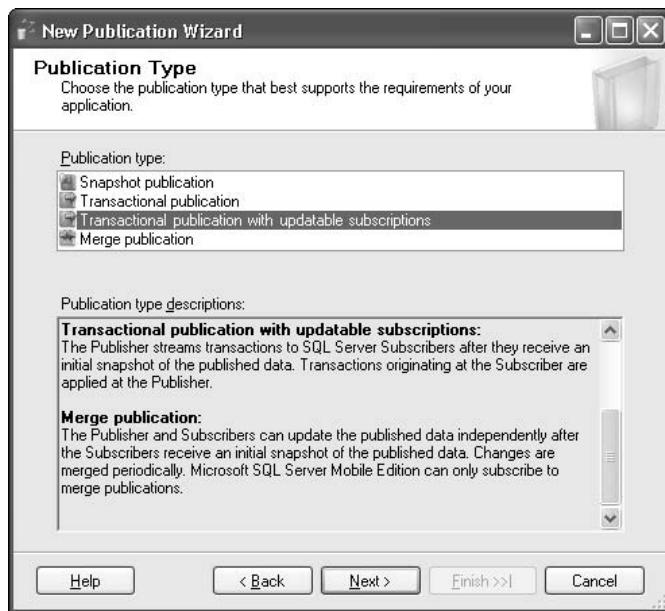


Figure 20-22

This allows us to select between the replication types that we looked at earlier in the chapter. I've chosen Transactional publication with updatable subscriptions.

Click Next, and you move on to the Articles dialog.

In Figure 20-23, I've expanded the Tables node and selected the Person.Contact table. I'm taking most of that table, but I'm going to skip the AdditionalContactInfo column since it is a schema-bound XML column, and SQL Server does not allow for the replication of XML columns that are bound to an XML schema collection. I also could have taken other schema objects such as stored procedures (I'm sticking to just the one object for simplicity sake).

Click Next to be taken to the Article Issues table, as shown in Figure 20-24.

Notice that SQL Server detected several issues it wants to let us know about. This is one where I say "kudos to the SQL Server team" for attempting to let a user know about some fundamental things *before* they become a problem.

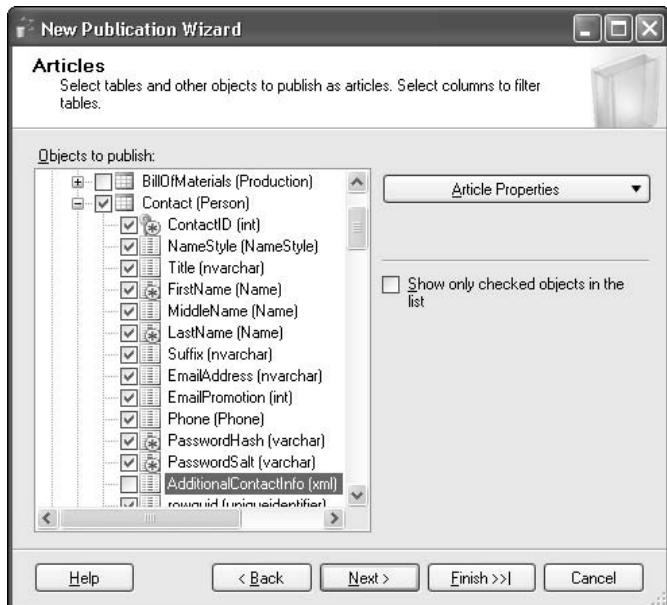


Figure 20-23



Figure 20-24

Chapter 20

Click Next to move on to the Filter Table Rows dialog shown in Figure 20-25.



Figure 20-25

This one allows us to do horizontal partitioning — essentially just applying a WHERE clause so that only rows that meet a specific condition will go across in our publication.

Click Add to get the dialog shown in Figure 20-26.

In our example here, we've restricted the rows being replicated to those where we have a phone number available.

Click OK to return to the Filter Table Rows dialog, and then click Next to move on to the Snapshot Agent dialog shown in Figure 20-27.

Remember that any subscription, regardless of whether it is to a snapshot, merge, or transactional replication model, must start by synchronizing based on a snapshot. Subsequent changes are begun relative to that snapshot.

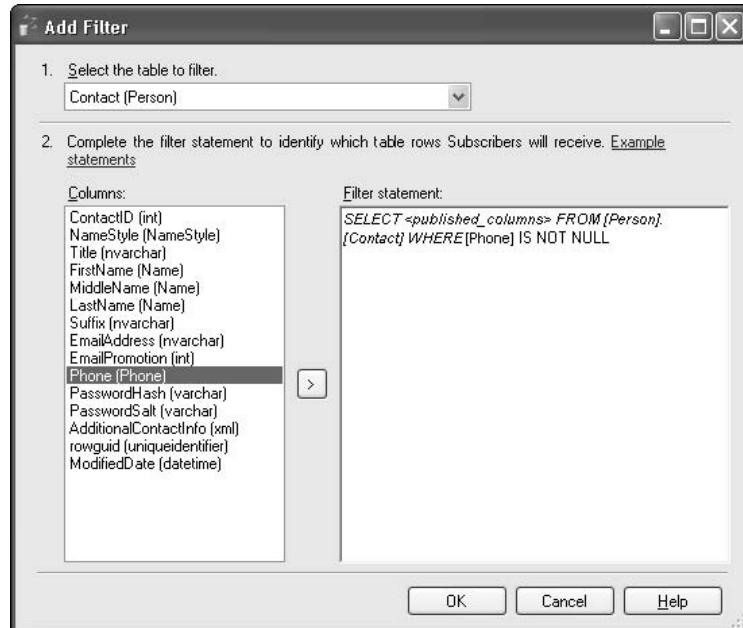


Figure 20-26

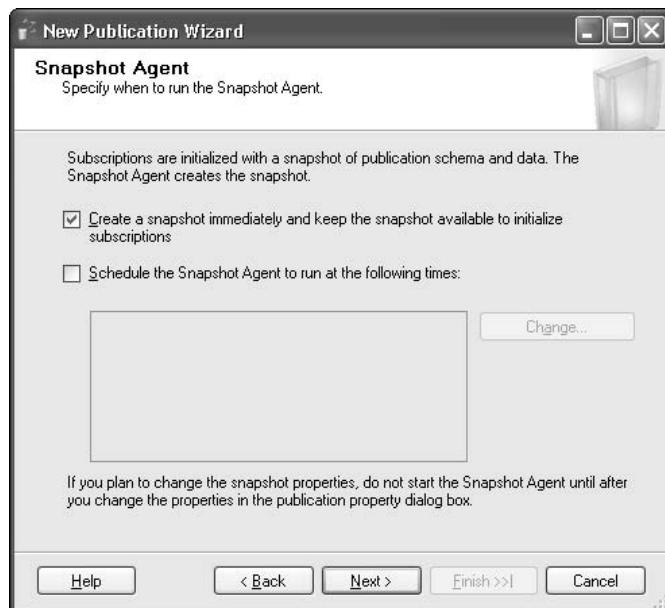


Figure 20-27

Chapter 20

I've configured mine to run the snapshot immediately, but I could have just as easily scheduled it to be generated at a later time (remember that snapshots place locks on every table the snapshot utilizes—do not run them at a time where such lock issues are going to block writes to your database that you need done in a timely fashion). If, for example, you are getting frequent new subscribers, you may want to schedule a periodic update to the snapshot to give them a more up-to-date time to synchronize to.

Click Next, and you're ready to define the Agent Security, as shown in Figure 20-28.

I've used the Security Settings dialogs to set the agents to use the SQL Server Agent account. This is not, however, good practice in a production environment for security reasons. Give the agents their own account to impersonate to both limit agent access and increase your ability to audit.



Figure 20-28

Click Next, and you'll find an Action dialog (just like the one back in Figure 20-20) where you can indicate whether you want the publication created immediately or scheduled for later execution.

One more click of the Next button, and you're ready for a summary and to define a publication name as shown in Figure 20-29.

Go ahead and click Finish to create your publication, and, just like that, you're ready to have subscribers!

Setting Up Subscribers (via Management Studio)

Setting up subscribers utilizes the same basic notions we've already leveraged with publications. Before we get started with an example, however, let's set up a dummy database to play the part of our subscriber.

```
CREATE DATABASE AWSubscriber
```



Figure 20-29

And, with that created, we're ready to subscribe to some data.

Start by right-clicking the Local Subscriptions sub-node below the Replication node in Management Studio, and selecting New Subscription. After the usual intro dialog, we move on to identifying our publication, as shown in Figure 20-30. Since we have only one publication, there really isn't a lot to choose from, but the list could have easily been many, many publications.



Figure 20-30

Chapter 20

Click Next to move on to the Agent location, as shown in Figure 20-31. Remember that we can run our replication agent on either the subscriber or the distributor. In our case, it doesn't matter much since these are the same box, but you may make different choices depending on server loading issues.

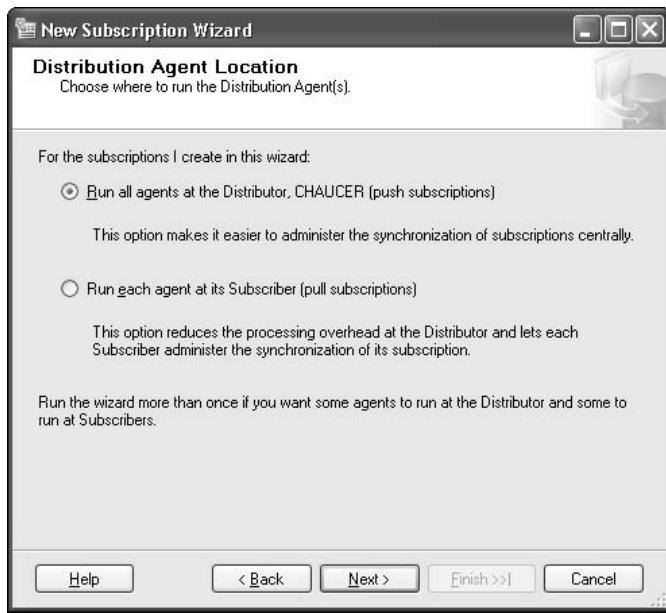


Figure 20-31

Click Next to move on to the Subscribers dialog shown in Figure 20-32. I've already chosen our AWSSubscriber database, but notice how we could choose Add SQL Server Subscriber and configure multiple subscribers at one time.

From there it's on to the Distribution Agent Security dialog. Here we define what security context we want to run under for both the distributor and subscriber (in this case, it's the same system, but it could have easily been remote). In Figure 20-33 I've chosen to impersonate the SQL Server Agent security context, but, again, on a production server you would generally want a more specific security context for your replication agent for security reasons.

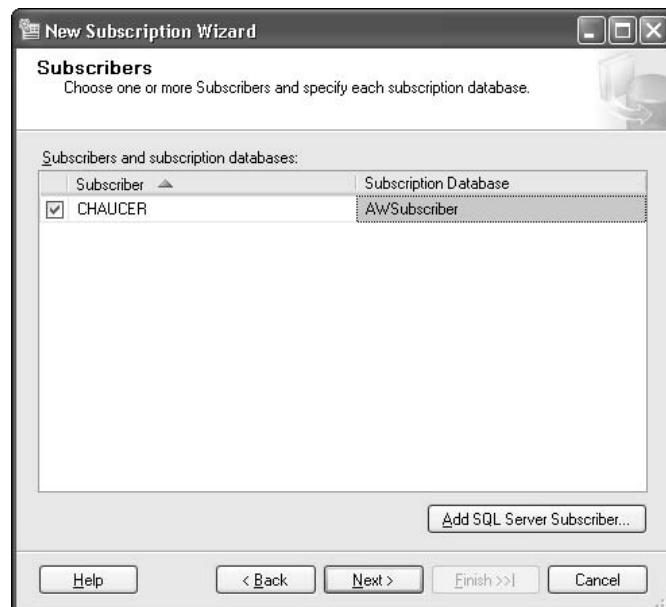


Figure 20-32



Figure 20-33

We can move quickly through the remaining dialogs by setting the agent to “Run continuously” and leaving the default “Commit at publisher” setting of “Simultaneously commit changes.” That takes us to the Login for Updatable Subscriptions dialog shown in Figure 20-34.

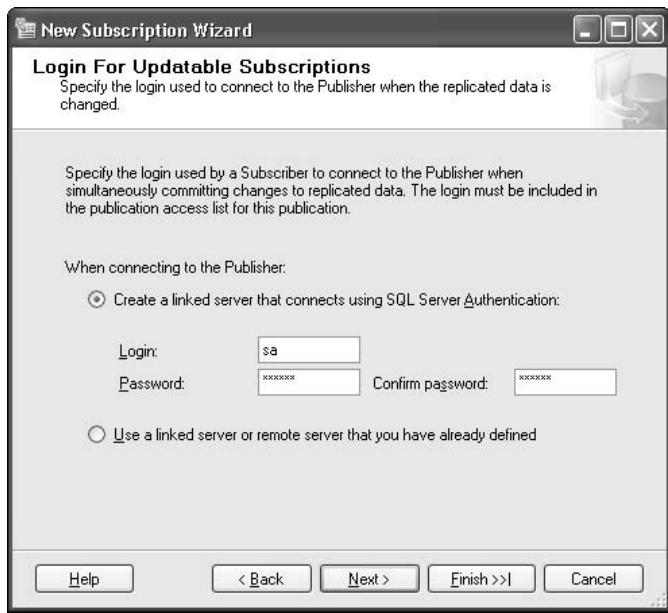


Figure 20-34

Since this is all (distribution and subscription) happening on the same server, a linked server is implied (a server is always available to itself as a linked server). Were we using a remote distributor, we could have either used a regular SQL Server login or again went with a linked server (though, in the later case, we would need to configure the linked server separately).

A linked server is another SQL Server or ODBC data source that has had an alias established for it on your server. When you refer to a linked server by name, you are essentially grabbing a reference to connection information to that linked server.

Figure 20-35 allows us to choose when to initialize our subscription (I've stayed with the default of immediately). The initialization involves pulling down the snapshot from the distributor and applying it. Subsequent synchronizations will be done using the snapshot as a baseline to apply changes to.

Click Next to get the same finishing dialogs that we've seen in prior examples (when to run things and a summary page), and then click Finish.

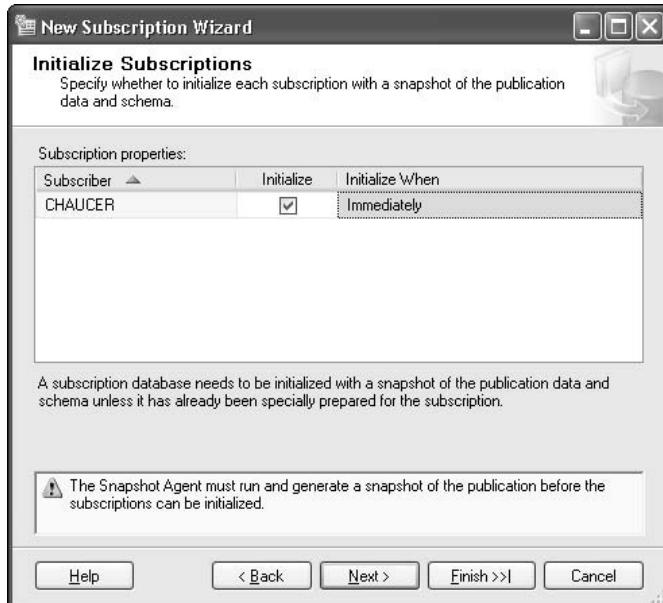


Figure 20-35

Using Our Replicated Database

Once the replicated database is in place, the problem largely becomes one of administration. If things are running smoothly, there is very little to see. Users can access our AWSubscriber database and the Person.Contact table within it. Since we configured for updating subscribers, changes made to the AWSubscriber version of the Person.Contact table will be immediately reflected in the source AdventureWorks database. Likewise, changes made to our AdventureWorks database will be reflected in our subscriber database.

You can start by taking a look in the AWSubscriber table list. As you can see in Figure 20-36, the replication agent has added two tables for us. The first is easy enough to recognize—it is the Person.Contact table that we requested be replicated. The second is a conflict tracking table—it should receive data only in the event that changes we make in our subscriber run into a conflict with those on the publisher.

In the event of a conflict, the default publishing agent chooses the publisher's data over the client's. You can change this to prefer things based on such things as which is the most recent change and other ready-made criteria. You can also write custom resolution algorithms to encompass any unusual rules you may have for resolving conflicts.

Chapter 20

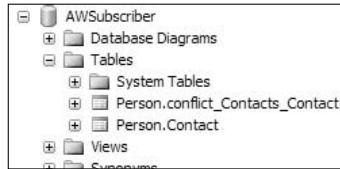


Figure 20-36

Now let's test out our transaction-based replication by making a change to our data. We'll start by taking a look at the starting value of the row we're going to change:

```
SELECT aw.FirstName AS PubFirst,
       aw.LastName AS PubLast,
       aws.FirstName AS SubFirst,
       aws.LastName AS SubLast
  FROM AdventureWorks.Person.Contact aw
 JOIN AWSubscriber.Person.Contact aws
    ON aw.ContactID = aws.ContactID
 WHERE aw.ContactID = 18
```

What I've done here is join across the databases so that we can see both the publisher and subscriber at the same time. This way, we can, in one query, compare the source and the destination. The first time we run this script (before we make any changes), we can see our starting values, and that they are indeed the same:

PubFirst	PubLast	SubFirst	SubLast
Anna	Albright	Anna	Albright

(1 row(s) affected)

Okay, now let's make a change. We'll say that Anna has gotten married and decided to change her name to Albright-Smith.

```
USE AdventureWorks

UPDATE Person.Contact
SET LastName = 'Albright-Smith'
WHERE ContactID = 18
```

Now, we run our original SELECT statement again to check the results:

PubFirst	PubLast	SubFirst	SubLast
Anna	Albright-Smith	Anna	Albright-Smith

(1 row(s) affected)

As you can see, both the publisher and subscriber received the update.

Now, let's change the script just slightly to run inside the subscriber database, and see what happens on the publisher's side. This time, we'll change Anna's name back (perhaps she changed her mind . . .):

```
USE AWSubscriber

UPDATE Person.Contact
SET LastName = 'Albright'
WHERE ContactID = 18
```

And now we're ready to run our original select statement one more time:

PubFirst	PubLast	SubFirst	SubLast
-----	-----	-----	-----
Anna	Albright	Anna	Albright

(1 row(s) affected)

Again, our change was seen in both databases.

The change was seen going both directions and was replicated immediately because we had selected transactional replication with immediately updating subscribers. Other replication choices would have introduced latency in the change, or potentially not replicated the change at all without some form of manual intervention. Be sure to review all of the replication types (discussed earlier in the chapter) to understand the behavior of each.

Replication Management Objects (RMO)

Replication Management Objects, or RMO, is a new .NET object model that replaces the replication portion of the COM-based Distributed Management Objects (DMO) object model that was used in SQL Server 2000 and earlier. You can think of RMO as being something of a companion to SQL Management Objects (SMO), which we discuss extensively in Chapter 25.

RMO gives you programmatic access to any portion of your replication creation and configuration using any .NET language. Examples of RMO use would be automating operations such as:

- Creating and configuring a publication** — You can make use of the `ReplicationDatabase` as well as the `TransPublication` or `MergePublication` objects to define publications.
- Adding and removing articles** — The `TransArticle` object supports the addition and removal of articles within your publication. In addition, you can add column filters or add a `FilterClause` property to limit what rows are replicated.
- Republishing your snapshot**

These are just some more everyday use kinds of examples. RMO is, however, capable of creating, modifying or deleting any part of the replication process.

RMO can be utilized in Visual Studio by adding a reference to the *Microsoft.SqlServer.Replication .NET Programming Interface* library. You then point your include, imports, or using directives to `Microsoft.SqlServer.Replication`. As with any of the management libraries that support SQL Server, you will also need to have a reference to the `Microsoft.SqlServer.ConnectionInfo` library.

An example application that utilizes RMO to create the same publication we created earlier in the chapter using the GUI can be downloaded from the Wrox Web site or professionalsql.com.

Summary

As much as there was to take in this chapter, this really was something of an introduction to replication. We covered a lot of the considerations for architects reasonably well, but the scope of replication is such that entire books are written on just that topic. Indeed, there is much to consider in order to build just the right model for complex scenarios. The good news is that, if you really grasped this chapter, then you are prepared for perhaps 90 percent of what you are likely to ever face. Time and the proverbial “school of hard knocks” will teach you the rest.

If you’re taken anything from this chapter, I hope that it’s an understanding of some of the general problems that replication can solve and how replication works best when you plan ahead both in terms of topology planning and in your applications general architecture (making sure it understands the special needs of replication).

In our next chapter, we’ll take a look at yet another “extension” area for SQL Server—full-text indexing.

21

Looking at Things in Full: Full-Text Search

Using plain old T-SQL (without full-text functionality), our options for querying text information are somewhat limited. Indeed, we have only a couple of options:

- Use a LIKE clause. This is generally woefully inefficient, and is not able to utilize any kind of index structure unless your search pattern starts with an explicit value. If the search starts with a wildcard (say "%" or "_"), then SQL Server wouldn't know which spot in the index to begin with—any indexes become worthless.
- Use some other form of pattern matching, such as PATINDEX or CHARINDEX. This is generally even more inefficient, but this can allow us to do things that LIKE will not.

With Full-Text Search, however, we gain the ability to index the contents of the text—essentially keeping a word list that lets us know what words we can find and in what rows. In addition, we are not limited to just pattern-matching algorithms—we can search for the inflected forms of words. For example, we might use the word *university* but have SQL Server still find the word *universities*, or, even better, SQL Server can find a word like *drunk* when the word we asked for was *drink*. It's up to us to decide how precise we want to be, but even if the word we are searching for is located deep in the middle of a large text block, SQL Server can quickly find the rows that contain the word in question.

Full-Text Search, or FTS, supports any document type that is supported by Microsoft Index Server—this means that you can store things like Word, Excel, Acrobat, and other supported files in an image data type, but still perform full-text searches against that data! Indeed, the format for building Index Server plug-ins to support new document types is a published API, so you could even write your own extensions to support other document types if necessary.

Personally, I find this later point to be extremely cool. As we saw back in Chapter 16, we're now living in an XML world. We can use OPENXML to query XML on the fly, but there isn't currently any way to index the contents. Since Index Server file extensions are now supported, you could, for example, build an extension that knows the format of your XML document, store the document in an image data type using the extension, and, whammo, you can now use Full-Text Search to be able to quickly search a large store of XML documents for those that contain certain data. It's not an ideal query model by any means—searching for the existence of a text string in an XML document rather than querying individual columns as we can in a relational data store—still, it has some slick possibilities to it.

In this chapter, we'll take a look at all of these Full-Text Search features. Full-Text is something of a different animal from the kinds of things that we've seen in SQL Server thus far—living partly in SQL Server itself and partly as an autonomous unit that leverages non-SQL Server technologies. We'll examine some of the issues that are unique to full-text indexing and search, and explore the special syntax that is used in Full-Text queries.

Among the sections we'll look at are:

- Full-Text Search architecture
- Setting Up Full-Text indexes and catalogs
- Full-Text query syntax
- Full-Text quirks
- Noise words

In addition, we'll see how there are now two ways of completing most Full-Text-related operations. By the time we're done, you should be prepared for the hassles that FTS creates for you, but you should also be ready to utilize what can be some wonderful functionality in return.

Full-Text Search Architecture

The architecture surrounding FTS is something that confuses a lot of people. When you realize how the different pieces play together to make FTS happen, the confusion isn't that surprising.

The first thing to recognize is that the core of Full-Text Search isn't really part of SQL Server at all. Actually, it is a shared technology item that originally comes from Microsoft Index Server. The technology was originally implemented through a service known as MSSearch, and is now in its own service that is controlled by the SQL Server team (same original code base by my understanding though). This new service, called SQL Server FullText Search (it will have an instance name after it in the services applet if it is not part of the default SQL Server instance), is excellent at examining raw text data and aggregating word lists. By separating SQL Server's implementation of Full-Text Search from the O/S Indexing Service, Microsoft was able to increase the efficiency of how the full-text service communicates with the core database engine. The SQL Server FullText Service—or FTS—maintains an association between the individual words and phrases and the places that the FTS has encountered them.

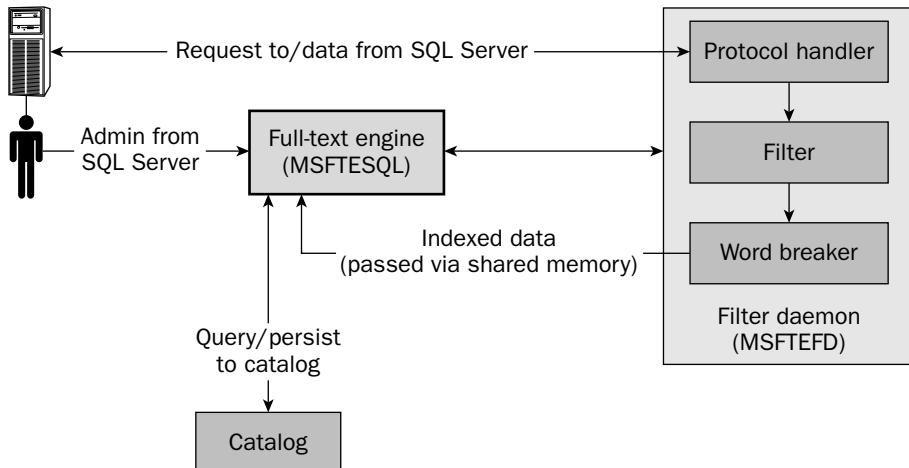


Figure 21-1

In order to perform full-text queries against any SQL Server table, you must build a full-text index for that table. The construction and maintenance of this full-text index—or the population of the index—is done through a process of SQL Server passing text streams to the Full-Text engine, the words in the stream being catalogued, and an association being made between the catalog entry and the row the word was sourced from.

By default, the full-text engine is installed using the “manual” startup option for services—that is, you need to start it if you want it to be running. In order to run any of the examples in this chapter, you will need to have the service running—I recommend changing the startup option to “Automatic” once you build your first full-text index on any server.

By default, tables have no full-text functionality at all. The fact that there is a table and that it has text data types is no guarantee that there is a full-text index on the table—if you want it, you need to create it. Even after you create the full-text index, the index will have nothing in it—in order to make the index fully functional, you need to *populate* the index.

The population process looks over the columns specified by the index and builds the word list that is going to be used. Much like standard indexes in SQL Server, only the columns you specify to include in the index will become part of the index. Unlike normal indexes in SQL Server, however, you are allowed only one index per table—so every column you want to have participate in full-text queries needs to be part of the index.

The differences don’t stop there though—actually there are several. The major differences include:

- ❑ **Storage Location**—SQL Server indexes are stored as part of the main database file(s). Full-text indexes, however, are stored in what is called a full-text catalog. This makes it very easy to store the full-text catalog on a separate physical device than the rest of the database (which can help performance).

- ❑ **Method of Creation**—SQL Server indexes are created using the `CREATE INDEX` command in T-SQL, SMO, or WMI (You can use the Management Studio, but it just uses SMO). Full-text indexes are created either through the use of special system stored procedures or through the use of the new `CREATE FULLTEXT INDEX` command.
- ❑ **Method of Update**—SQL Server indexes are automatically updated in the normal course of changes to the underlying SQL Server data. Full-text indexes can either be populated on demand or through a “change tracking” mechanism with an on-demand cleanup.

So that’s the quick lesson in Full-Text Architecture 101. As we move through the rest of the chapter, the impact of the differences should become apparent versus the more “normal” way things are implemented in SQL Server.

Setting Up Full-Text Indexes and Catalogs

As we saw in the last section, each table in a SQL Server table can have zero or one full-text indexes. These full-text indexes are stored in a file—called a *full-text catalog*—that is external to the SQL Server database and system files. A catalog can store multiple full-text indexes. The indexes must be from the same database; you may, however, want to store indexes from one database in multiple catalogs, so you can manage the population of those indexes on separate schedules.

Enabling Full-Text for Your Database

For SQL Server 2005, full-text indexing is turned on by default for every database you create. In previous versions, however, SQL Server would not allow full text catalogs or indexes to be created until you enabled full-text search for the database in question. Keep this in mind if you’re working with an earlier version of SQL Server.

To check whether full-text indexing is on or off, you make use of the `DATABASEPROPERTYEX` system function. Just write a simple query to call the function:

```
SELECT DATABASEPROPERTYEX('AdventureWorks', 'IsFulltextEnabled')
```

This returns a 1 if full text is enabled and a 0 if it is not. (See Appendix A for more information on what `DATABASEPROPERTYEX` can obtain information on.)

In the event that you do need to enable (or disable for that matter) full-text indexing on your database, you make use of a special system stored procedure called `sp_fulltext_database`. The syntax is amazingly simple:

```
EXEC sp_fulltext_database [@action =] '{enable|disable}'
```

The `enable` option turns on full-text indexing for the current database. If there are any full-text catalogs already existing for the current database, they are deleted, as are any associations your tables have with those catalogs.

Really be careful of this later point. There is no prompting and no warning—anything to do with full-text in the current database is immediately and irrevocably destroyed to make way for a fresh start. This means you need to be careful in two ways: (1) be sure this is something you really want to do, and (2) make SURE, for positive, that the correct database is current when you execute this stored procedure (sproc).

The disable option turns off full-text operations for the current database and removes any current catalogs and indexes.

As a quick example, imagine for a moment that we wanted to turn off full-text indexing for the AdventureWorksDW sample database:

```
USE AdventureWorksDW  
  
EXEC sp_fulltext_database @action = 'disable'
```

And it's done.

Be warned that SQL Server doesn't really tell you much to confirm that full-text operations have been successfully enabled. All you'll see are a few messages about rows being affected—there isn't even a print statement that says "done enabling full-text search" or something like that. You should, however, get some form of an error if there is a problem.

Creating, Altering, Dropping, and Manipulating a Full-Text Catalog

With SQL Server 2005, a new syntax was added for creating, altering, dropping, and otherwise manipulating full-text catalogs. This probably should be the way that you do things for new development moving forward as long as you don't need backward compatibility with SQL Server 2000 and older. What's cool about the new syntax is that it uses language elements rather than system stored procedures. In addition, these new language elementals are far more compatible with the syntax used by other DBMS products in their full-text syntax.

Creating Full-Text Catalogs

Beginning with SQL Server 2005, we have a new syntax available for creating full text indexes.

The CREATE syntax looks much like other CREATE syntaxes, but with a few additional twists:

```
CREATE FULLTEXT CATALOG <catalog name>  
[ON FILEGROUP <filegroup> ]  
[IN PATH <'rootpath'>]  
[WITH ACCENT_SENSITIVITY = {ON|OFF}]  
[AS DEFAULT]  
[AUTHORIZATION <owner name> ]
```

Chapter 21

Most of this should be fairly self explanatory, but let's take a look anyway:

ON FILEGROUP

When a full-text catalog is created, SQL Server creates some internal structures (system objects) to support it. Using `ON FILEGROUP` tells SQL Server which file group you want those internal structures stored with. For the vast, vast majority of installations, you're going to want to stick with the primary file group (which is default). So, in short, now that I've told you about this option, forget about it.

IN PATH

The actual full-text catalogs are not created inside the database but rather as a separate file on disk. This option tells SQL Server what path you want that file created in. If you do not specify a path, then SQL Server will create the full-text catalog in a default directory that was set at the time you installed your SQL Server.

This is one of those that I recommend you at least fully think about each time you create a full-text catalog. Remember that, wherever this is stored, it is potentially competing with other files for I/O operations. Consider the location your full-text catalog is being created in and whether you need to move the related I/O to a different physical path.

WITH ACCENT_SENSITIVITY

Pretty much what it sounds like. This determines whether searches will take into account accents or not (for example, is "e" the same as "é"). Keep in mind that, if you change this setting after the catalog is created, the entire catalog will need to be repopulated.

AS DEFAULT

Another one that is what it sounds like; this one sets the full-text catalog you're creating to be the default catalog for any new full-text indexes you create.

AUTHORIZATION

Mildly more complex. As you might imagine, this one is about security and rights. It changes the ownership of the full-text catalog to be the user or role specified instead of the default (which would be the user that actually creates the catalog). This one has gotten muddled quite a bit by SQL Server's change from ownership to schemas. Ownership has largely morphed into schemas, but the nature of this particular setting more closely fits with the older ownership notion. The key thing to realize here is that a role can be the owner of a full-text catalog—not just a user. If you're changing the ownership to a specific role, then the user creating the full-text catalog must be a member of that role at the time that he or she creates the catalog.

So, let's create a full-text catalog for AdventureWorks. We'll simply call it:

```
USE AdventureWorks

CREATE FULLTEXT CATALOG MainCatalog
    IN PATH 'C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\FTData'
```

This is another one of those commands you issue where you don't get much feedback—as long as you don't see an error, the catalog should be created just fine.

In this example, I've specified a specific path—this is the option you are most likely to use. As it happens, I specified a path that was the default anyway, but I could have specified any valid directory (the directory must, however, already exist). I did not specify this to be the default catalog—this means any full-text indexes created will need to explicitly state that they want to utilize this catalog.

Altering Full-Text Catalogs

Altering full-text indexes works pretty much the same as creating them, save for the fact that you are really limited in what can be altered. The syntax is:

```
ALTER FULLTEXT CATALOG catalog_name
{ REBUILD [WITH ACCENT_SENSITIVITY = {ON|OFF} ]
    | REORGANIZE
    | AS DEFAULT
}
```

There are three top-level options you can set with this ALTER. Let's take a look at them:

REBUILD

Does what it says it does—completely rebuilds the full-text catalog in question. The actual physical file in the O/S is deleted and recreated from scratch. By default, it will be created with exactly the same settings the catalog had before. Only the physical file is rebuilt (the metadata is left as is).

Keep in mind that your full-text catalog, and every index that catalog contains, will be offline while the rebuild is in progress

In addition to the simple rebuild that you would typically do just to compact the file (for deleted rows and such), you can also rebuild in order to change the accent sensitivity. If you want to reset the accent sensitivity, just specify whether you want it on or off as you issue the REBUILD command.

REORGANIZE

This is similar to REBUILD, but with some pros and cons.

REORGANIZE cleans up your file for you, but in an online fashion. The result is like most situations where you rearrange things instead of moving things all the way out and starting over—it looks pretty good, but perhaps not as good as if you had started from scratch.

You can think of REORGANIZE as being like a defragmentation process. It merges what may well be several different index structures internal to the file (for performance reasons at the time the full-text was analyzed, some items may have been kept in their own substructure in the index rather than merged into

Chapter 21

the master index for the catalog). This command attempts to rectify that. Unlike REBUILD, REORGANIZE does also reorganize the internal structures for your full-text catalog (the ones that store metadata).

AS DEFAULT

This works just like it did under CREATE. It establishes this particular catalog as being the default full-text catalog for new full-text indexes you create for this database.

Dropping Full-Text Catalogs

I know you can see this one coming — after all, it's that same core DROP syntax we've been using all along:

```
DROP FULLTEXT CATALOG <catalog name>
```

And, of course, it's gone (including the underlying file in the O/S).

Creating, Altering, Dropping, and Manipulating Full-Text Indexes

Okay, so what we had with a full-text catalog was largely just a container. A full-text catalog, by itself, is nothing at all. What we need are the actual full-text indexes. Whereas a full-text catalog is the place to store full-text indexes, the indexes themselves are what provide the actual reference information that allows your full-text queries to operate quickly and efficiently.

Creating Full-Text Indexes

When you go to create a full-text index, the core items of the command are not all that different from regular indexes; however, much as regular indexes have properties such as whether they are clustered or non-clustered, full-text indexes also have their own properties.

The syntax for creating a full-text index looks like this:

```
CREATE FULLTEXT INDEX ON <table name>
    [ ( <column name> [TYPE COLUMN <type column name> ]
        [LANGUAGE <language term>] [,....n]) ]
    KEY INDEX index_name
        [ON <fulltext catalog name> ]
    [WITH
        { CHANGE_TRACKING { MANUAL | AUTO | OFF }
        [, NO POPULATION] }
    ]
```

Note that what is optional is a bit atypical here. Most of the time, required items are listed first, but the quirks of this syntax give us an optional parameter (a column list) before a required parameter (the key index). Let's start with a quick example and then take a look at the parts.

```
CREATE FULLTEXT INDEX ON Production.ProductModel
    ( Name LANGUAGE English)
    KEY INDEX PK_ProductModel_ProductModelID
        ON MainCatalog
    WITH CHANGE_TRACKING OFF, NO POPULATION
```

So, what we've created here is a full-text index for the `Production.ProductModel` table. We've explicitly stated that the language used in that column is U.S. English. If we had wanted, we could have added a comma followed by another column name and potentially a `TYPE COLUMN` or another `LANGUAGE` identifier. After the language, we specifically stated what full-text catalog we wanted this index stored in as well as that we wanted change tracking turned off and no initial population of the index.

That's a lot to think about, so let's take a look at those parts a bit closer.

Notice that I did not supply a name for my full-text index. There can only be one full-text index for any given table, so there is no need to name it (it is essentially identified by the table it's built on). Be sure that you have includes *all* the columns you want to perform full text searches on.

Column List

This is probably the trickiest part of the whole thing. Even though it says "column name" in the preceding syntax, you're really working on a column *list*. The issue is that for each column you list you need to include everything about that column before you move on to the next column. That is, you need to include the `TYPE COLUMN` and `LANGUAGE` parameters (if you're going to) *before* you name the next column.

So, for example, if we had also wanted to include the catalog description, we could have done that, too, by adding it at the end of the first column definition:

```
CREATE FULLTEXT INDEX ON Production.ProductModel
    ( Name LANGUAGE English,
      CatalogDescription)
    KEY INDEX PK_ProductModel_ProductModelID
        ON MainCatalog
    WITH CHANGE_TRACKING OFF, NO POPULATION
```

LANGUAGE

This specifies what language the column we've just identified is in. This is important for determination of noise words (words that occur frequently but add little to your search — we'll see more about these later in this chapter), as well as things like collation. Any language that SQL Server has localization support for (33 localizations as of this writing) is valid. To get a list of the aliases you would use, you can query the `syslanguages` table in the master database:

```
SELECT name, alias FROM master..syslanguages
```

TYPE COLUMN

AdventureWorks isn't giving me a good example for use in this book (audible sigh here!), but this option is for use when you want to do full-text indexing against documents stored in an image or a varbinary column.

Imagine for a moment that you're doing document management using SQL Server (not at all an uncommon use for SQL Server). If you are storing documents written in a mix of one or more applications, such

Chapter 21

as Microsoft Word (.doc), Acrobat (.PDF), Excel (.xls), or a text editor (.txt), then full-text search will need to know what kind of document is stored for each row it analyzes, so it knows what analysis plug-in to use.

In this case, you need to add another column to your table (in addition to the image or varbinary column) that contains the extension (.DOC, .PDF, etc) of the document stored in the binary column.

KEY INDEX

Unlike all the other options in the `CREATE FULLTEXT INDEX` command, this one is required.

Any table that full-text indexing *must* have a column that uniquely identifies each row — this can be a primary key or a unique constraint. The thing to remember on this point is that you are supplying the name of the *index* associated with the unique identifier, *not* the column or constraint name.

ON

This is simply the name of the full-text catalog you want this index stored in. This is optional if your database has a default full-text catalog, and required if no default catalog is has been established.

WITH

This supplies instructions regarding how your index is populated with data and how it copes with changes to the table that the index is built over.

CHANGE_TRACKING

Change tracking is all about how your full-text index deals with changes to the underlying table.

Unlike regular indexes, which are fundamental to the table and are always kept up to date as data changes, full-text indexes are stored in a separate file and maintained by a different service. This creates a dilemma regarding how accurate you want your full-text searches to be versus how much overhead you want to endure on keeping what is essentially an external system up to date.

Change tracking gives us three levels of support for changes:

OFF	<p>The full-text index is updated only when you perform a full population of the index. Essentially, you need to rebuild from scratch each time you populate. This means there is no ongoing maintenance overhead, but it also means that there may be rows in the table that will not be returned in your full-text queries or, perhaps worse, that rows may come back as containing the word you are interested in when, due to changes, they no longer do.</p> <p>This option is great when your data is slow moving (doesn't change often) and/or accuracy doesn't need to be perfect. In return for giving up that accuracy, it means you have no ongoing overhead and that your indexes are always as compact as they can be because they have no issues with fragmentation. It does mean, however, that when you do repopulate, you have a period of downtime and the overall process takes longer.</p>
-----	--

AUTO	<p>Under this model, SQL Server is constantly updating the index for things happening in the table. While there still may be a lag between when the change is made and when it is reflected in full text, that lag is minimal and you are getting something approximating real-time updates.</p> <p>This is the way to go when you have fast-moving data or your need for accuracy is very high. You are enduring a high degree of overhead since SQL Server will use smaller, intermediate structures to keep track of the changes. These can become inefficient over time and may hurt search performance but are not that big of a deal in the short run. If you use this option, consider still performing a reorganization or full repopulation regularly.</p>
MANUAL	<p>This is something of a middle ground, and in the old syntax would equate to what was called “incremental.”</p> <p>This one does tracking to be able to identify changes but does not update the full-text index immediately. You can then manually perform updates that apply the changes to the existing index without a full repopulation.</p>

NO POPULATION

This applies only if you have chosen OFF for change tracking.

By default, when you create a full-text index, SQL Server starts a background process to populate that index. If you turn off change tracking and specify NO_POPULATION, then you are limiting yourself solely to defining the full-text index but not actually putting any data in it to start. You can then schedule your own index population job to run later (presumably in low-demand hours of the day).

Altering Full-Text Indexes

Okay, so now you have an index, and you want to make changes to it. As you might expect, the new full-text syntax supports the notion of an ALTER statement. It is in the form of:

```
ALTER FULLTEXT INDEX ON <table name>
{
    ENABLE
    | DISABLE
    | SET CHANGE_TRACKING { MANUAL | AUTO | OFF }
    | ADD (<column name>
        [TYPE COLUMN <type column name> ]
        [LANGUAGE <language alias>] [,...n] )
    | WITH NO POPULATION
    | DROP (<column name> [,...n] )
    | WITH NO POPULATION
    | START { FULL | INCREMENTAL | UPDATE } POPULATION
    | STOP POPULATION
}
```

This ALTER has some substantial differences to previous ALTER statements we've dealt with! See how verbs like START and STOP are in there? This ALTER not only changes the definition of our full-text index but also can be used to manage the index somewhat —keep this difference in mind, as it is not very intuitive when you compare it to the other ALTER statements we use in SQL Server.

Chapter 21

Several elements of these work exactly as they did for the `CREATE` statement—we are merely changing a chosen option from one thing to another—however, some of this is totally new. Let's start with the more traditional `ALTER` statement items and then move on to the portions of this statement that are more management oriented.

ENABLE/DISABLE

These do what they say. If you disable a full-text index, the index is kept in place and all data remains intact. What changes is that the index is not available for full-text queries, and the index data is not updated (any updates that were in process when the `DISABLE` was issued will be stopped immediately).

When you `ENABLE`, it picks up where the index left off (it likely has catching up to do, but any data already there is kept intact, and you do not need to do a full repopulation).

ADD

This works just like the ignition definition of columns. For example, if we not wanted to add the `Instructions` column to our full-text index on `Production.ProductModel`, it would look like:

```
ALTER FULLTEXT INDEX ON Production.ProductModel  
    ADD ( Instructions )
```

The `LANGUAGE` and `TYPE COLUMN` properties also work just as they did in our early `CREATE`.

DROP

Again, works much as you would expect. If we were dropping the `Instructions` column we just added, that would look like:

```
ALTER FULLTEXT INDEX ON Production.ProductModel  
    DROP ( Instructions )
```

START ... POPULATION

`START` gives us three options as to what kind of populations we want to use.

FULL

The nice simple one—think of this as the command to “start over!” Every row will be reexamined, and the index will be rebuilt from scratch.

INCREMENTAL

This one is valid only if you have a timestamp column in your table and will start a population of rows changes since the last time a population was performed for the table. Think of this one as the “catch up on your work please!” version of populating. Incremental population does *not* require that change tracking be turned on.

UPDATE

This one is like incremental but does not require a timestamp column. However, it *does* require that change tracking be turned on. Like incremental, it will update only those rows that have changes since the last population.

STOP

This cancels any population that is currently running against this full-text index. This does *not* stop automatic change tracking—only full or incremental updates.

Dropping Full-Text Indexes

I'm sure by this point that you could figure this one out for yourself, but for the sake of completeness, here we go:

```
DROP FULLTEXT INDEX ON <table name>
```

So, were we to run the command:

```
DROP FULLTEXT INDEX ON Production.ProductModel
```

the full-text index would be gone!

Creating Full-Text Catalogs Using the Old Syntax

Prior to SQL Server 2005, we used a special system stored procedure called `sp_fulltext_catalog`. Technically speaking, this sproc does about everything needed to manage the catalog—essentially, it is all the commands we've talked about up to now all in one command. We use just this one stored procedure with a mixture of parameters to fit our particular situation.

Keep in mind that the various full-text-related stored procedures I discuss here are considered deprecated by Microsoft, and that means they will go away at some point. I include them here primarily for backward-compatibility reasons.

The general syntax is:

```
EXEC sp_fulltext_catalog [@ftcat =] '<name of catalog>' ,
    [@action =] '{CREATE|DROP|START_INCREMENTAL|START_FULL|STOP|REBUILD}' ,
    [,[@path =] '<root directory>' ]
```

The three arguments to the stored procedure are as follows:

Argument	Description
<code>@ftcat</code>	Does what it says. If you are using the CREATE option for the <code>@action</code> parameter, this is used as the name for your full-text catalog. For the other options, it is the name of the catalog you want to perform your action against.

Table continued on following page

Chapter 21

Argument	Description
@action	<p>This is basically a “What do you want to do” parameter. Since <code>sp_fulltext_catalog</code> can do about everything you need done to a catalog, you use this parameter to indicate what action you want done this time. It has six possible options:</p> <p>CREATE—Okay—who can figure this one out? Tick, tick, tick—bzzzt! That’s right! It means that you want to create a full-text catalog. When this option is used, the name specific with the <code>@ftcat</code> parameter must <i>not</i> already exist in the database or the create will fail. This is essentially the same as <code>CREATE FULLTEXT CATALOG</code>.</p> <p>DROP—Another toughie for sure—choose this option and your full-text catalog is gone. Note that you must delete all indexes that were in the catalog before you try and drop it or the drop operation will fail. This is the same as <code>DROP FULLTEXT CATALOG</code>.</p> <p>START_INCREMENTAL—This one starts the catalog population. Think of this as one piece of the <code>ALTER FULLTEXT CATALOG</code> command.</p> <p>START_FULL—Similar to <code>START_INCREMENTAL</code>, but a full population.</p> <p>STOP—Stops any population process that is already in progress.</p> <p>REBUILD—Effectively deletes and recreates the specified catalog. The catalog is left in an empty (unpopulated) state—if you want to populate the catalog after a REBUILD, then you must explicitly call <code>sp_fulltext_catalog</code> again and specify the <code>START_FULL</code> option.</p>
@path	<p>This optional argument specifies the path to the root directory that will contain the catalog. If the path is not specified, it is the ..\MSSQL\FTDATA directory under wherever your installation of SQL Server went.</p>

Okay, so now we’re armed and ready to fire away with a full-text catalog. For our examples in this chapter, it probably won’t come as any surprise to learn that I’m going to point you at the AdventureWorks database.

So if we were creating the same catalog we did with `CREATE FULLTEXT CATALOG`, it would look like:

```
USE AdventureWorks

EXEC sp_fulltext_catalog @ftcat = 'MainCatalog',
    @action = 'CREATE'
```

Old Syntax for Indexes

As was the case for the catalog, we're going to use another system sproc to create our actual indexes for us. Following the same general naming theme as our previous sproc, this one is called `sp_fulltext_table`, and the syntax looks like this:

```
EXEC sp_fulltext_table [@tablename =] '[<schema>.]<table>'
    ,[@action =] '{create|drop|deactivate|start_change_tracking|
stop_change_tracking|start_background_updateindex|
stop_background_updateindex|update_index|
start_full|start_incremental|stop}'
    ,[@ftcat =] '<fulltext catalog>'
    ,[@keyname =] '<index name>'
```

Let's take a brief look at each of the parameters, and you should quickly see that they all map to options in our `ALTER FULLTEXT INDEX` command:

<code>@tablename</code>	The name of the table. Optionally, the two-part name in the form of owner.object. The table is assumed to exist in the current database. If no such object exists in the database, an error is returned.
<code>@action</code>	The "What do you want done?" question again. This parameter has a long list of possible choices—we will explore each next.
<code>@ftcat</code>	The name of the full-text catalog that this index is to be created in. Should be provided only if the option for <code>@action</code> was <code>create</code> ; otherwise, leave this option off.
<code>@keyname</code>	Name of a single column unique key for this table. Usually the primary key, but, if you have a composite key, you may use a column with a <code>UNIQUE</code> constraint (you may need to add an identity column or similar mechanism just for this purpose).

The `@action` parameter has several options to it, so let's look at those.

<code>create</code>	Enables full-text search for the table specified in <code>@tablename</code> .
<code>drop</code>	Deactivates full-text for this table and deletes the full-text index from the catalog.
<code>activate</code>	If the table has had full-text deactivated, this option reactivates it. If change tracking is turned on, then running this option will also start population the next time a scheduled population run begins.
<code>deactivate</code>	Disables the table from participating in full-text search activity but leaves any population data in place for use if the table is reactivated.

Table continued on following page

start_change_tracking	If the table contains a timestamp column (required for incremental updates), this starts an incremental population of the full-text index. If no timestamp column exists, then this starts a full population of the full-text index. Repopulates (incremental or full as appropriate) automatically (may be immediate or next scheduled population run depending on other settings) if any changes happen to the table for non-BLOB columns.
stop_change_tracking	Ceases change tracking. Does not affect populations already in progress but eliminates the automatic population-based on changes.
start_background_updateindex	Begins immediate population of index with any changes found as a result of change tracking. Any subsequent updates are added to the index immediately.
stop_background_updateindex	Stops the immediate population process.
update_index	Causes any tracked changes not already applied to the full-text index to be applied immediately.
start_full	Immediately begin a full population for the target index.
start_incremental	Immediately begin an incremental population for the target index.
stop	Stop a population in progress.

More on Index Population

Unlike “normal” SQL Server indexes, which are naturally kept up to date by the very nature of SQL Server and the way it stores data, full-text indexes exist partially outside the realm of SQL Server and, as such, require a certain degree of intervention before the index will be up to date with the actual data it is supposed to represent.

Population comes in three—well, more like two and a half—flavors. Let’s look at each:

- ❑ **Full**—Is what it sounds like. With this kind of population, SQL Server basically forgets anything that it knew about the data previously and starts over. Every row is rescanned, and the index is rebuilt from scratch.
- ❑ **Incremental**—Under this option, SQL Server utilizes a column of type timestamp in order to keep track of what columns have changed since the last population. In this scenario, SQL Server only needs to record the changes for those rows that have changed in some manner. This option requires that the table in question have a timestamp column. Any updates that do not cause a change in the timestamp (nonlogged operations—usually BLOB activity) will not be detected unless something else in the same row changed.
- ❑ **Change tracking**—Tracks the actual changes since the last population. This option can help you keep your full-text indexes up to date at near real time; however, keep in mind that full-text

population is very CPU and memory intensive, and can bog down your server—weigh the notion of immediate updates against the notion that you may be able to hold your updates to off-peak hours for your server.

Unless you're using change tracking, population of your full-text indexes will occur only when you specifically start the process or according to a population schedule that you establish. With change tracking, you are provided two options: Scheduled Propagation and Background Update. The former applies the tracked changes at scheduled intervals; the latter is essentially always running in the background and applies changes as soon as possible after the changes are made.

Obviously, whenever you first create a full-text index or change the list of columns participating in the index, you need to completely repopulate the index (an incremental change of a previously empty index would mean that every row would have to be scanned in—right?). We can perform this repopulation at either the catalog or the table level. Typically, you'll perform repopulation at the table level for newly added or changed indexes, and repopulate at the catalog level when you are performing routine maintenance.

So, with this in mind, we should be ready to populate the full-text index we have created on our `Production.ProductModel` table. Had we not specifically stated `NO POPULATION`, then SQL Server would have populated the index automatically; however, since we did tell it not to populate, we have to order up our population. Since this is the first population, we probably want a full population (frankly, an incremental would have the same result, so it doesn't really matter, but it reads more logically this way). Using the new syntax, this would look like:

```
ALTER FULLTEXT INDEX ON Production.ProductModel  
    START FULL POPULATION
```

Full-text population runs as a background process. As such, your command will return a "completed successfully" message as soon as the population job is started. Do not take this message to mean that your index is done populating, which, if the index is against a large table, could potentially take hours to complete.

The older syntax makes use of the same sproc we already have seen—`sp_fulltext_table`. We'll run it against our `Production.ProductModel` table:

```
EXEC sp_fulltext_table @tabname = 'Production.ProductModel' ,  
    @action = 'start_full'
```

Notice that we did not include the full-text catalog and key name options—those are for use only when creating the full-text index. After that, they should not be used.

Since this table is relatively small, you shouldn't have to wait terribly long before you can run a query against it and get results:

```
SELECT ProductModelID, Name  
FROM Production.ProductModel  
WHERE CONTAINS(Name, 'Frame')
```

Chapter 21

This should get back something on the order of 10 rows:

```
ProductModelID Name
-----
5          HL Mountain Frame
6          HL Road Frame
7          HL Touring Frame
8          LL Mountain Frame
9          LL Road Frame
10         LL Touring Frame
14         ML Mountain Frame
15         ML Mountain Frame-W
16         ML Road Frame
17         ML Road Frame-W
```

(10 row(s) affected)

We have a full-text index, and it works! Time to move on to what that query we just ran is supposed to do and what other options we have available.

Full-Text Query Syntax

Full-text search has its own brand of query syntax. It adds special commands to extend T-SQL and to clearly indicate that we want the full-text engine to support our query rather than the regular SQL Server engine.

Fortunately, the basics of full-text queries are just that—basic. There are only four base statements to work with the full-text engine. They actually fall into two overlapping categories of two statements each:

	Exact or Inflectional Term	Meaning
Conditional	CONTAINS	FREETEXT
Ranked Table	CONTAINSTABLE	FREETEXTTABLE

The conditional predicates both work an awful lot like an `EXISTS` operator. Essentially they, for each row, provide a simple yes or no as to whether the row qualifies against the search condition provided. You use both of these in the `WHERE` clause of your queries. On the other hand, the two ranked queries do not provide conditions at all—instead, they return a tabular result set (which you can join to) that includes the key value of all the rows that found matches (that's what you join to) as well as a ranking to indicate the strength of the match.

Let's look more closely at each of the four keywords.

CONTAINS

This term looks for a match based on a particular word or phrase. By default, it's looking for an exact match (that is `swim` must be `swim`—not `swam`), but it can also use modifiers to look for what are called inflectional matches—(words that have the same root—such as `swim` and `swam`). `CONTAINS` recognizes certain keywords.

Looking at Things in Full: Full-Text Search

For now, we're going to stick with the simple form of `CONTAINS`. We will look at the advanced features after we have the basics of our four statements down (since they share certain modifiers, we'll look at those all at once).

The basic syntax, then, looks like this:

```
CONTAINS({<column>} * , '<search condition>')
```

You can name a specific column to check, or use `*`—in which case the condition will be compared for matches against any of the indexed columns. In its simplest form, the search condition should contain only a word or phrase.

There are two things worth pointing out here. First, remember that you will only get back results against columns that were included in the full-text index. In the final index we created on the ProductModel table—that means the search only includes the Name and CatalogDescription columns —columns like Introduction are not included in the search because they are not included in the index (you may recall that we dropped that column in a text of our ALTER syntax). Second, the search condition can be far more complex than the simple condition that we've shown here—but we'll get to that after you have the basic operations down.

For an example, let's go back to the query we used to prove that our population exercise had worked.

```
SELECT ProductModelID, Name  
FROM Production.ProductModel  
WHERE CONTAINS(Name, 'Frame')
```

What we've said we want here is the `ProductModelID` and `Name` columns for all the rows where the `Name` column in the index includes the word `Frame`.

If you check out the `Name` column for the results, you'll see that every row has an exact match.

Let's quickly look at another example. This time, we're going to run pretty much the same query, but we're going to look for the word `Sport`.

```
SELECT ProductModelID, Name  
FROM Production.ProductModel  
WHERE CONTAINS(Name, 'Sport'))
```

This time we get back just one row:

```
ProductModelID Name
```

```
-----  
33 Sport-100
```

```
(1 row(s) affected)
```

Again, we got back all the rows where the `Name` column had an exact match with the word `Sport`. Were you to look through the other rows in the table, however, you would find that there were other variations of the word `Sport` (a plural in this case), but they were not returned.

Again—the default behavior of `CONTAINS` is an exact match behavior.

FREETEXT

FREETEXT is an incredibly close cousin to CONTAINS. Indeed, their syntax is nearly identical:

```
FREETEXT({<column>}|*}, '<search condition>')
```

So, the only real difference is in the results you get back. You see, FREETEXT is a lot more forgiving in just how exact of a match it looks for. It is more interested in the meaning of the word than it is the exact letter-for-letter spelling.

To illustrate my point rather quickly here, let's look at our *Sport* query from the last section, but modify it to use FREETEXT instead of CONTAINS:

```
SELECT ProductModelID, Name  
FROM Production.ProductModel  
WHERE FREETEXT(Name, 'Sport')
```

When we execute this, we get back slightly different results than we did with CONTAINS:

```
ProductModelID Name  
-----  
13 Men's Sports Shorts  
33 Sport-100  
  
(2 row(s) affected)
```

The difference in this case comes in interpretation of the plurals—our FREETEXT query has picked up the row that contains the word *Sports*—not just those with the word *Sport*. FREETEXT can also handle things like swim versus swam and other word variations.

CONTAINSTABLE

CONTAINSTABLE, in terms of figuring out which rows would be match, works identically to CONTAINS. The difference is how the results are dealt with.

The syntax is similar, but with the twist of identifying which table the CONTAINSTABLE is going to operate against plus an optional limitation to just a top set of matches:

```
CONTAINSTABLE (<table>, {column|*}, '<contains search condition>' [, <top 'n'>])
```

Where CONTAINS returns a simple Boolean response suitable for use in a WHERE clause, CONTAINSTABLE returns a table—complete with rankings of how well the search phrase matched the row being returned.

Let's see what I mean here by running our original query, but with a CONTAINSTABLE this time:

```
SELECT *  
FROM CONTAINSTABLE(Production.ProductModel,Name, 'Sport')
```

This gets us back on row—just like with CONTAINS—but the information provided by the returned values is somewhat different:

```
KEY          RANK
-----
33           128
(1 row(s) affected)
```

We are provided with two columns:

- ❑ KEY — Remember when we said that our full-text index had to be able to relate to a single column key in the indexed table? Well, the KEY returned by CONTAINSTABLE relates exactly to that key column. That is, the value output in the column called KEY matches with a single unique row, as identified by the key, in the index table.
- ❑ RANK — A value from 0 to 1000 that indicates just how well the search result matched the row being returned—the higher the value, the better the match.

To make use of CONTAINSTABLE , we simply join our original table back to the CONTAINSTABLE result. For example:

```
SELECT Rank, ProductModelID, Name
FROM Production.ProductModel p
JOIN CONTAINSTABLE(Production.ProductModel, Name, 'Sport') ct
    ON p.ProductModelID = ct.[KEY]
```

Notice the use of brackets around the KEY column name. The reason why is that KEY is also a keyword. Remember from our rules of naming that, if we use a keyword for a column or table name (which you shouldn't do), you need to enclose them in square brackets.

This gets us back our original row, but this time we have the extra added information from the underlying table:

```
Rank          ProductModelID Name
-----
128           33             Sport-100
(1 row(s) affected)
```

In this case, the values in the Rank are the same, but, given more diverse values, we could have done things like:

- ❑ Filter based on some arbitrary Rank value. For example, we could want to return only the best matches based on score.
- ❑ Order by the rank (sort the rankings—most likely highest to lowest).

FREETEXTTABLE

Much as FREETEXT was the close cousin to CONTAINS, so too is FREETEXTTABLE the close cousin to CONTAINSTABLE. FREETEXTTABLE simply combines the more inexact word matching of FREETEXT with the tabular presentation found in CONTAINSTABLE.

We can then combine some of our previous examples to see how FREETEXTTABLE changes things:

```
SELECT Rank, ProductModelID, Name
FROM Production.ProductModel p
JOIN FREETEXTTABLE(Production.ProductModel,Name, 'Sport') ct
ON p.ProductModelID = ct.[KEY]
```

This gets us the same two rows we had with our original FREETEXT query, but with the kind of rankings we had with our CONTAINSTABLE:

Rank	ProductModelID	Name
102	13	Men's Sports Shorts
102	33	Sport-100

(2 row(s) affected)

Experiment with this some in your full-text efforts, and you'll see how rankings can give you a lot to work with.

Dealing with Phrases

All of our various full-text keywords can deal with the concept of phrases. How the phrases are parsed and handled, however, is somewhat different.

Let's start off with the most simple of examples—a simple two-word phrase. This time we'll say that the phrase we want to look for is *larger diameter*. To add a twist to thing, we want it no matter what column it is in (as long as the column is part of our full-text index).

```
SELECT ProductModelID, Name, CatalogDescription
FROM Production.ProductModel
WHERE CONTAINS(*, '"larger diameter")
```

Notice that the phrase was included in double quotation marks—we need to do this any time we want a set of words to be considered as a single unit. This does, however, get us back one row. The result is a little large (due to the size of the CatalogDescription column) to put in this text, but the relevant section is:

"The heat-treated welded aluminum frame has a larger diameter tube"

Our CONTAINS will check for rows that exactly match the phrase, as long as we enclose that phrase in double quotation marks (within the single quotes we always need on our search phrase). FREETEXT works in the same way.

Booleans

SQL Server also supports the use of Booleans in your searches. The Boolean keywords apply:

- AND
- OR
- AND NOT

There really isn't a whole lot of rocket science to these, so I'll launch right into a simple example and point out one caveat. Let's go with a variation on an example we used earlier:

```
SELECT ProductModelID, Name, CatalogDescription  
FROM Production.ProductModel  
WHERE CONTAINS(*, '"larger" OR "diameter"')
```

What we've done here is change from where we were searching for the exact phrase *larger diameter* to a search that is looking for *either* word without worrying about whether the words are used together or not. Execute this, and you'll see we get back three rows instead of just one.

The caveat that I mentioned earlier is that NOT cannot be used on its own—NOT is relevant only to full-text searches when used in conjunction with AND.

Proximity

FullText Search also allows us to make use of proximity terms. Currently, the list of supported proximity terms is a whopping one term long—NEAR. NEAR works a lot like it sounds—it says that the terms on either side of the NEAR keyword must be close to each other. Microsoft hasn't told us how close the words have to be to be considered NEAR, but figure around eight to ten words for most situations.

Technically, there is one more “word” on the proximity keyword list, but it isn't a “word” at all—rather a symbol. You can, if you choose, use a tilde (~) instead of the NEAR keyword. It works just the same. Personally, I recommend against this for readability reasons—not too many readers of your code are going to recognize what ~ means—but most of them will at least make a guess at NEAR.

For examples on how NEAR works, we're going to stick with CONTAINSTABLE. NEAR works much the same in the other full-text query operators, so we're just going to focus on what happens to the rankings in a NEAR query as well as what does and doesn't get included in the query.

For this example, we'll look at the words *welded* and *frame*:

```
SELECT Rank, ProductModelID, CatalogDescription  
FROM Production.ProductModel p  
JOIN CONTAINSTABLE(Production.ProductModel, *, 'welded near frame') ct  
ON p.ProductModelID = ct.[KEY]
```

I include only the first two columns here for brevity, but notice that we have different rankings on the two rows returned.

Chapter 21

```
Rank      ProductModelID
-----
14          19
2           25

(2 row(s) affected)
```

If you look carefully at the `CatalogDescription` column in your results (again, for brevity's sake, I haven't included all of the `CatalogDescription` column here), you'll see that both rows do indeed have both words but that the words are slightly closer together in the `ProductModelID` 19 row — thus a higher ranking.

Don't be surprised to see situations where a record that has your search criteria closer together gets ranked lower than one where the search criteria are not as close — remember that, even when you use the `NEAR` keyword, nearness is only one of several criteria that SQL Server uses to rank the rows. Other considerations such as percentage of words that match, case values, and more can play with the numbers on you.

Weighting

So, these rankings are all cool and whatnot, but what would we do if one of the words in our search criteria was more important than another?

To deal with situations where you need to give precedence to one or more words, Full-Text provides us with the `ISABOUT()` function and `WEIGHT` keyword. This syntax looks like this:

```
ISABOUT(<weighted term> WEIGHT (<weight value>), <weighted term> WEIGHT (<weighted term>), ...n)
```

Let's say that you want to allow customers to select among several kinds of bikes, but to further allow for selecting "preferred" options. For our example, let's say our customer is most interested in mountain bikes but is also interested in touring and road bikes — in that order. You could get a ranked listing using the following:

```
SELECT Rank, ProductModelID, Name
FROM Production.ProductModel pm
JOIN CONTAINSTABLE(Production.ProductModel, Name, 'ISABOUT (Road WEIGHT (.2),
Touring WEIGHT (.4), Mountain WEIGHT (.8) )' ) ct
ON pm.ProductModelID = ct.[KEY]
ORDER BY Rank DESC
```

Now take a look at the results:

Rank	ProductModelID	Name
31	121	Fender Set - Mountain
31	5	HL Mountain Frame
31	46	HL Mountain Front Wheel
...		
...		
...		

```
31      35          Touring-2000
31      36          Touring-3000
31     120          Touring-Panniers
31      37          Women's Mountain Shorts
7       113          Road Bottle Cage
7       93           Road Tire Tube
7       25           Road-150
7       26           Road-250
...
...
...
7       78           HL Road Rear Wheel
7       76           HL Road Seat/Saddle 1
7       84           HL Road Seat/Saddle 2
7       90           HL Road Tire

(89 row(s) affected)
```

Note that not everything is perfect in our world—some touring entries come before our more heavily weighted mountain options, but if you look the list over, you will see we have indeed created a very heavy bias toward mountain bikes in our rankings.

Inflectional

This one doesn't really apply to FREETEXT, as FREETEXT is inherently inflectional. What is INFLECTIONAL you ask? Well, it's basically telling SQL Server that different forms of the word that have the same general meaning. The syntax looks like this:

```
FORMSOF(INFLECTIONAL, <term>[, <term>[, ...n]] )
```

An inflectional form of a word is one that has the same general meaning—for example, *swam* is just the past tense of *swim*. The underlying meaning is the same.

Noise Words

There are tons and tons of words in use in different languages (Full-Text supports more than just U.S. English!). Most languages have certain words that appear over and over again with little intrinsic meaning to them. In the English language, for example, prepositions (you, she, he, etc.), articles (the, a, an), and conjunctions (and, but, or) are just few examples of words that appear in many, many sentences but are not integral to the meaning of that sentence.

If SQL Server paid attention to those words, and we did searches based on them, then we would drown in the results that SQL Server gave us in our queries—quite often, every single row in the table would be returned! The solution comes in the form of what is called a *noise word* list. This is a list of words that SQL Server ignores when considering matches.

The noise word list is, by default, stored in a text file in the path:

```
Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\FTData
```

Chapter 21

For U.S. English, the name of the file is `noiseENU.txt`. Other noise files can be found in the same subdirectory to support several other languages.

You can add and delete words from this list as suits the particular needs of your application. For example, if you are in the business of selling tractor-trailer rigs, then you might want to add words like *hauling* to your noise word list—more than likely, a huge percentage of your customers have that word in their name, so it is relatively unhelpful in searches. To do this, you would just add the word *hauling* to `noiseENU.txt` (or whatever language noise word file is appropriate for your installation), and, after the next population, hauling would begin being ignored.

Adding and removing words from the noise list is something of a double-edged sword. When you add a word to the list, it means that searches involving that word are no longer going to return the results that users are more than likely going to expect. By the same token, it also, depending on the frequency with which the word is used, can dramatically shrink the processing time and size of your catalogs.

Summary

Full-Text runs as a separate service in on the SQL Server box and takes up a separate block of memory. Be sure to keep both services in mind when planning how much memory your system needs, and also when establishing the amount of memory used by either service.

When you implement Full-Text, also consider the load the population process is going to place of your server, and balance that against how quickly you need changes reflected in search results. If possible, delay repopulation of full-text indexes until the non-peak hours on your system.

Full-Text Search is a powerful and fast way of referencing the contents of most any character-based columns. It is substantially more efficient and powerful than a `LIKE` clause but comes with additional overhead in terms of both space and processing time.

22

Security

There are probably as many ideas on security as there are programmers. It's one of those things where there isn't necessarily a right way to do it, but there are definitely plenty of wrong ones.

The first thing to understand about security is that there is no such thing as a totally secure application. If you can make it secure, rest assured that someone, somewhere, can defeat your efforts and "hack" into the system. Even with this knowledge, the goal still needs to be to keep unwanted intruders out of your system. The good news about security is that, for most instances, you can fairly easily make it such a hassle that 99.999 percent of people out there won't want to bother with it. For the other .001 percent, I can only encourage you to make sure that all your employees have a life so they fall into the 99.999 percent. The .001 percent will hopefully find someplace else to go.

With SQL Server 2005, Microsoft has gotten very serious about security for SQL Server. There are a ton of new features here, and, while there were already books that were specific to SQL Server security out there before, I can imagine them as being huge tomes now—the subject has grown that much with this release.

In this chapter, we're going to cover:

- Security basics
- SQL Server security options
- Database and server roles
- Application roles
- Credentials
- Certificates
- Schema management
- XML integration security issues
- More advanced security

What we'll discover is that there are a lot of different ways to approach the security problem. Security goes way beyond giving someone a user ID and a password—we'll see many of the things that you need to think about.

Before beginning any of the examples in this chapter, you'll need to load and execute the script called `NorthwindSecure.sql`. This builds a special database we'll use throughout this chapter. You can download what you need for this at the book's Web site at www.wrox.com.

Okay, so this is a chapter where I have to make you create a working database in order for the examples to work—my apologies for that. What we're going to utilize is the old Northwind database but with any changes to permissions removed. The NorthwindSecure database that we'll use throughout this chapter is a more typical database scenario—that is, it has absolutely no permissions added to it beyond what comes naturally with creating tables and objects (which means NONE). We'll learn how to deal with this and explicitly add what permissions we want as the chapter progresses.

Security Basics

I'm sure that a fair amount of what we're going to look into in this section is going to seem exceedingly stupid—I mean, won't everyone know this stuff? Judging by how often I see violations of even the most simple of these rules, I would say, "No, apparently they don't." All I can ask is that you bear with me, and don't skip ahead. As seemingly obvious as some of this stuff is, you'd be amazed how often it gets forgotten or just plain ignored.

Among the different basics that we'll look at here are:

- ❑ One person, one login ID, one password
- ❑ Password expirations
- ❑ Password length and makeup
- ❑ Number of attempts to log in
- ❑ Storage of user ID and password information

One Person, One Login, One Password

It never ceases to shock me how, everywhere I go, I almost never fail to find that the establishment has at least one "global" user—some login into the network or particular applications that is usually known by nearly everyone in the department or even the whole company. Often, this "global" user has *carte blanche* (in other words, complete) access. For SQL Server, it used to be common that installations hadn't even bothered to set the `sa` password to something other than a blank password. This is a very bad scenario indeed.

Prior to SQL Server 2000, the default password for the `sa` account was null—that is, it didn't have one. Thankfully, SQL Server 2000 not only changed this default, SQL Server will now proactively tell you that you are effectively being an idiot if you insist on making it blank—good for the dev team. The thing to watch out for is that, while you're developing, it's really common to still set it to something "easy." You still need to remember to change it before you go into production or to make it something hard from the beginning if your development server is going to be exposed directly to the Internet or some other nontrustworthy access.

Even now, when most installations do have something other than a null password, it is very common for lots of people to know what that password is.

The first basic then is that if everyone has access to a user ID that is essentially anonymous (if everyone knows it, it could be that anyone has used it) and has access to everything, then you've defeated your security model entirely. Likewise, if you give every user a login that has full access to everything, you've again severely damaged your security prospects. The only real benefit that's left is being able to tell who's who as far as who is connected at any point in time (assuming that they are really using their individual login rather than the global login).

Users that have *carte blanche* access should be limited to just one or two people. Ideally, if you need passwords for such *carte blanche* access, then you would want separate logins that each have the access, but only one person would know the password for each login.

You'll find that users will often share their passwords with someone else in order to let someone temporarily gain some level of access (usually because the owner of the login ID is either out of the office or doesn't have time to bother with doing it themselves at the time)—you should make this nothing short of a hanging offense if possible.

The problem created by password sharing is multifold. First, some users are getting access to something that you previously decided not to give them (otherwise, why don't they have the necessary rights for themselves?)—if you didn't want them to have that access before, why do you want them to have it now? Second, a user that's not supposed to have access probably will now have that access semipermanently. Since users almost never change their passwords (unless forced to), the person they gave the password to will probably be able to use that login ID indefinitely—and, I assure you, they will! Third, you again lose auditing. You may have something that tracks which user did what based on the login ID. If more than one person has the password for that login ID, how can you be sure which person was logged into that login ID at the time?

This means that if someone is going to be out of the office for some time, perhaps because he is sick or on vacation, and someone else is temporarily going to be doing his job, a new login ID and password should be created specifically for that replacement person (or a modification to the access rights of his existing login ID should be made), and it should be deleted as soon as the original person has returned.

To summarize, stay away from global user accounts whenever possible. If you must have them, keep their use limited to as few people as at all possible—usually this should be kept to just two (one to be a main user, and one person as a backup if the first person isn't available). If you really must have more than one person with significant access, then consider creating multiple accounts (one per user) that have the necessary level of access. By following these simple steps, you'll find you'll do a lot for both the security and auditability of the system.

Password Expiration

Using expiration of passwords tends to be either abused or ignored. That's because it's a good idea that often goes bad.

The principle behind password expiration is to set up your system to have passwords that automatically expire after a certain period of time. After that time, the user must change the password to continue to have access to the account. The concept has been around many years, and if you work in a larger corporation, there's a good chance that the auditors from your accounting firm are already insisting that you implement some form of password expiration.

Beginning with SQL Server 2005, you can now enforce Windows security rights even for your SQL Server specific passwords. Alternatively, you can just use Windows-based security (more on that in the next section).

What Do You Get for Your Effort?

So, what does password expiration get you? Well, remember that, in the final part of the previous section, I said that once a password is shared, the user would have that access forever? Well, this is the exception. If you expire passwords, then you refresh the level of your security—at least temporarily. The password would have to be shared a second time in order for the user to regain access. While this is far from foolproof (often, the owner of the login ID will often be more than happy to share it again), it does deal with the situation where the sharing of the password was really just intended for one-time use. Often, users who share their passwords don't even realize that months later; the other user still has the password and may be using it on occasion to gain access to something they would not have, based on their own security.

Now the Bad News

It is very possible to get too much of a good thing. I mentioned earlier how many audit firms will expect their clients to implement a model where a user's password regularly expires, say, every 30 days—this is a very bad idea indeed.

Every installation that I've seen that does this—with exception—has *worse* security after implementing a 30-day expiration policy. The problem is, as you might expect, multi-fold in nature.

- ❑ First, technical support calls go way up. When users change passwords that often, they simply can't memorize them all. They can't remember which month's password they are supposed to use, so they are constantly calling for support to reset the password because they forgot what it is.
- ❑ Second, and much more important, the users get tired of both thinking of new passwords and remembering them. Experience has shown me that, for more than 90 percent of the users I've worked with in installations that use a 30-day expiration, users change their passwords to incredibly predictable (and therefore hackable) words or word/number combinations. Indeed, this often gets to a level where perhaps 50 percent or more of your users will have the same password—they are all using things like MMMYY where MMM is the month and YY is the year. For example, for January 1996 they might have used JAN96 for their password. Pretty soon, everyone in the place is doing something like that.

I've seen some companies try and deal with this by implementing something of a password sniffer—it checks the password when you go to change it. The sniffing process looks for passwords that incorporate your name or start with a month prefix. These mechanisms are weak at best.

Users are far smarter than you often give them credit for. It took about a week for most users to circumvent the first one of these password sniffers I saw—they simply changed their passwords to have an "X" prefix on them, and otherwise stayed with the same MMMYY format they had been using before. In short, the sniffer wound up doing next to nothing.

The bottom line here is to not get carried away with your expiration policy. Make it short enough to get reasonable turnover and deal with shared or stolen passwords but don't make it so often that users rebel and start using weak passwords. Personally, I suggest nothing more frequent than 90 days and nothing longer than 180 days.

Password Length and Makeup

Ah, an era of rejoicing for SQL Server in this area. In previous versions, you really didn't have much control over this if you were using SQL Server security. You can now have SQL Server enforce your Windows password policy (which you can adjust using utilities in Windows).

Password Length

Realize that, for each possible alphanumeric digit the user includes in the password, they are increasing the number of possible passwords by a factor of at least 36 (really a few more given special characters, but even 36 is enough to make the point here). That means there are only 36 possible single character passwords, but 1,296 possible two-character passwords. Go up to three characters, and you increase the possibilities to 46,656. By the time you add a fourth character, you're well over a million possibilities. The permutations just keep going up as you require more and more characters. The downside, though, is that it becomes more and more difficult for your users to remember what their password was and to actually think up passwords. Indeed, I suspect that you'll find that requiring anything more than 5 or 6 characters will generate a full-scale revolt from your end users.

Password Makeup

All right, so I've pointed out that, if you make it a requirement to use at least four alphanumeric characters, you've created a situation where there are over a million possible password combinations. The problem comes when you realize that people aren't really going to use all those combinations — they are going to use words or names that they are familiar with. Considering that the average person only uses about 5,000 words on a regular basis, that doesn't leave you with very many words to try out if you're a hacker.

If you're implementing something other than the default Windows password policy, then consider requiring that at least one character be alphabetic in nature (no numbers, just letters) and that at least one character be numeric. This rules out simple numbers that are easy to guess (people really like to use their Social Security number, telephone number, or birthdays) and all words. The user can still create things that are easy to remember for them — say "77pizzas" — but the password can't be pulled out of a dictionary. Any hacker is forced to truly try each permutation in order to try and break in.

Number of Tries to Log In

Regardless of how you're physically storing the user and password information, your login screen should have logic to it that limits the number of tries that someone gets to log in. The response if they go over the limit can range in strength, but you want to make sure you throw in some sort of device that makes it difficult to set up a routine to try out all the passwords programmatically.

How many tries to allow isn't really that important as long as it's a reasonably small number — I usually use three times, but I've seen four and five in some places and that's fine too.

If you're utilizing the Windows password policy enforcement, then SQL Server will check the login attempts versus a bad password limit and enforce that policy.

Storage of User and Password Information

This obviously only applies if you are cooking your own security system rather than using the built-in Windows and/or SQL Server security systems (but many Web applications will do that), and, for the

Chapter 22

most part, there's no rocket science in how to store user profile and password information. There are, however, a few things to think about:

- ❑ Since you need to be able to get at the information initially, you will have to do one of the following three things:
 - ❑ Compile a password right into the client application or component (and then make sure that the proper login and password is created on any server that you install your application on).
 - ❑ Utilize SQL Server's new encryption technologies to encrypt and decrypt the data in the database.
 - ❑ Require something of a double password situation—one to get the user as far as the regular password information, and one to get them to the real application. Forcing a user into two logins is generally unacceptable, which pushes you back to one of the other two options in most cases.
- ❑ If you go with a double password scenario, you'll want the access for the first login to be limited to just a stored procedure execution if possible. By doing this, you can allow the first login to obtain the validation that it needs while not revealing anything to anyone that tries to login through Management Studio. Have your stored procedure (sproc) accept a user ID and password, and simply pass back either a Boolean (true/false that they can log in) or pass back a recordset that lists what screens and functions the user can see at the client end. If you use a raw SELECT statement, then you won't be able to restrict what they can see.

One solution I've implemented close to this scenario was to have a view that mapped the current SQL Server login to other login information. In this case, an application role was used that gave the application complete access to everything—the application had to know what the user could and couldn't do. All the user's login had a right to do was execute a stored procedure to request a listing of their rights. The sproc looked something like this:

```
CREATE PROC GetUserRights
AS

DECLARE @User varchar(128)
SELECT @User = USER_NAME()
SELECT * FROM UserPermissions WHERE LoginID = @User
```

- ❑ If you're going to store password information in the system—encrypt it!!! I can't say enough about the importance of this. Most users will use their passwords for more than one thing—it just makes life a lot easier when you have less to remember. By encrypting the data before you put it in the database, you ensure that no one is going to stumble across a user's password information—even accidentally. They may see it, but what they see is not usable unless they have the key to decrypt it.

There is really very little excuse for not encrypting data in SQL Server 2005. We now have a wide array of very, very secure encryption options available (more on this later in the chapter)—**use them!**

Security Options

As far as built-in options go, you have two choices in how to set up security under SQL Server.

- Windows integrated security** — The user logs in to Windows not SQL Server. Authentication is done via Windows with trusted connections.
- Standard security** — The user logs into SQL Server separately from logging in to Windows. Authentication is done using SQL Server.

Let's take a look at both.

SQL Server Security

We'll start with SQL Server's built-in login model. This has gotten substantially more robust with SQL Server 2005. The relatively simplistic model is still available, but there is now tons more you can do to add extra touches to just how secure your server and databases are.

With SQL Server security, you create a *login ID* that is completely separate from your network login information. Some of the pros for using SQL Server security include:

- The user doesn't necessarily have to be a domain user in order to gain access to the system.
- It's easier to gain programmatic control over the user information.

Some of the cons are:

- Your users may have to login twice or more — once into whatever network access they have, and once into the SQL Server for each connection they create from a separate application.
- Two logins mean more maintenance for your DBA.
- If multiple passwords are required, they can easily get out of synch, and that leads to an awful lot of failed logins or forgotten passwords. (Does this sound familiar, "Let's see now, which one was it for this login?")

An example of logging in using SQL Server security would be the use of the `sa` account that you've probably been using for much of this book. It doesn't matter how you've logged in to your network, you log in to the SQL Server using a login ID of `sa` and a separate password (which you've hopefully set to something very secure).

On an ongoing basis, you really don't want to be doing things day-to-day logged in as `sa`. Why? Well, it will probably only take you a minute or two of thought to figure out many of the terrible things you can do by sheer accident when you're using the `sa` account. Using `sa` means you have complete access to everything; that means the `DROP TABLE` statement you execute when you are in the wrong database will actually do what you told it — drop that table!!! About all you'll be left to say is "oops!" Your boss will probably be saying something completely different.

Even if you do want to always have *carte blanche* access, just use the `sa` account to make your regular user account a member of the `sysadmins` server role. That gives you the power of `sa`, but gains you the extra security of separate passwords and the audit trail (in Profiler or when looking at system activity) of who is currently logged in to the system.

Creating and Managing Logins

There are currently five ways to create logins on a SQL Server:

- ❑ By using CREATE LOGIN
- ❑ By using the Management Studio
- ❑ SQL Management Objects (SMO)
- ❑ By using one of the several other options that remain solely for backward compatibility

CREATE LOGIN

`CREATE LOGIN` is new with SQL Server 2005 and is part of a general effort by Microsoft to standardize the syntax used to create database and server objects. It deprecates `sp_addlogin`, which was the procedural way of adding logins in prior versions, and looks like the `CREATE <object> <object type>` syntax that we've seen repeatedly in SQL but with some of the extra option requirements that we've seen with things like stored procedures.

The most basic syntax is straightforward, but how the options can be mixed can become something of a pain to understand. The overall syntax looks like this:

```
CREATE LOGIN <login name>
[ { WITH
    PASSWORD = '<password>' [ HASHED ] [ MUST_CHANGE ]
    [, SID = <sid>
      | DEFAULT_DATABASE = <database>
      | DEFAULT_LANGUAGE = <language>
      | CHECK_EXPIRATION = { ON | OFF }
      | CHECK_POLICY = { ON | OFF }
      | CREDENTIAL = <credential name>
      [, ... <next option>] ]
    } |
    { FROM
      WINDOWS
        [ WITH DEFAULT_DATABASE = <database>
          | DEFAULT_LANGUAGE = <language> ]
        | CERTIFICATE <certificate name>
        | ASYMMETRIC KEY <asymmetric key name>
      }
    ]
}
```

The key part that sets the tone for things is the choice of a `FROM` versus a `WITH` clause immediately following the login name, so let's look at those along with the options as they are relevant to either the `FROM` or `WITH` clause they belong to.

CREATE LOGIN . . . WITH

The `WITH` clause immediately puts you into defining options that go with SQL Server authentication based logins as opposed to any other authentication method. It is only relevant if you have SQL Server security enabled (as opposed to just Windows security). The number of options here can seem daunting, so let's break them down.

Most of these options are genuinely new. Okay, yeah sure, you're saying: "I thought this whole command was new?" Yeah, it is, but CREATE LOGIN ... WITH is essentially the part of CREATE LOGIN that deprecates the old sp_addlogin. With SQL Server 2005, we have greatly improved security options even for SQL Server based security, and those new options are reflected in options for CREATE LOGIN ... WITH that did not exist under sp_addlogin.

Option	Description
PASSWORD	This is, of course, just what it sounds like. The tricky part of this is the question of whether the password is in clear text (in which case SQL Server will encrypt it as it adds it) or whether it is already hashed (in which case you need to supply the HASHED keyword that is covered next).
HASHED	This follows your password, and is used only if the password you supplied was already hashed (encrypted). In that case, SQL Server adds the password without re-encrypting it.
MUST_CHANGE	This is another one of those "is what it sounds like" things. In short, if you supply this option, then the users will be prompted to change their password the first time they login.
SID	Allows you to manually specify what GUID SQL Server will use to identify this login. If you don't supply this (and doing so is something I would consider to be an extreme case), then SQL Server will generate one for you.
DEFAULT_DATABASE	This is the database that will be made current each time the user logs in.
DEFAULT_LANGUAGE	This is the language that things like errors and other system messages will be delivered in for the user.
CHECK_EXPIRATION	Sets whether SQL Server will enforce the password expiration policy. By default, the password will <i>not</i> expire. Setting this to ON will enforce policy.
CHECK_POLICY	Sets whether SQL Server will enforce the password policy (length, character requirements, etc.). By default, the password must meet the Windows password policy. Setting this to OFF will allow virtually any password to be used.
CREDENTIAL	This names a credential (and you'll cover what these are more later) for this login to be mapped to. In short, this maps this login to a set of permissions that may allow them to perform actions outside of SQL Server (such as network access and such).

Any of these can be missed together, and the order in which you provide them matters only in the cast of HASHED and MUST_CHANGE (which must follow the PASSWORD option if you're going to utilize them at all).

CREATE LOGIN ... FROM

The FROM clause implies that this login isn't SQL Server specific. The FROM clause specifies the source of that login. The source falls into a few different categories:

- ❑ WINDOWS—In this case, we are mapping to an existing windows login or group. This is basically saying “Take this existing Windows user or group, and give them rights to my SQL Server.” Notice that I say “or group.” You can map SQL Server to a Windows group, and that implies that any member of that group will be granted that level of access to your SQL Server. This is really handy for managing users in your network. For example, if you want everyone in accounting to have a certain set of rights in SQL Server, you could create a Windows group called Accounting and map that to a SQL Server login. If you hire someone new, then as soon as you add them to the Accounting group they will have access not only to whatever Windows resources the Accounting group has, but also all the SQL Server permissions that the Accounting group has.
If you use Windows as your `FROM` sources, then you can also supply a `WITH` clause similar to a SQL Server-based login, but limited to just the default database and language.
- ❑ CERTIFICATE—This kind of login is based off of a X.509 certificate that you’ve already associated with your server by using the `CREATE CERTIFICATE` command. Certificates can be used in several different ways, but in the end, they essentially serve as a recognized secure encryption key. SQL Server has its own “certificate authority” or can import those generated from other sources. Essentially, presentation of this certificate serves as authorization to login to the SQL Server.
- ❑ ASYMMETRIC KEY—Asymmetric keys are a different flavor of the same general notion that certificates work under. Essentially, a key that is presented that SQL Server trusts, and therefore it grants access. Asymmetric keys are merely a different method of presenting a secure key.

ALTER LOGIN

As with most `CREATE` statements we’ve seen in SQL, `CREATE LOGIN` has a complimenting statement in the form of `ALTER LOGIN`. As with most `ALTER` statements, the syntax is primarily a subset of the options found in the related `CREATE` statement:

```
ALTER LOGIN <login name>
[ { ENABLE | DISABLE } ]
[ { WITH
    PASSWORD = '<password>'
    [ { OLD_PASSWORD = '<old password>' }
    | [ UNLOCK ] [ MUST_CHANGE ] ]
    | DEFAULT_DATABASE = <database>
    | DEFAULT_LANGUAGE = <language>
    | NAME = <new login name>
    | CHECK_EXPIRATION = { ON | OFF }
    | CHECK_POLICY = { ON | OFF }
    | CREDENTIAL = <credential name>
    | NO CREDENTIAL ] ]
```

Most of these are exactly the same as they were with the `CREATE` statement, but let’s look at the few differences.

Option	Description
ENABLE DISABLE	Enables or disables the login. This is something of an indicator of whether the login is considered active in the system or not, and ENABLE should not be confused with UNLOCK (they are different things). Disabling a login leaves it in place but disallows use of the login. Enabling reactivates the login.
OLD_PASSWORD	This one applies only if a given login is utilizing ALTER LOGIN to change its own password. Security administrators with the rights to change the password at all are unlikely to know the old password and have the right to set a new password without knowing the old one.
UNLOCK	This allows a user to attempt to login again after the login has been locked out due to exceeding the bad password count.
NAME	This allows you to change the login name, while otherwise retaining all of the old rights and other properties of the login.
NO_CREDENTIAL	This disassociates the login with whatever credential it may have previously been mapped to.

DROP LOGIN

This works just like any other DROP statement in SQL Server.

```
DROP LOGIN <login name>
```

And it's gone.

Creating a Login Using the Management Studio

Creating a login using Management Studio is fairly straightforward and is much the same as it is for most other objects in SQL Server. Just navigate to the appropriate mode in the Object Explorer (in this case, Security→Logins), right-click, and choose New Login..., as shown in Figure 22-1.

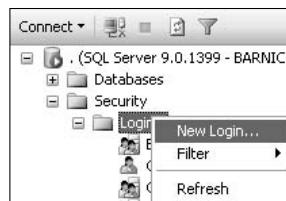


Figure 22-1

This gets us the typical CREATE dialog that we've seen repeatedly in this book, but adjusted for the properties that are appropriate for a login (all the same things we reviewed in the CREATE LOGIN section earlier in the chapter, plus a number of additional areas we have yet to take a look at), as shown in Figure 22-2.

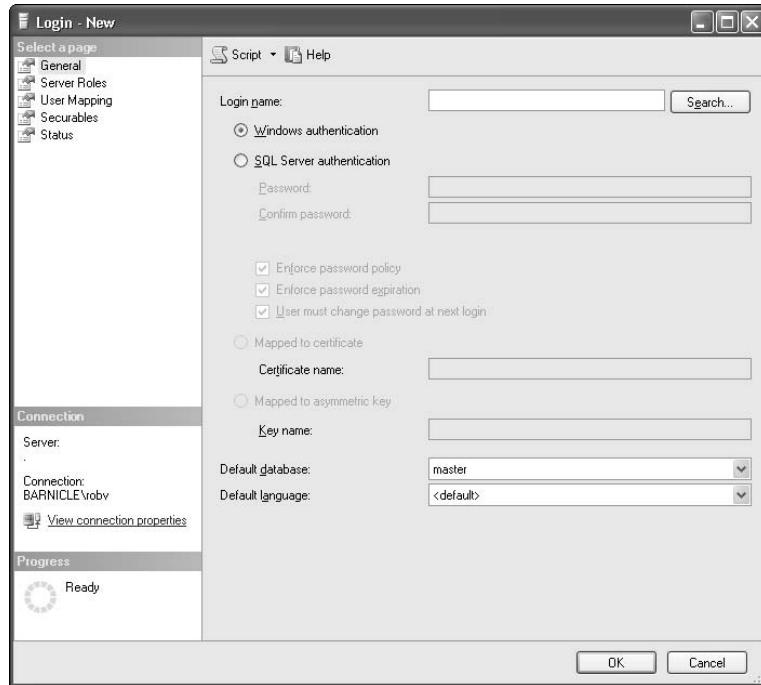


Figure 22-2

Only this first set of properties (the General properties) maps to the `CREATE LOGIN` syntax. The additional tabs map to other objects we will be creating as we continue through the chapter.

We will be reviewing several other kinds of objects that get associated with logins in some fashion. For now, the thing to notice is how the user interface in Management Studio lets you do everything at once. As we'll see as we continue the chapter, when creating these objects using code, we have to do each step separately rather than all at once as Management Studio offers (as you might imagine, it's really just collecting all the necessary information in advance and then issuing all those individual programmatic steps for us).

SQL Management Objects

This is largely out of scope for this chapter (We cover SMO in its own chapter later on), but I did want to specifically point out that SMO can create logins for you using a straightforward object model as opposed to the `CREATE` statement approach. See Chapter 25 for more information.

Legacy Options

There are three older options of significance when considering the way that logins have been created in past versions of SQL Server.

- ❑ `sp_addlogin` and related sprocs—This was a stored procedure that essentially maps to `CREATE LOGIN` except that several parts of the `CREATE LOGIN` statement implement things that are not supported in SQL Server 2000 and prior versions. The basics (creating the typical login as opposed to the certificate or asymmetric key approach) are all there though. We'll take a fuller look at `sp_addlogin` shortly.

- ❑ **SQL-DMO**—Distributed Management Objects (DMO) has finished being deprecated with SQL Server 2005. I say “finished” because when SQL Server 2000 came out, WMI was supposed to be the new way of doing things—well, it didn’t turn out so well. Fairly early on in the 2005 dev cycle, a decision was made to go with a more .NET-oriented model, and, thus, SMO was born. SMO is a book unto itself and is well outside the scope of this book. That said, think of DMO as being SMO before there was SMO. The models are fairly different, but the basic goal was the same.
- ❑ **WMI**—Windows Management Instrumentation is an implementation of an industry-standard Web management protocol. As I mentioned a bit ago, when SQL Server 2000 first came out, the thinking was that a WMI-based model was going to take over as the primary way of automating SQL Server management. In the end, there was no WMI-based model implemented that came anywhere close to being up to the task of exposing all the things we need in SQL Server, and that effort would seem to have been largely junked. Much like DMO, WMI is now outside the scope of this book, but realize that it’s out there and remains an option if you need to manage older versions of SQL Server.

A Quick Look at `sp_addlogin`

This sproc does exactly what it says, and it was the old way of implementing the things that `CREATE LOGIN` does for us today. It requires only one parameter, but most of the time you’ll use two or three. There are a couple of additional parameters, but you’ll find that you use those far more rarely. The syntax looks like this:

```
EXEC sp_addlogin [@loginame =] <'login'>
    [,[@passwd =] <'password'>]
    [,[@defdb =] <'database'>]
    [,[@deflanguage =] <'language'>]
    [,[@sid =] 'sid']
    [,[@encryptopt =] <'encryption_option'>]
```

Parameter	Description
<code>@loginame</code>	Just what it sounds like—this is the login ID that will be used.
<code>@passwd</code>	Even more what it sounds like—the password that is used to log in using the aforementioned login ID.
<code>@defdb</code>	The default database. This defines what is the first “current” database when the user logs in. Normally, this will be the main database your application uses. If left unspecified, the default will be the <code>master</code> database (you usually don’t want that, so be sure to provide this parameter).
<code>@deflanguage</code>	The default language for this user. You can use this to override the system default if you are supporting localization.
<code>@sid</code>	A binary number that becomes the <i>system identifier (SID)</i> for your login ID. If you don’t supply an SID, SQL Server generates one for you. Since SIDs must be unique, any SID you supply must not already exist in the system. Using a specific SID can be handy when you are restoring your database to a different server or are otherwise migrating login information.

Table continued on following page

Parameter	Description
@encryptopt	The user's login ID and password information is stored in the sysusers table in the master database. The @encryptopt determines whether the password stored in the master database is encrypted or not. By default, (or if you provide a NULL in this parameter), the password is indeed encrypted. The other options are skip_encryption, which does just what it says — the password is not encrypted, and skip_encryption_old, which is there only for backward compatibility, and should not be used.

As you can see, most of the items here map directly to CREATE LOGIN, and that is the way I recommend doing things unless you need to utilize sp_addlogin for backward-compatibility reasons.

sp_password

Since we've looked at sp_addlogin, we ought to look at sp_password. While ALTER LOGIN gives you the ability to address password maintenance on a login, sp_addlogin had no such functionality — sp_password takes care of that. The syntax is pretty straightforward:

```
sp_password [[@old =] <'old password'>,]
[@new =] <'new password'>
[,[@loginname =] <'login'>]
```

The new and old password parameters work, of course, just exactly as you would expect. You need to accept those from the user and pass them into the sproc. Note, however, that the login is an optional parameter. If you don't supply it, then it will assume that you want to change the password on the login used for the current connection. Note that sp_password cannot be executed as part of a transaction.

You might be thinking something like, "Don't most systems require you to enter the new password twice?" Indeed they do. So the follow up question is, "How come sp_password doesn't do that?" The answer is a simple one — because SQL Server leaves that up to you. You would include the logic to check for a double entry of the new password in your client application before you ever got as far as using sp_password. This same issue exists for ALTER LOGIN.

sp_grantlogin

This simulates the CREATE LOGIN...FROM functionality as relates to Windows logins (mapping from certificates and asymmetric keys did not exist as they do now prior to SQL Server 2005). The syntax is straightforward:

```
sp_grantlogin [@loginname = ] '<Domain Name>\<Windows User Name>'
```

Windows Integrated Security

Windows security gives us the capability to map logins from trusted Windows domains into our SQL Server.

It is simply a model where you take existing Windows domain user accounts or groups and provide SQL Server rights to them directly rather than forcing users to keep separate passwords and make separate logins.

Windows security allows:

- Maintenance of a user's access from just one place
- Granting of SQL Server rights simply by adding a user to a Windows group (this means that you often don't have to even go into SQL Server in order to grant access to a user)
- Your users to need to remember only one password and login

That being said, let's take a look at how to grant specific rights to specific users.

User Rights

The simplest definition of what a *user right* is would be something like, "what a user can and can't do." In this case, the simple definition is a pretty good one.

User rights fall into three categories:

- The right to login
- The right to access a specific database
- The right to perform specific actions on particular objects within that database

Since we've already looked at creating logins, we'll focus here on the specific rights that a login can have.

Granting Access to a Specific Database

The first thing that you need to do if you want a user to have access to a database is to grant the user the right to access that database. This can be done in Management Studio by adding the user to the Users member of the Databases node of your server. To add a user using T-SQL, you need to use `CREATE USER` or the legacy stored procedure `sp_grantdbaccess`.

Note that as you `CREATE` a user in the database, those permissions are actually stored in the database and mapped to the server's identifier for that user. As you restore a database, you may have to remap user rights to the server identifiers where you restored the database.

CREATE USER

The `CREATE USER` command adds a new user to the database. That user can be sourced from an existing login, certificate, or asymmetric key, or can be local to just the current database. The syntax looks like this:

```
CREATE USER <user name>
[ { { FOR | FROM }
{
    LOGIN <login name>
    | CERTIFICATE <certificate name>
```

Chapter 22

```
| ASYMMETRIC KEY <key name>
}
| WITHOUT LOGIN ]
[ WITH DEFAULT_SCHEMA = <schema name> ]
```

Let's take a quick look at what some of these elements mean:

Option	Description
LOGIN	The name of the login you want to grant access to for the current database.
CERTIFICATE	Logical name of the certificate to be associated with this user. Note that the certificate must have already been created using the CREATE CERTIFICATE command.
ASYMMETRIC KEY	Logical name of the asymmetric key to be associated with this user. Note that the key must have already been created using the CREATE ASYMMETRIC KEY command.
WITHOUT LOGIN	Creates a user that is local to the current database. It can be used to set up a specific security context but cannot be mapped to a login outside of the current database nor can it access any other database.
WITH DEFAULT_SCHEMA	Establishes a schema other than the default "dbo" as being the default schema for the current user.

sp_grantdbaccess

This is the legacy method for granting a login access to a specific database. The syntax looks like this:

```
sp_grantdbaccess [@loginname =] <'login'>[, [@name_in_db =] <'name in this db'>
```

Note that the access granted will be to the current database—that is, you need to make sure that the database you want the user to have access to is the current database when you issue the command. The login name is the actual login ID that was used to login to SQL Server. The name_in_db parameter allows you to alias this user to another identification. The alias serves for this database only—all other databases will still use the default of the login ID or whatever alias you defined when you granted the user access to that database. The aliasing will affect identification functions such as USER_NAME(). Functions that look at things at the system level, such as SYSTEM_USER, will still return the base login ID.

Granting Object Permissions within the Database

Okay, so the user has a login and access to the database you want him or her to have access to, so now everything's done—right? Ah, if only it were that simple! We are, of course, not done yet.

SQL Server gives us a pretty fine degree of control over what our users can access. Most of the time, you have some information that you want your users to be able to get to, but you also have other information in the database to which you don't want them to have access. For example, you might have a

customer service person who has to be able to look at and maintain order information—but you probably don't want them messing around with the salary information. The opposite is also probably true—you need your human resource people to be able to edit employee records, but you probably don't want them giving somebody a major discount on a sale.

SQL Server allows you to assign a separate set of rights to some of the different objects within SQL Server. The objects you can assign rights to include tables, views, and stored procedures. Triggers are implied to have the rights of the person that created them.

User rights on objects fall into six different types:

User Right	Description
SELECT	Allows a user to “see” the data. If a user has this permission, the user has the right to run a <code>SELECT</code> statement against the table or view on which the permission is granted.
INSERT	Allows a user to create new data. Users with this permission can run an <code>INSERT</code> statement. Note that, unlike many systems, having <code>INSERT</code> capability does not necessarily mean that you have <code>SELECT</code> rights.
UPDATE	Allows a user to modify existing data. Users with this permission can run an <code>UPDATE</code> statement. Like the <code>INSERT</code> statement, having <code>UPDATE</code> capability does not necessarily mean that you have <code>SELECT</code> rights.
DELETE	Allows a user to delete data. Users with this permission can run a <code>DELETE</code> statement. Again, having <code>DELETE</code> capability does not necessarily mean that you have <code>SELECT</code> rights.
REFERENCES	Allows a user to insert rows, where the table that is being inserted into has a foreign key constraint, which references another table to which that a user doesn't have <code>SELECT</code> rights.
EXECUTE	Allows a user to <code>EXECUTE</code> a specified stored procedure.

You can mix and match these rights as needed on the particular table, view, or sproc to which you're assigning rights.

You can assign these rights in Enterprise Manager simply by navigating to the Logins option of the Security node of your server. Just right-click on the user and choose Properties. You'll be presented with a different dialog depending on whether you're in the database or security node, but, in either case, you'll have the option of setting permissions. Assigning rights using T-SQL uses three commands that are good to know even if you're only going to assign rights through EM (the terminology is the same).

GRANT

GRANT gives the specified user or role the access specified for the object that is the subject of the GRANT statement.

Chapter 22

The syntax for a GRANT statement looks like this:

```
GRANT  
    ALL [PRIVILEGES] | <permission>[,...n]  
    ON  
        <table or view name>[(<column name>[,...n])]  
        | <stored or extended stored procedure name>  
    TO <login or role name>[,...n]  
    [WITH GRANT OPTION]  
    [AS <role name>]
```

The ALL keyword indicates that you want to grant all the rights that are applicable for that object type (EXECUTE *never* applies to a table). If you don't use the ALL keyword, then you need to supply one or more specific permissions that you want granted for that object.

PRIVILEGES is a new keyword that has no real function other than to provide ANSI-92 compatibility.

The ON keyword serves as a placeholder to say that what comes next is the object for which you want the permissions granted. Note that, if you are granting rights on a table, you can specify permissions down to the column level by specifying a column list to be affected—if you don't supply specific columns, then it's assumed to affect all columns.

Microsoft appears to have done something of an about face in their opinion of column-level permissions. Being able to say that a user can do a SELECT on a particular table but only on certain columns seems like a cool idea, but it really convolutes the security process both in its use and in the work it takes Microsoft to implement it. As such, recent literature on the subject, plus what I've been told by insiders, seems to indicate that Microsoft wishes that column-level security would go away. They have recommended against its use—if you need to restrict a user to seeing particular columns, consider using a view instead.

The TO statement does what you would expect—it specifies those to whom you want this access granted. It can be a login ID or a role name.

WITH GRANT OPTION allows the user that you're granting access to, in turn, also grant access to other users.

I recommend against the use of this option since it can quickly become a pain to keep track of who has got access to what. Sure, you can always go into Management Studio and look at the permissions for that object, but then you're in a reactive mode rather than a proactive one—you're looking for what's wrong with the current access levels rather than stopping unwanted access up front.

Last, but not least, is the AS keyword. This one deals with the issue of a login belonging to multiple roles.

Now, we can go ahead and move on to an example or two. We'll see later that the TestAccount that we created already has some access based on being a member of the Public role—something that every database user belongs to, and from which you can't remove them. There are, however, a large number of items to which TestAccount doesn't have access (because Public is the only role it belongs to, and Public doesn't have rights either).

Start by logging in with the TestAccount user. Then try a SELECT statement against the Region table:

```
SELECT * FROM Region
```

You'll quickly get a message from SQL Server telling you that you are a scoundrel, and you are attempting to go to places that you shouldn't be going:

```
Server: Msg 229, Level 14, State 5, Line 1
SELECT permission denied on object 'Region', database 'NorthwindSecure', owner
'dbo'.
```

Log in separately as sa—you can do this in the same instance of QA if you like by choosing the File→Connect menu choice. Then select SQL Server security for the new connection and log in as sa with the appropriate password. Now execute a GRANT statement:

```
USE NorthwindSecure
GRANT SELECT ON Region TO [ARISTOTLE\TestAccount]
```

Now switch back to the TestAccount connection (remember, the information for what user you're connected in as is in the Title Bar of the connection window), and try that SELECT statement again: This time, you get better results:

RegionID	RegionDescription
1	Eastern
2	Western
3	Northern
4	Southern

(4 row(s) affected)

Let's go ahead and try another one. This time, let's run the same tests and commands against the EmployeeTerritories table:

```
SELECT * FROM EmployeeTerritories
```

This one fails—again, you don't have rights to it, so let's grant the rights to this table:

```
USE NorthwindSecure
GRANT SELECT ON EmployeeTerritories TO [ARISTOTLE\TestAccount]
```

Now, if you re-run the select statement, things work just fine:

EmployeeID	TerritoryID
1	06897
1	19713
...	
...	
...	
9	48304

Chapter 22

```
9           55113  
9           55439  
  
(49 row(s) affected)
```

To add an additional twist, however, let's try an `INSERT` into this table:

```
INSERT INTO EmployeeTerritories  
VALUES  
(1, '01581')
```

SQL Server wastes no time in telling us to get lost—we don't have the required permissions, so let's grant them (using the `sa` connection):

```
USE NorthwindSecure  
GRANT INSERT ON EmployeeTerritories TO [ARISTOTLE\TestAccount]
```

Now try that `INSERT` statement again:

```
INSERT INTO EmployeeTerritories  
VALUES  
(1, '01581')
```

Everything works great.

DENY

`DENY` explicitly prevents the user from the access specified on the targeted object. The key to `DENY` is that it overrides any `GRANT` statements. Since a user can belong to multiple roles (discussed shortly), it's possible for a user to be part of a role that's granted access but also have a `DENY` in affect. If a `DENY` and a `GRANT` both exist in a user's mix of individual and role-based rights, then the `DENY` wins every time. In short, if the user or any role the user belongs to has a `DENY` for the right in question, then the user will not be able to make use of that access on that object.

The syntax looks an awful lot like the `GRANT` statement:

```
DENY  
ALL [PRIVILEGES] |<permission>[,...n]  
ON  
<table or view name>[(column[,...n])]  
|<stored or extended stored procedure name>  
TO <login ID or roll name>[,...n]  
[CASCADE]
```

Again, the `ALL` keyword indicates that you want to deny all the rights that are applicable for that object type (`EXECUTE` *never* applies to a table). If you don't use the `ALL` keyword, then you need to supply one or more specific permissions that you want to be denied for that object.

`PRIVILEGES` is still a new keyword and has no real function other than to provide ANSI-92 compatibility.

The `ON` keyword serves as a placeholder to say that what comes next is the object on which you want the permissions denied.

Everything has worked pretty much the same as with a `GRANT` statement until now. The `CASCADE` keyword matches up with the `WITH GRANT OPTION` that was in the `GRANT` statement. `CASCADE` tells SQL Server that you want to also deny access to anyone that this user has granted access to under the rules of the `WITH GRANT OPTION`.

To run an example on `DENY`, let's try a simple `SELECT` statement using the `TestAccount` login:

```
USE NorthwindSecure
SELECT * FROM Employees
```

This should get you nine records or so. How did you get access when we haven't granted it to `TestAccount`? `TestAccount` belongs to `Public`, and `Public` has been granted access to `Employees`.

Let's say that we don't want `TestAccount` to have access. For whatever reason, `TestAccount` is the exception, and we don't want that user snooping in that data—we just issue our `DENY` statement (remember to issue the `DENY` using the `sa` login):

```
USE NorthwindSecure
DENY ALL ON Employees TO [ARISTOTLE\TestAccount]
```

When you run the `SELECT` statement again using `TestAccount`, you'll get an error—you no longer have access. Note also that, since we used the `ALL` keyword, the `INSERT`, `DELETE`, and `UPDATE` access that `Public` has is now also denied from `TestAccount`.

Note that `DENY` is new to SQL Server 7.0. The concept of a deny was there in 6.5, but it was implemented differently. Instead of `DENY`, you would issue a `REVOKE` statement twice. The new `DENY` keyword makes things much clearer.

REVOKE

`REVOKE` eliminates the effects of a previously issued `GRANT` or `DENY` statement. Think of this one as like a targeted "Undo" statement.

The syntax is a mix of the `GRANT` and `DENY` statements:

```
REVOKE [GRANT OPTION FOR]
    ALL [PRIVILEGES] | <permission>[,...n]
    ON
    <table or view name>[(column name [,...n])]
        |<stored or extended stored procedure name>
    TO | FROM <login ID or roll name>[,...n]
    [CASCADE]
    [AS <role name>]
```

The explanations here are virtually identical to those of the `GRANT` and `DENY` statements—I put them here again in case you're pulling the book back off the shelf for a quick lookup on `REVOKE`.

Chapter 22

Once again, the `ALL` keyword indicates that you want to revoke all the rights that are applicable for that object type. If you don't use the `ALL` keyword, then you need to supply one or more specific permissions that you want to be revoked for that object.

`PRIVILEGES` still has no real function other than to provide ANSI-92 compatibility.

The `ON` keyword serves as a placeholder to say that what comes next is the object on which you want the permissions revoked.

The `CASCADE` keyword matches up with the `WITH GRANT OPTION` that was in the `GRANT` statement. `CASCADE` tells SQL Server that you want also revoke access from anyone that this user granted access to under the rules of the `WITH GRANT OPTION`.

The `AS` keyword again just specifies which role you want to issue this command based on.

Using the `sa` connection, let's undo the access that we granted to the `Region` table in `NorthwindSecure`:

```
REVOKE ALL ON Region FROM [ARISTOTLE\TestAccount]
```

After executing this, our `TestAccount` can no longer run a `SELECT` statement against the `Region` table.

In order to remove a `DENY`, we also issue a `REVOKE` statement. This time, we'll regain access to the `Employees` table:

```
USE NorthwindSecure  
REVOKE ALL ON Employees TO [ARISTOTLE\TestAccount]
```

Now that we've seen how all the commands to control access work for individual users, let's take a look at the way we can greatly simplify management of these rights by managing in groupings.

User Rights and Statement-Level Permissions

User permissions don't just stop with the objects in your database—they also extend to certain statements that aren't immediately tied to any particular object. SQL Server gives you control over permissions to run several different statements, including:

- CREATE DATABASE
- CREATE DEFAULT
- CREATE PROCEDURE
- CREATE RULE
- CREATE TABLE
- CREATE VIEW
- BACKUP DATABASE
- BACKUP LOG

At this point, we've already seen all of these commands at work except for the two backup commands—what those are about is pretty self-explanatory, so I'm not going to spend any time on them here (we'll look at them in Chapter 24)—just keep in mind that they are something you can control at the statement level.

Okay, so how do we assign these permissions? Actually, now that you've already seen GRANT, REVOKE, and DENY in action for objects, you're pretty much already schooled on statement-level permissions, too. Syntactically speaking, they work just the same as object-level permissions, except that they are even simpler (you don't have to fill in as much). The syntax looks like this:

```
GRANT <ALL | statement[, . . . n]> TO <login ID>[, . . . n]
```

Easy, hey? To do a quick test, let's start by verifying that our test user doesn't already have authority to CREATE. Make sure you are logged in as your TestAccount, and then run the following command (don't forget to switch your domain name for ARISTOTLE in the following):

```
USE NorthwindSecure

CREATE TABLE TestCreate
(
    Col1 int Primary Key
)
```

This gets us nowhere fast:

```
Server: Msg 262, Level 14, State 1, Line 2
CREATE TABLE permission denied, database 'NorthwindSecure', owner 'dbo'.
```

Now log in to SQL Server using the sa account (or another account with dbo authority for NorthwindSecure). Then run our command to grant permissions:

```
GRANT CREATE TABLE TO [ARISTOTLE\TestAccount]
```

You should get confirmation that your command completed successfully. Then just try running the CREATE statement again (remember to log back in using the TestAccount):

```
USE NorthwindSecure

CREATE TABLE TestCreate
(
    Col1 int Primary Key
)
```

This time everything works.

DENY and REVOKE also work the same way as they did for object-level permissions.

Server and Database Roles

Prior to version 7.0, SQL Server had the concept of a *group*—which was a grouping of user rights that you could assign all at once by simply assigning the user to that group. This was quite different from the

Chapter 22

way Windows groups operated, where a user could belong to more than one Windows group, so you could mix and match them as needed. In SQL Server 6.5 (and earlier) only one user was allowed to belong to one group per database.

The fallout from the pre-SQL Server 7.0 way of doing things was that SQL Server groups fell into one of three categories:

- They were frequently modified by user-level permissions.
- They were only a slight variation off the main group.
- They had more access than required (to make the life of the DBA easier).

Basically, they were one great big hassle, albeit a rather necessary one.

Along came version 7.0, and with it some very big changes. Instead of a group, a user now belongs to a *role*. A role is, in the most general sense, the same thing as a group.

A role is a collection of access rights that can be assigned to a user en masse simply by assigning a user to that role.

The similarities begin to fade there though. With roles, a user can belong to several at one time. This can be incredibly handy since you can group access rights into smaller and more logical groups and then mix and match them into the formula that best fits a user.

Roles fall into two categories:

- Server roles
- Database roles

We'll soon see a third thing that's also called role — though I wish that Microsoft had chosen another name — application roles. These are a special way to alias a user into a different set of permissions. An application role isn't something you assign a user to; it's a way of letting an application have a different set of rights from the user. For this reason, I don't usually think of application roles as a "role" in the true sense of the word.

Server roles are limited to those that are already built into SQL Server when it ships and are primarily there for the maintenance of the system as well as granting the capability to do non-database-specific things like creating login accounts and creating linked servers.

Much like server roles, there are a number of built-in (or “fixed”) database roles, but you can also define your own database roles to meet your particular needs. Database roles are for setting up and grouping specific user rights within a single given database.

Let's look at both of these types of roles individually.

Server Roles

All server roles available are “fixed” roles and are there right from the beginning—all the server roles that you’re ever going to have existed from the moment your SQL Server was installed.

Role	Nature
sysadmin	<p>This role can perform any activity on your SQL Server. Anyone with this role is essentially the sa for that server. The creation of this server role provides Microsoft with the capability to one day eliminate the sa login—indeed, the Books Online refers to sa as being legacy in nature.</p> <p>It’s worth noting that the Windows Administrators group on the SQL Server is automatically mapped into the sysadmin role. This means that anyone who is a member of your server’s Administrators group also has sa-level access to your SQL data. You can, if you need to, remove the Windows administrators group from the sysadmin role to tighten that security loophole.</p>
serveradmin	<p>This one can set server-wide configuration options or shut down the server. It’s rather limited in scope, yet the functions controlled by members of this role can have a very significant impact on the performance of your server.</p>
setupadmin	<p>This one is limited to managing linked servers and startup procedures.</p>
securityadmin	<p>This one is very handy for logins that you create specifically to manage logins, read error logs, and CREATE DATABASE permissions. In many ways, this one is the classic system operator role—it can handle most of the day-to-day stuff, but doesn’t have the kind of global access that a true omnipotent superuser would have.</p>
processadmin	<p>Has the capability to manage processes running in SQL Server—this one can kill long-running processes if necessary.</p>
dbcreator	<p>Is limited to creating and altering databases.</p>
diskadmin	<p>Manages disk files (what file group things are assigned to, attaching and detaching databases, etc.).</p>
bulkadmin	<p>This one is something of an oddity. It is created explicitly to give rights to execute the BULK INSERT statement, which otherwise is executable only by someone with sysadmin rights. Frankly, I don’t understand why this statement isn’t granted with the GRANT command like everything else, but it isn’t. Keep in mind that, even if a user has been added to the bulkadmin group, that just gives them access to the statement, not the table that they want to run it against. This means that you need, in addition to adding the user to the bulkadmin task, GRANT them INSERT permissions to any table you want them to be able to perform the BULK INSERT against. In addition, you’ll need to make sure they have proper SELECT access to any tables that they will be referencing in their BULK INSERT statement.</p>

You can mix and match these roles to individual users that are responsible for administrative roles on your server. In general, I suspect that only the very largest of database shops will use more than the sysadmin and securityadmin roles, but they’re still handy to have around.

Earlier in this chapter, I got into a lengthy soapbox diatribe on the evils of global users. It probably comes as no surprise to you to learn that I was positively ecstatic when the new sysadmin role was added back in version 7.0. The existence of this role means that, on an ongoing basis, you should not need to have anyone have the sa login—just let the users that need that level of access become members of the sysadmin role, and they shouldn't ever need to log in as sa.

Database Roles

Database roles are limited in scope to just one database—just because a user belongs to the db_datareader role in one database doesn't mean that it belongs to that role in another database. Database roles fall into two subcategories: fixed and user defined.

Fixed Database Roles

Much as there are several fixed server roles, there are also a number of fixed database roles. Some of them have a special predefined purpose, which cannot be duplicated using normal statements (that is you cannot create a user-defined database role that had the same functionality). However, most exist to deal with the more common situations and make things easier for you.

Role	Nature
db_owner	This role performs as if it were a member of all the other database roles. Using this role, you can create a situation where multiple users can perform the same functions and tasks as if they were the database owner.
db_accessadmin	Performs a portion of the functions similar to the securityadmin server role, except this role is limited to the individual database where it is assigned and the creation of users (not individual rights). It cannot create new SQL Server logins, but members of this role can add Windows users and groups as well as existing SQL Server logins into the database.
db_datareader	Can issue a SELECT statement on all user tables in the database.
db_datawriter	Can issue INSERT, UPDATE, and DELETE statements on all user tables in the database.
db_ddladmin	Can add, modify, or drop objects in the database.
db_securityadmin	The other part of the database-level equivalent of the securityadmin server role. This database role cannot create new users in the database, but does manage roles and members of database roles as well as manage statement and object permissions in the database.
db_backupoperator	Backs up the database (gee, bet you wouldn't have guessed that one!).
db_denydatareader	Provides the equivalent of a DENY SELECT on every table and view in the database.
db_denydatawriter	Similar to db_denydatareader, only affects INSERT, update, and delete statements.

Much as with the fixed server roles, you're probably not going to see all of these used in anything but the largest of database shops. Some of the roles are not replaceable with your own database roles, and others are just very handy to deal with the quick-and-dirty situations that seem to frequently come up.

User-Defined Database Roles

The fixed roles that are available are really only meant to be there to help you get started. The real main-stay of your security is going to be the creation and assignment of user-defined database roles. For these roles, you decide what permissions they include.

With user-defined roles, you can GRANT, DENY, and REVOKE in exactly the same way as we did for individual users. The nice thing about using roles is that users tend to fall into categories of access needs—by using roles you can make a change in one place and have it propagate to all the similar users (at least the ones that you have assigned to that role).

Creating a User-Defined Role

To create our own role, we use the `sp_addrole` system sproc. The syntax is pretty simple:

```
sp_addrole [@rolename =] <'role name'>
[,[@ownername =] <'owner'>]
```

The `role name` is simply what you want to call that role. Examples of common naming schemas would include by department (`Accounting`, `Sales`, `Marketing`, etc.) or by specific job (`CustomerService`, `Salesperson`, `President`, etc.). Using roles like this can make it really easy to add new users to the system. If your accounting department hires someone new, you can just add him or her to the `Accounting` role (or, if you're being more specific, it might even be the `AccountsPayable` role) and forget it—no researching "What should this person have for rights?"

The `owner` is the same thing as it is for all other objects in the system. The default is the database owner, and I strongly suggest leaving it that way (in other words, just ignore this optional parameter).

Let's go ahead and create ourselves a role:

```
USE NorthwindSecure
EXEC sp_addrole 'OurTestRole'
```

When you execute this, you should get back a nice friendly message telling you that the new role has been added.

Now what we need is to add some value to this role in the form of it actually having some rights assigned to it. To do this, we just use our GRANT, DENY, or REVOKE statements just as we did for actual users earlier in the chapter:

```
USE NorthwindSecure
GRANT SELECT ON Territories TO OurTestRole
```

Anyone who belongs to our role now has SELECT access to the `Territories` table (unless they have a DENY somewhere else in their security information).

At this point, you're ready to start adding users.

Chapter 22

Adding Users to a Role

Having all these roles around is great, but they are of no use if they don't have anyone assigned to them. Adding a user to a role is as simple as using the `sp_addrolemember` system sproc and providing the database name and login ID:

```
sp_addrolemember [@rolename =] <role name>,
[@membername =] <Login ID>
```

Everything is pretty self-explanatory on the parameters for this one, so let's move right into an example.

Let's start off by verifying that our `TestAccount` doesn't have access to the `Territories` table:

```
SELECT * FROM Territories
```

Sure enough, we are rejected (no access yet):

```
Server: Msg 229, Level 14, State 5, Line 1
SELECT permission denied on object 'Territories', database 'Northwind', owner
'dbo'.
```

Now we'll go ahead and add our `TestAccount` Windows user to our `OurTestRole` role:

```
USE NorthwindSecure
EXEC sp_addrolemember OurTestRole, [ARISTOTLE\TestAccount]
```

Again, we get a friendly confirmation that things have worked properly:

```
'ARISTOTLE\TestAccount' added to role 'OurTestRole'.
```

It's time to try and run the `SELECT` statement again—this time with much more success (you should get about 53 rows back).

Removing a User from a Role

What goes up must come down, and users that are added to a role will also inevitably be removed from roles.

Removing a user from a role works almost exactly as adding them does, except we use a different system sproc called `sp_dropprolemember` in the form of:

```
sp_dropprolemember [@rolename =] <role name>,
[@membername =] <security account>
```

So, let's go right back to our example and remove the `TestAccount` from the `OurTestRole` database role:

```
USE NorthwindSecure
EXEC sp_dropprolemember OurTestRole, [ARISTOTLE\TestAccount]
```

You should receive another friendly confirmation that things have gone well. Now try our SELECT statement again:

```
SELECT * FROM Territories
```

And, sure enough, we are again given the error that we don't have access.

You can add and drop users from any role this way — it doesn't matter whether the role is user-defined or fixed, or whether it's a system or database role. In any case, they work pretty much the same.

Note also that you can do all of this through Enterprise Manager. To change the rights associated with a role, just click on the Roles member of the Databases node, and assign permissions by using the check-boxes. When you want to add a user to the role, just go to the user properties, select either the Server or Database roles tab, and then put a check mark in all the roles you want that user to have.

Dropping Roles

Dropping a role is as easy as adding one. The syntax is simply:

```
EXEC sp_droprole <'role name'>
```

And it's gone.

Application Roles

Application roles are something of a different animal than are database and server roles. Indeed, the fact that the term *role* is used would make you think that they are closely related — they aren't.

Application roles are really much more like a security alias for the user. Application roles allow you to define an access list (made up of individual rights or groupings of databases). They are also similar to a user in that they have their own password. They are, however, different from a user login because they cannot "log in" as such — a user account must first log in, then he or she can activate the application role.

So what do we need application roles for? For applications — what else? Time and time again, you'll run into the situation where you would like a user to have a separate set of rights depending on under what context he or she is accessing the database. With an application role, you can do things like grant users no more than read-only access to the database (SELECT statements only), but still allow them to modify data when they do so within the confines of your application.

Note that application roles are a one-way trip — that is, once you've established an application role as being active for a given connection, you can't go back to the user's own security settings for that connection. In order for users to go back to their own security information, they must terminate the connection and log in again.

The process works like this:

1. The user logs in (presumably using a login screen provided by your application).
2. The login is validated, and the user receives his or her access rights.

Chapter 22

3. The application executes a system sproc called `sp_setapprole` and provides a role name and password.
4. The application role is validated, and the connection is switched to the context of that application role (all the rights the user had are gone—he or she now has the rights of the application role).
5. The user continues with access based on the application role rather than his or her personal login throughout the duration of the connection—the user cannot go back to his or her own access information.

You would only want to use application roles as part of a true application situation, and you would build the code to set the application role right into the application. You would also compile the required password into the application or store the information in some local file to be accessed when it is needed.

Creating Application Roles

To create an application role, we use a new system sproc called `sp_addapprole`. This is another pretty easy one to use; its syntax looks like this:

```
sp_addapprole [@rolename =] <role name>,
    [@password =] <'password'>
```

Much like many of the sprocs in this chapter, the parameters are pretty self-explanatory; so let's move right on to using it by creating ourselves an application role:

```
EXEC sp_addapprole OurAppRole, 'password'
```

Just that quick, our application role is created.

Adding Permissions to the Application Role

Adding permissions to application roles works just like adding permissions to anything else. Just substitute the application role name anywhere that you would use a login ID or regular server or database role.

Again, we'll move to the quick example:

```
GRANT SELECT ON Region TO OurAppRole
```

Our application role now has `SELECT` rights on the `Region` table—it doesn't, as yet, have access to anything else.

Using the Application Role

Using the application role is a matter of calling a system sproc (`sp_setapprole`) and providing both the application role name and the password for that application role. The syntax looks like this:

```
sp_setapprole [@rolename =] <role name>,
    [@password =] {Encrypt N'password'} | 'password'
    [,[@encrypt =] '<encryption style>']
```

The `role name` is simply the name of whatever application role you want to activate.

The password can be either supplied as is or encrypted using the ODBC encrypt function. If you're going to encrypt the password, then you need to enclose the password in quotes after the `Encrypt` keyword and precede the password with a capital N—this indicates to SQL Server that you're dealing with a Unicode string, and it will be treated accordingly. Note the use of a curly braces {} rather than parentheses for the encryption parameter. If you don't want encryption, then just supply the password without using the `Encrypt` keyword.

The `encryption style` is needed only if you chose the encryption option for the password parameter. If you are encrypting, then supply "ODBC" as the encryption style.

It's worth noting that encryption is only an option with ODBC and OLE DB clients. You cannot use DB-Lib with encryption.

Moving right into the example category, let's start by verifying a couple of things about the status of our `TestAccount` user. At this point in the chapter (assuming you've been following along with all the examples), your `TestAccount` user should not be able to access the `Region` table but should be able to access the `EmployeeTerritories` table. You can verify this to be the case by running a couple of `SELECT` statements:

```
SELECT * FROM Region
SELECT * FROM EmployeeTerritories
```

The first `SELECT` should give you an error, and the second should return around 50 rows or so.

Now let's activate the application role that we created a short time ago, type this in using `TestAccount` user:

```
sp_setapprole OurAppRole, {Encrypt N'password'}, 'odbc'
```

When you execute this, you should get back a confirmation that your application role is now "active."

Try it out by running our two `SELECT` statements—you'll find that what does and doesn't work has been exactly reversed. That is, `TestAccount` had access to `EmployeeTerritories`, but that was lost when we went to the application role. `TestAccount` did not have access to the `Regions` table, but the application role now provides that access.

There is no way to terminate the application role for the current connection, so you can go ahead and terminate your `TestAccount` connection. Then, create a new connection with Windows security for your `TestAccount`. Try running those `SELECT` statements again, and you'll find that your original set of rights has been restored.

Getting Rid of Application Roles

When you no longer need the application role on your server, you can use `sp_dropapprole` to eliminate it from the system. The syntax is as follows:

```
sp_dropapprole [@rolename =] <role name>
```

To eliminate our application role from the system, we would just issue the command (from sa):

```
EXEC sp_dropapprole OurAppRole
```

More Advanced Security

This section is really nothing more than an “extra things to think about” section. All of these fall outside the realm of the basic rules we defined at the beginning of the chapter, but they address ways around some problems and also how to close some common loopholes in your system.

What to Do about the guest Account

The guest account provides a way of having default access. When you have the guest account active, several things happen:

- ❑ Logins gain guest-level access to any database to which they are not explicitly given access.
- ❑ Outside users can log in through the guest account to gain access. This requires that they know the password for guest, but they’ll already know the user exists (although, they probably also know that the sa account exists).

Personally, one of the first things I do with my SQL Server is to eliminate every ounce of access the guest account has. It’s a loophole, and it winds up providing access in a way you don’t intuitively think of. (You probably think that when you assign rights to someone—that’s all the rights they have. With guest active, that isn’t necessarily so.) I recommend that you do the same.

There is, however, one use that I’m aware of where the guest account actually serves a fairly slick purpose—when it is used with application roles. In this scenario, you leave the guest account with access to a database but without any rights beyond simply logging into that database—that is the guest account only makes the logged on database “current.” You can then use sp_setapprole to activate an application role, and, boom, you now have a way for otherwise anonymous users to log in to your server with appropriate rights. They can, however, only perform any *useful* login if they are using your application.

This is definitely a scenario where you want to be protecting that application role password as if your job depended on it (it probably does). Use the ODBC encryption option—particularly if your connection is via the Internet!

TCP/IP Port Settings

By default when using TCP/IP, SQL Server uses port number 1433. A port can be thought of as something like a radio channel—it doesn’t matter what channel you’re broadcasting on, it won’t do you any good if no one is listening to that channel.

Leaving things with the default value of 1433 can be very convenient—all of your clients will automatically use port 1433 unless you specify otherwise, so this means that you have one less thing to worry about being set right if you just leave well enough alone.

The problem, however, is that just about any potential SQL Server hacker also knows that port 1433 is the one to which 99 percent of all SQL Servers are listening. If your SQL Server has a direct connection to the Internet, I strongly recommend changing to a non-standard port number — check with your network administrator for what he or she recommends as an available port. Just remember that, when you change what the server is “listening” to, you’ll also need to change what all the IP-based clients are using. For example, if we were going to change to using port 1402, we would go into the Client Network Utility and set up a specific entry for our server with 1402 as the IP port to use.

Beginning with SQL Server 2000, we also have the option of telling the client to dynamically determine the port, by checking the Dynamically determine port box.

Don’t Use the sa Account

Everyone who’s studied SQL Server for more than about 10 minutes knows about the system administrator account. Now that SQL Server has the `sysadmin` fixed server role, I strongly suggest adding true logins to that role, then changing the `sa` password to something very long and very incomprehensible — something not worth spending the time to hack into. If you only need Windows security, then turn SQL Server security off, and that will deal with the `sa` account issue once and for all.

Keep `xp_cmdshell` under Wraps

Remember to be careful about who you grant access to use `xp_cmdshell`. It will run any DOS or Windows command prompt command. The amount of authority that it grants to your users depends on what account SQL Server is running under. If it is a system account (as the majority are), then the users of `xp_cmdshell` will have very significant access to your server (they could, for example, copy files onto the server from elsewhere on the network, then execute those files). Let’s raise the stakes a bit though — there are also a fair number of servers running out there under the context of a Windows domain administrator account — anyone using `xp_cmdshell` now has fairly open access to your entire network!!!

The short rendition here is not to give anyone access to `xp_cmdshell` that you wouldn’t give administrative rights to for your server or possibly even your domain.

Don’t Forget Views, Stored Procedures, and UDFs as Security Tools

Remember that views, sprocs, and UDFs all have a lot to offer in terms of hiding data. Views can usually take the place of column-level security. They can do wonders to make a user think they have access to an entire table, when they, in reality, have access to only a subset of the entire data (remember our example of filtering out sensitive employee information, such as salary?). Sprocs and UDFs can do much the same — you can grant execute rights to a sproc or UDF, but that doesn’t mean users get all the data from a table (they only get what the sproc or UDF gives them) — the end user may not even know what underlying table is supplying the data. In addition, views, sprocs, and UDFs have their own implied authority — that is, just because views and sprocs use a table, it doesn’t mean that the user has access rights for that table.

Certificates and Asymmetric Keys

We have, at a few different points in the book (including earlier in this chapter), mentioned the notion of encryption. Certificates and asymmetric keys are the primary mechanism for defining the encryption keys for the different levels of your server architecture. Both of these are different methods of doing the same basic thing, and they are largely interchangeable. Whether you use certificates or asynchronous keys, you need to keep in mind that these are much like the keys to your house—if you let everyone have them, then they quickly lose their value (now anyone can get in, so why bother locking anyone out?).

SQL Server supports the notion of keys at several different levels based on the notion that you may want to separate several different silos of control under different encryption keys. SQL Server maintains a *Service Master Key* that goes with each server installation. It is encrypted by the Windows-level Service Master Key. Likewise, each database contains a *Database Master Key*, which can, if you choose, itself be encrypted based on the Service Master Key. Then, within each database, you can define certificates and/or asymmetric keys (both of which are a form of key). Overall, the hierarchy looks something like Figure 22-3.

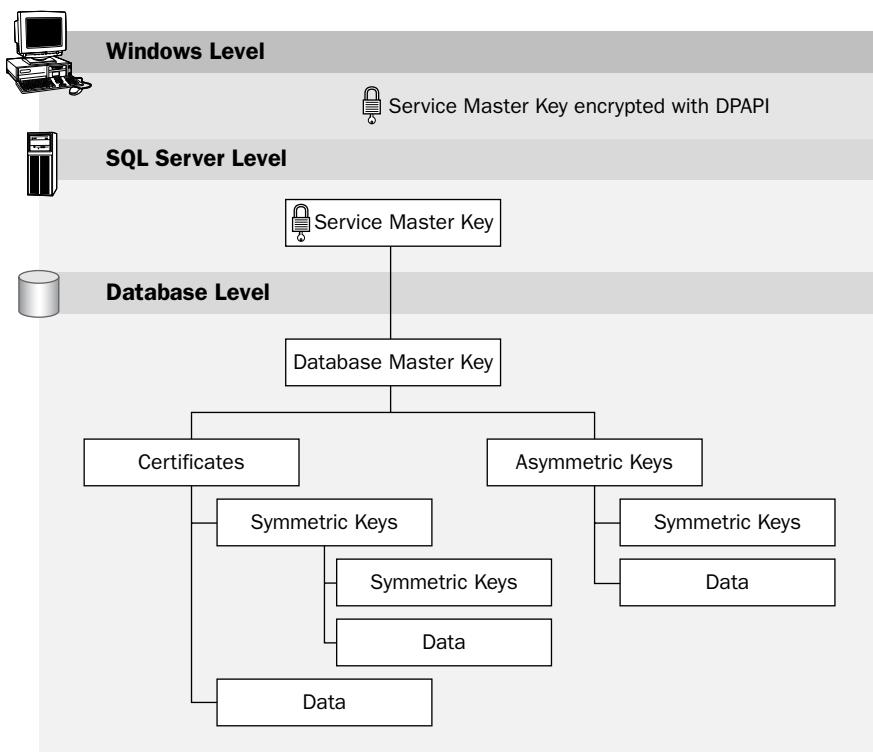


Figure 22-3

Certificates

Since SQL Server 2000, SQL Server has included its own *certificate authority*, or CA. Third-party CAs are also supported. A CA issues a certificate, which includes an encryption key along with some basic information to go with the certificate such as what date range the certificate is valid for (a starting and expiration date), the name of the holder, and information on the authority that issued the certificate. A certificate is added to a server using the `CREATE CERTIFICATE` command.

Asymmetric Keys

An asymmetric key works much as a certificate does but is specified directly and is not validated by any issuing authority. Like a certificate, the encryption key is specified and then utilized to encrypt sensitive information. Asymmetric keys are added using the `CREATE ASYMMETRIC KEY` command.

Summary

Security is one of those areas that tend to be ignored by developers. Unfortunately, the security of your system is going to be determined by how your client application handles things, so there's only so much a DBA can do after you've shipped your application.

Treat security as if it is the lifeblood for the success or failure of your system at your customer site (which, if you're building internal projects, may be your site)—it probably is a critical factor.

23

Playing a Good Tune: Performance Tuning

This is probably the toughest chapter in the book from my perspective as the author, but not for the normal reasons. Usually, the issue is how to relate complex information in a manner that's easy to understand. As we're getting near the end of the book, I hope that I've succeeded there—even if there is still more to come. At this point, you have a solid foundation in everything we're going to discuss in this chapter. That means I'm relatively free to get to the nitty-gritty and not worry quite as much about confusion.

Why then would this be a tough chapter for me to write? Well, because deciding exactly what to put into this chapter is difficult. You see, this isn't a book on performance tuning—that can easily be a book unto itself. It is, however, a book about making you successful in your experience developing with SQL Server. Having a well-performing system is critical to that success. The problem lies in a line from Bob Seger: "What to leave in, what to leave out." What can we focus on here that's going to get you the most bang for your buck?

Perhaps the most important thing to understand about performance tuning is that you are never going to know everything there is to know about it. If you're the average SQL developer, you're going to be lucky if you know 20 percent of what there is to know. Fortunately, performance tuning is one of those areas where the old 80-20 rule (80 percent of the benefit comes from the right 20 percent of the work) definitely applies.

With that in mind, we're going to be roaming around quite a bit in this chapter topically speaking. Everything we talk about is going to be performance related in some fashion, but we'll touch on a wide range of ways to squeeze the most out of the system performance-wise. The topics we'll go into include both new and old subjects. In many cases, it will be a subject we've already covered, but with a particular eye on performance. Some of the places we'll focus on include:

- Index choices
- Client vs. server-side processing

Chapter 23

- ❑ Strategic de-normalization
- ❑ Routine maintenance
- ❑ Organizing your sprocs
- ❑ Uses for temporary tables
- ❑ Small gains in repetitive processes vs. big gains in long-running processes
- ❑ Hardware configuration issues
- ❑ Troubleshooting

Even though we're going to touch on these, there is a more important concept to be sure that you get—this is only the beginning. The biggest thing in performance is really just to stop and think about it. There is, for some strange reason, a tendency when working with SQL to just use the first thing that comes to mind that will work. You need to give the same kind of thought to your queries, sprocs, database designs—whatever—that you would give to any other development work that you're doing. Also, keep in mind that your T-SQL code is only one part of the picture—hardware, client code, SQL Server configuration, and network issues are examples of things that are “outside the code” that can have a dramatic impact on your system.

Performance means a lot of different things to a lot of different people. For example, many will think in terms of simple response time (how fast does my query finish). There is also the notion of *perceived* performance (many users will think in terms of how fast they receive enough to start working on, rather than how fast it actually finishes). Yet another perspective might focus on scalability (for example, how much load can I put on the system before my response time suffers or until users start colliding with each other?).

Many of the examples and suggestions in this chapter are about raw speed—how fast do I return results—we do, however, touch on perceived performance and scalability issues where appropriate. Make sure that all facets of performance are considered in your designs—not just time to completion.

When to Tune

Okay, so this is probably going to seem a little obvious, but performance starts much earlier in the process than when you are writing your code. Indeed, it really should start in the requirements-gathering process and then never end.

What's the big deal about performance tuning in the requirements-gathering stage? Well, while you obviously can't do anything yet to *physically* tune your system, you can do a lot to *logically* tune your system. For example, is the concern of the customer more toward the side of *perceived* performance or actual completion of the job? For interactive processes, users will generally be more satisfied and *think* the system is faster if you do something to show them that something is happening (even if it's just a progress bar). In addition, sometimes it's worth having a process that completes a little more slowly as long as the

“first response”—that is, when it starts outputting something—is faster. Which of these is preferable is something you should know in the requirements-gathering stage. Finally, you should, in the requirements-gathering process, determine what your performance requirements are for the system.

Many is the time that I have seen the system that the developer thought was “fast enough” only to find out that the performance was unacceptable to the user. This can happen for a lot of reasons, though the most common is certainly the developer having his or her head buried in the sand.

Find out what’s expected! Also, remember to test whether you’ve met expectations under a realistic load on something resembling the real live hardware—not a load based on one or two developers sitting at their development system.

Performance obviously also continues into design. If you design for performance, then you will generally greatly reduce the effort required to tune at completion. What’s more, you’ll find that you’ve greatly enhanced what the “best” numbers you can achieve are.

I’m starting to drone on here, but performance never stops—when you’re actually coding, get it working, but then STOP! Stop and take a look at your code. Once an entire system is together, the actual code will almost never be looked at again unless:

- Something breaks (there’s a bug).
- You need to upgrade that part of the system.
- There is an overt performance problem (usually, a *very* bad one).

In the first two of these instances, you probably won’t be looking at the performance issues, just how to get things fixed or the additional functionality added. The point here is that an extra few minutes of looking at your code and asking yourself “Could I have done it better?” or “Hey, have I done anything stupid here?” can shave a little bit here and a little bit there and, occasionally, a whole lot in some other place.

Simply put: I make stupid mistakes, and so will you. It is, however, amazing how often you can step back from your code for a minute or two, then look at it again with a critical eye and say, “Geez, I can’t believe I did that!” Hopefully, those moments will be rare, but, if you take the time to be critical of your own code, you’ll find most of those critical gaffs that could really bog your system down. As for the ones you don’t find, hopefully you’ll turn those up as you continue down the path we’re laying out here.

The next big testing milestone time is in the quality assurance process. At this juncture you should be establishing general system benchmarks and comparing those against the performance requirements established during the requirements phase.

Last, but not least—never stop. Ask end users where their pain is from performance perspective. Is there something they say is slow? Don’t wait for them to tell you (often, they think “that’s just the way it is” and say nothing—except to your boss, of course); go ask.

Index Choices

Again, this is something that was covered in extreme depth previously, but the topic still deserves something more than a mention here because of its sheer importance to query performance.

Chapter 23

People tend to go to extremes with indexes—I’m encouraging you not to follow any one rule but to instead think about the full range of items that your index choices impact.

Any table that has a primary key (and with very rare exception, all tables should have a primary key) has at least one index. This doesn’t mean, however, that it is a very useful index from a performance perspective. Indexes should be considered for any column that you’re going to be frequently using as a target in a WHERE or JOIN, and, to a lesser extent, an ORDER BY clause.

Remember though, that the more indexes you have, the slower that your inserts are going to be. When you insert a record, one or more entries (depending on what’s going on in the non-leaf levels of the B-Tree) have to be made for that index. That means more indexes and also more for SQL Server to do on inserts. In an Online Transaction Processing (OLTP) environment (where you tend to have a lot of inserts, updates, and deletes), this can be a killer. In an Online Analytical Processing (OLAP) environment, this is probably no big deal since your OLAP data is usually relatively stable (few inserts), and what inserts are made are usually done through a highly repetitive batch process (doesn’t have quite the lack of predictability that users have).

Technically speaking, having additional indexes can also have an impact on updates similar to that for inserts. The problem is, however, to a much smaller degree. Your indexes need to be updated only if the column that was changed is part of the key for that index. If you do indeed need to update the index though, think about it as a delete and insert—that means that you’re exposed to page splits again.

So, what then about deletes? Well, again, when you delete a record you’re going to need to go delete all the entries from your indexes too, so you do add some additional overhead. The bright spot is that you don’t have to worry about page splits and having to physically move data around.

The bottom line here is that if you’re doing a lot more querying than modifying, then more indexes are okay. However, if you’re doing lots of modifications to your data keep your indexes limited to high use columns.

If you’re treating this book as more of a reference than a full “learn how” book and haven’t taken the time to read the index chapter (Chapter 8) yet—do it!

Check the Index Tuning Tool in the Database Tuning Advisor

The *Database Tuning Advisor* is a descendant of the Index Tuning Wizard that made its first appearance back in version 7.0. While the Database Tuning Advisor has grown to include much more than just index tuning, it still has this key feature.

Be very careful when using automated tuning tools with indexes. In particular, watch out about what indexes you let it delete. It makes its recommendations based on the workload it has been exposed to—that workload may not include all of the queries that make up your system. Take a look at the recommendations and ask yourself why those recommendations might help. Particularly with deletions, ask yourself what that index might be used for—does deleting it make sense? Is there some long-running report that didn’t run when you were capturing the workload file that might make use of that index?

Client vs. Server-Side Processing

Where you decide to “do the work” can have a very serious impact—for better or worse—on overall system performance.

When client/server computing first came along, the assumption was that you would get more/faster/cheaper by “distributing” the computing. For some tasks, this is true. For others though, you lose more than you gain.

Here’s a quick review of some preferences as to which end to do things:

Static cursors	Usually much better on the client. Since the data isn’t going to change, you want to package it up and send it all to the client in one pass—thus limiting roundtrips and network impact. The obvious exception is if the cursor is generated for the sole purpose of modifying other records. In such a case, you should try and do the entire process at the server-side (most likely in the form of a stored procedure)—again eliminating roundtrips.
Forward-only, read-only cursors	Client-side again. The ODBC libraries can take special advantage of the FAST_FORWARD cursor type to gain maximum performance. Just let the server spew the records into the client cursor, and then move on with life.
HOLDLOCK situations	Most transactioning works much better on the server than on the client.
Processes that require working tables	This is another of those situations where you want to try to have the finished product created before you attempt to move records to the client. If you keep all of the data server-side until it is really ready to be used, you minimize roundtrips to the server and speed up performance.
Minimizing client installations	Okay, so this isn’t “performance” as such, but it can be a significant cost factor. If you want to minimize the number of client installations you have to do, then keep as much of the business logic out of the client as possible. Either perform that logic in sprocs, or look at using component-based development with .NET. In an ideal world, you’ll have what I like to call “data logic” (logic that exists only for the purpose of figuring out how to get the final data) in sprocs and “business logic” in components.
Significant filtering and/or resorting.	Use ADO.NET. It has a great set of tools for receiving the data from the server just once (fewer roundtrips!), then applying filters and sorts locally using the ADO engine. If you wanted the data filtered or sorted differently by SQL Server, it would run an entirely new query using the new criteria. It doesn’t take a rocket scientist to figure out that the overhead on that can get rather expensive. ADO.NET also has some cool things built in to allow you to join different .NET data sets right at the client. Note, however, that with very large result sets, your client computer may not have the wherewithal to deal with the filters and sorts effectively—you may be forced to go back to the server.

Chapter 23

These really just scratch the surface. The big thing to remember is that roundtrips are a killer even in this age of gigabit Ethernet (keep in mind that connection overhead is often more of the issue than raw bandwidth). What you need to do is move the smallest amount of data back and forth—and only move it once. Usually, this means that you'll pre-process the data as much as possible on the server side, and then move the entire result to the client if possible.

Keep in mind, though, that you need to be sure that your client is going to be able to handle what you give it. Servers are usually much better equipped to handle the resource demands of larger queries. By the same token, you also have to remember that the server is going to be doing this for multiple users—that means the server needs to have adequate resources to store all of the server-side activity for that number of users. If you take a process that was too big for the client to handle and move it server-side for resource reasons; just remember that you may also run out of resources on the server, if more than one client goes to use that process at one time. The best thing is to try to keep result sets and processes the smallest size possible.

Strategic De-Normalization

This could also be called, "When following the rules can kill you." Normalized data tends to work for both data integrity and performance in an OLTP environment. The problem is that not everything that goes on in an OLTP database is necessarily transaction processing related. Even OLTP systems have to do a little bit of reporting (a summary of transactions entered that day for example).

Often, adding just one extra column to a table can prevent a large join, or worse, a join involving several tables. I've seen situations where adding one column made the difference between a two-table join and a nine-table join. We're talking the difference between 100,000 records being involved and several million. This one change made the difference in a query dropping from a runtime of several minutes down to just seconds.

Like most things, however, this isn't something with which you should get carried away. Normalization is the way that most things are implemented for a reason. It adds a lot to data integrity and can make a big positive difference performance wise in many situations. Don't de-normalize just for the sake of it. Know exactly what you're trying to accomplish, and test to make sure that it had the expected impact. If it didn't, then look at going back to the original way of doing things.

Routine Maintenance

I hate it when good systems go bad. It happens on a regular basis though. It usually happens when people buy or build systems, put them into operation, and then forget about them.

Maintenance is as much about performance as it is about system integrity. Query plans get out of date, index pages get full (so you have a lot of page splits), fragmentation happens, the best indexes have to have changes as usage and the amount of data in various tables changes.

Watch the newsgroups. Talk to a few people who have older systems running. You'll hear the same story over and over again. "My system used to run great, but it just keeps getting slower and slower—I haven't changed anything, so what happened?" Well, systems will naturally become slower as the

amount of data they have to search over increases; however, the change doesn't have to be all that remarkable and usually it shouldn't be. Instead, the cause is usually that the performance enhancements you put in place when you first installed the system don't really apply anymore, as the way your users use the system and the amount of data has changed, so has the mix of things that will give you the best performance.

We'll be looking at maintenance quite a bit in the next chapter; however, we've discussed it here for two reasons. First, it will help if you are checking out this chapter because you have a specific performance problem. Second, and perhaps more importantly, because there is a tendency to just think about maintenance as being something you do to prevent the system from going down and to ensure backups are available should the worst happen. This simply isn't the case. Maintenance is also a key from a performance perspective.

Organizing Your Sprocs Well

I'm not talking from the outside (naming conventions and such are important, but that's not what I'm getting at here) but rather from a "how they operate" standpoint. The next few sections discuss this.

Keeping Transactions Short

Long transactions cannot only cause deadlock situations but also basic blocking (where someone else's process has to wait for yours because you haven't finished with the locks yet). Any time you have a process that is blocked—even if it will eventually be able to continue after the blocking transaction is complete—you are delaying, and therefore hurting the performance of, that blocked procedure. There is nothing that has a more immediate effect on performance than that a process has to simply stop and wait.

Using the Least Restrictive Transaction Isolation Level Possible

The tighter you hold those locks, the more likely that you're going to wind up blocking another process. You need to be sure that you take the number of locks that you really need to ensure data integrity—but try not to take any more than that.

If you need more information on isolation levels, check out transactions and locks in Chapter 12.

Implementing Multiple Solutions if Necessary

An example here is a search query that accepts multiple parameters but doesn't require all of them. It's quite possible to write your sproc so that it just uses one query, regardless of how many parameters were actually supplied—a "one-size-fits-all" kind of approach. This can be a real timesaver from a development perspective, but it is really deadly from a performance point of view. More than likely, it means that you are joining several unnecessary tables for every run of the sproc!

The thing to do here is add a few `IF . . . ELSE` statements to check things out. This is more of a "look before you leap" kind of approach. It means that you will have to write multiple queries to deal with each possible mix of supplied parameters, but once you have the first one written, the others can often be cloned and then altered from the first one.

This is a real problem area in lots of code out there. Developers are a fickle bunch. We generally only like doing things as long as they are interesting. If you take the preceding example, you can probably see that it would get very boring very quickly to be writing what amounts to a very similar query over and over to deal with the nuances of what parameters were supplied.

All I can say about this is—well, not everything can be fun, or everyone would want to be a software developer! Sometimes you just have to grin and bear it for the sake of the finished product.

Avoiding Cursors if Possible

If you're a programmer who has come from an ISAM or VSAM environment (these were older database storage methods), doing things by cursor is probably going to be something toward which you'll naturally gravitate. After all, the cursor process works an awful lot more like what you're used to in those environments.

Don't go there!

Almost all things that are first thought of as something you can do by cursors can actually be done as a set operation. Sometimes it takes some pretty careful thought, but it usually can be done.

By way of illustration, I was asked several years ago for a way to take a multi-line cursor-based operation and make it into a single statement if possible. The existing process ran something like 20 minutes. The runtime was definitely problematic, but the customer wasn't really looking to do this for performance reasons (they had accepted that the process was going to take that long). Instead, they were just trying to simplify the code.

They had a large product database, and they were trying to set things up to automatically price their available products based on cost. If the markup had been a flat percentage (say 10 percent), then the UPDATE statement would have been easy—say something like:

```
UPDATE Products  
SET UnitPrice = UnitCost * 1.1
```

The problem was that it wasn't a straight markup—there was a logic pattern to it. The logic went something like this:

- If the pennies on the product after the markup are greater than or equal to .50, then price it at .95.
- If the pennies are below .50, then mark it at .49.

The pseudocode to do this by cursor would look something like:

```
Declare and open the cursor  
Fetch the first record  
Begin Loop Until the end of the result set  
    Multiply cost * 1.1  
    If result has cents of < .50  
        Change cents to .49  
    Else  
        Change cents to .95  
    Loop
```

This is, of course, an extremely simplified version of things. There would actually be about 30–40 lines of code to get this done. Instead, we changed it around to work with one single correlated subquery (which had a `CASE` statement embedded in it). The runtime dropped down to something like 12 seconds.

The point here, of course, is that, by eliminating cursors wherever reasonably possible, we can really give a boost to not only complexity (as was the original goal here), but also performance.

Uses for Temporary Tables

The use of temporary tables can sometimes help performance—usually by allowing the elimination of cursors.

As we've seen before, cursors can be the very bane of our existence. Using temporary tables, we can sometimes eliminate the cursor by processing the operation as a series of two or more set operations. An initial query creates a working data set. Then another process comes along and operates on that working data.

We can actually make use of the pricing example we laid out in the last section to illustrate the temporary table concept, too. This solution wouldn't be quite as good as the correlated subquery, but it is still quite workable and much faster than the cursor option. The steps would look something like:

```
SELECT ProductID, FLOOR(UnitCost * 1.1) + .49 AS TempUnitPrice
    INTO #WorkingData
    FROM Products
    WHERE (UnitCost * 1.1) - FLOOR(UnitCost * 1.1) < .50
INSERT INTO #WorkingData
SELECT ProductID, FLOOR(UnitCost * 1.1) + .95 AS TempUnitPrice
    FROM Products
    WHERE (UnitCost * 1.1) - FLOOR(UnitCost * 1.1) >= .50
UPDATE p
    SET p.UnitPrice = t.TempUnitPrice
    FROM Product p
    JOIN #WorkingData t
        ON p.ProductID = t.ProductID
```

With this, we wind up with three steps instead of thirty or forty. This won't operate quite as fast as the correlated subquery would, but it still positively screams in comparison to the cursor solution.

Keep this little interim step using temporary tables in mind when you run into complex problems that you think are going to require cursors. Try to avoid the temptation of just automatically taking this route—look for the single statement query before choosing this option—but if all else fails, this can really save you a lot of time versus using a cursor option.

Sometimes, It's the Little Things

A common mistake in all programming for performance efforts is to ignore the small things. Whenever you're trying to squeeze performance, the natural line of thinking is that you want to work on the long-running stuff.

It's true that the long-running processes are the ones for which you stand the biggest chance of getting big one-time performance gains. It's too bad that this often leads people to forget that it's the total time saved that they're interested in—that is, how much time when the process is really live.

While it's definitely true that a single change in a query can often turn a several-minute query into seconds (I've actually seen a few that took literally days trimmed to just seconds by index and query tuning), the biggest gains for your application often lie in getting just a little bit more out of what already seems like a fast query. These are usually tied to often-repeated functions or items that are often executed within a loop.

Think about this for a bit. Say you have a query that currently takes three seconds to run, and this query is used every time an order taker looks up a part for possible sale—say 5,000 items looked up a day. Now imagine that you are able to squeeze one second off the query time. That's 5,000 seconds, or over and hour and 20 minutes!

Hardware Considerations

Forgive me if I get too bland here—I'll try to keep it interesting, but if you're like the average developer, you'll probably already know enough about this to make it very boring, yet not enough about it to save yourself a degree of grief.

Hardware prices have been falling like a rock over the last few years—unfortunately, so has what your manager or customer is probably budgeting for your hardware purchases. When deciding on a budget for your hardware, remember:

- ❑ Once you've deployed, the hardware is what's keeping your data safe—just how much is that data worth?
- ❑ Once you've deployed, you're likely to have many users—if you're creating a public Web site, it's possible that you'll have tens of thousands of users active on your system 24 hours per day. What is it going to cost you in terms of productivity loss, lost sales, loss of face, and just general credibility loss if that server is unavailable or—worse—you lose some of your data?
- ❑ Maintaining your system will quickly cost more than the system itself. Dollars spent early on a mainstream system that is going to have fewer quirks may save you a ton of money in the long run.

There's a lot to think about when deciding who to purchase from and what specific equipment to buy. Forgetting the budget for a moment, some of the questions to ask yourself include:

- ❑ Will the box be used exclusively as a database server?
- ❑ Will the activity on the system be processor or I/O intensive? (For databases, it's almost always the latter, but there are exceptions.)
- ❑ Am I going to be running more than one production database? If so, is the other database of a different type (OLTP versus OLAP)?
- ❑ Will the server be on-site at my location, or do I have to travel to do maintenance on it?

- What are my risks if the system goes down?
- What are my risks if I lose data?
- Is performance “everything”?
- What kind of long-term driver support can you expect as your O/S and supporting systems are upgraded?

Again, we’re just scratching the surface of things—but we’ve got a good start. Let’s look at what these issues mean to us.

Exclusive Use of the Server

I suppose it doesn’t take a rocket scientist to figure out that, in most cases, having your SQL Server hardware dedicated to just SQL Server and having other applications reside on totally separate system(s) is the best way to go. Note, however, that this isn’t always the case.

If you’re running a relatively small and simple application that works with other subsystems (say IIS as a Web server for example), then you may actually be better off, performance-wise, to stay with one box. Why? Well, if there are large amounts of data going back and forth between the two subsystems (your database in SQL Server and your Web pages or whatever in a separate process), then memory space to memory space communications are going to be much faster than the bottleneck that the network can create—even in a relatively dedicated network backbone environment.

Remember that this is the exception, though, not the rule. The instance where this works best usually meets the following criteria:

- The systems have a very high level of interaction.
- The systems have little to do beyond their interactions (the activity that’s causing all the interaction is the main thing that the systems do).
- Only one of the two processes is CPU intensive and only one is I/O intensive.

If in doubt, go with conventional thinking on this and separate the processing into two or more systems.

I/O vs. CPU Intensive

I can just hear a bunch of you out there yelling “Both!” If that’s the case, then I hope you have a very large budget—but we’ll talk about that scenario, too.

If your system is already installed and running, then you can use a combination of Windows Server’s perfmon (short for Performance Monitor) and SQL Server Profiler to figure out just where your bottlenecks are—in CPU utilization or I/O. CPU utilization can be considered to be high as it starts approaching a consistent 60 percent level. Some would argue and say that number should be as high as 80 percent, but I’m a believer in the idea that people’s time is more expensive than the CPU’s, so I tend to set my thresholds a little lower. I/O depends a lot more on the performance characteristics of your drives and controller.

Chapter 23

Be aware of the concept of “thrash.” As most things under load get closer to their maximum capacity, they start having increasing issues with separate work they are doing colliding—that is, the work of one process starts to interfere in some way with the work of another process. This is part of why I have the 60 percent number for CPUs. I find that doubling your workload when you’re at 10 percent CPU yields you about 20 percent CPU most of the time. Doubling your workload at 20 percent will often yield you 50 percent CPU though as there is a much higher percentage of the time where different processes are pushing each other’s work out of cache or otherwise interfering. In short, I use 60 percent because I find that last 40 percent goes away a lot faster than the first 40 percent did.

You will find this same concept to be very true of I/O and network operations.

If you haven’t installed yet, then it’s a lot more guesswork. While almost anything you do in SQL Server is data based and will, therefore, certainly require a degree of I/O, how much of a burden your CPU is under varies widely depending on the types of queries you’re running.

Low CPU Load	High CPU Load
Simple, single-table queries and updates	Large joins
Joined queries over relatively small tables	Aggregations (SUM, AVG, etc.)
	Sorting of large result sets

With this in mind, let’s focus in a little closer on each situation.

I/O Intensive

I/O-intensive tasks should cause you to focus your budget more on the drive array than on the CPU(s). Notice that I said the drive “array”—I’m not laying that out as an option. In my not-so-humble opinion on this matter, if you don’t have some sort of redundancy arrangement on your database storage mechanism, then you have certainly lost your mind. Any data worth saving at all is worth protecting—we’ll talk about the options there in just a moment.

Before we get into talking about the options on I/O, let’s look briefly into what I mean by I/O intensive. In short, I mean that a lot of data retrieval is going on, but the processes being run on the system are almost exclusively queries (not complex business processes), and those do not include updates that require wild calculations. Remember—your hard drives are, more than likely, the slowest thing in your system (short of a CD-ROM) in terms of moving data around.

A Brief Look at RAID

RAID; it brings images of barbarian tribes raining terror down on the masses. Actually, most of the RAID levels are there for creating something of a fail-safe mechanism against the attack of the barbarian called “lost data.” If you’re not a RAID aficionado, then it might surprise you to learn that not all RAID levels provide protection against lost data.

RAID originally stood for *Redundant Array of Inexpensive Disks*. The notion was fairly simple—at the time, using a lot of little disks was cheaper than one great big one. In addition, an array of disks meant that you had multiple drive heads at work and could also build in (if desired) redundancy.

Playing a Good Tune: Performance Tuning

Since drive prices have come down so much (I'd be guessing, but I'd bet that drive prices are, dollar per meg, far less than 1 percent of what they were when the term RAID was coined), I've heard other renditions of what RAID stands for. The most common are Random Array of Independent Disks (this one seems like a contradiction in terms to me) and Random Array of Individual Disks (this one's not that bad). The thing to remember, no matter what you think it's an acronym for, is that you have two or more drives working together — usually for the goal of some balance between performance and safety.

There are lots of places you can get information on RAID, but let's take a look at the three (well, four if you consider the one that combines two of the others) levels that are most commonly considered:

RAID Level	Description
RAID 0	a.k.a. Disk Striping without Parity. Out of the three that you are examining here, this is the one you are least likely to know. This requires at least three drives to work just as RAID 5 does. Unlike RAID 5, however, you get no safety net from lost data. (Parity is a special checksum value that allows reconstruction of lost data in some circumstances — as indicated by the name, RAID 0 doesn't have parity.)
	RAID 0's big claim to fame is giving you maximum performance without losing any drive space. With RAID zero, the data you store is spread across all the drives in the array (at least 3). While this may seem odd, it has the advantage of meaning that you always have three or more disk drives reading or writing your data for you at once. Under mirroring, the data is all on one drive (with a copy stored on a separate drive). This means you'll just have to wait for that one head to do the work for you.
RAID 1	a.k.a. Mirroring. For each active drive in the system, there is a second drive that "mirrors" (keeps an exact copy of) the information. The two drives are usually identical in size and type, and store all the information to each drive at the same time. (Windows NT has software-based RAID that can mirror any two volumes as long as they are the same size.)
	Mirroring provides no performance increase when writing data (you still have to write to both drives) but can, depending on your controller arrangement, double your read performance since it will use both drives for the read. What's nice about mirroring is that as long as only one of the two mirrored drives fails, the other will go on running with no loss of data or performance (well, reads may be slower if you have a controller that does parallel reads). The biggest knock on mirroring is that you have to buy two drives to every one in order to have the disk space you need.
RAID 5	The most commonly used. Although, technically speaking, mirroring is a RAID (RAID 1), when people refer to using RAID, they usually mean RAID 5. RAID 5 works exactly as RAID 0 does with one very significant exception — parity information is kept for all the data in the array.

Table continued on following page

Chapter 23

RAID Level	Description
	<p>Say, for example, that you have a five-drive array. For any given write, data is stored across all five of the drives, but a percentage of each drive (the sum of which adds up to the space of one drive) is set aside to store parity information. Contrary to popular belief, no one drive is the parity drive. Instead, some of the parity information is written to all the drives—it's just that the parity information for a given byte of data is not stored on the same drive as the actual data is. If any one drive is lost, then the parity information from the other drives can be used to reconstruct the data that was lost.</p> <p>The great thing about RAID 5 is that you get the multidrive read performance. The downside is that you lose one drive's worth of space (if you have a three-drive array, you'll see the space of two; if it's a seven-drive array, you'll see the space of six). It's not as bad as mirroring in the price per megabyte category, but you still see great performance.</p>
RAID 10, a.k.a. RAID 1 + 0 or RAID 0 + 1	<p>Offers the best of both RAID 0 and RAID 1 in terms of performance and data protection. It is, however, far and away the most expensive of the options discussed here.</p> <p>RAID 10 is implemented in a coupling of both RAID 0 (striping without parity) and RAID 1 (mirroring). The end result: Either mirrored sets of striped data or striped sets of mirrored data. The end result in total drive count and general performance is pretty much the same.</p>

The long and the short of it is that RAID 5 is the de facto minimum for database installations. That being said, if you have a loose budget, then I'd actually suggest mixing things up a bit.

RAID 10 has become the standard in larger installations. For the average shop, however, RAID 5 will likely continue to rule the day for a while yet—perhaps that will change as we get into the era where drives are measured in terra-, peta-, and even exabytes. We certainly are getting there fast.

What you'd like to have is at least a RAID 5 setup for your main databases but a completely separate mirrored set for your logs. People who manage to do both usually put both Windows and the logs on the mirror set and the physical databases on the RAID 5 array. Since I'm sure inquiring minds want to know why you would want to do this, let's make a brief digression into how log data is read and written.

Unlike database information, which can be read in parallel (thus why RAID 5 or 10 works so well performance-wise), the transaction log is chronology dependent—that is, it needs to be written and read serially to be certain of integrity. I'm not necessarily saying that physically ordering the data in a constant stream is required; rather, I'm saying that everything needs to be logically done in a stream. As such, it actually works quite well if you can get the logs into their own drive situation where the head of the drive will only seldom have to move from the stream from which it is currently reading and writing. The upshot of this is that you really want your logs to be in a different physical device than your data, so the reading and writing of data won't upset the reading and writing of the log.

Logs, however, don't usually take up nearly as much space as the read data does. With mirroring, we can just buy two drives and have our redundancy. With RAID 5, we would have to buy three, but we don't see any read benefit from the parallel read nature of RAID 5. When you look at these facts together, it doesn't make much sense to go with RAID 5 for the logs or O/S.

You can have all the RAID arrays in the world, and they still wouldn't surpass a good backup in terms of long-term safety of your data. Backups are easy to take off-site, and are not subject to mechanical failure. RAID units, while redundant and very reliable, also become worthless if two (instead of just one) drives fail. Another issue—what if there's a fire? Probably all the drives will burn up—again, without a backup, you're in serious trouble. We'll look into how to back up your databases in Chapter 24.

CPU Intensive

On a SQL Server box, you'll almost always want to make sure that you go multiprocessor, even for a relatively low-utilization machine. This goes a long way to preventing little "pauses" in the system that will drive your users positively nuts, so consider this part of things to be a given—particularly in this day of dual core processors. Keep in mind that the Workgroup version of SQL Server supports only up to two processors—if you need to go higher than that, you'll need to go up to either Standard (four processors) or the Enterprise edition (which is limited only by your hardware and budget).

Even if you're only running SQL Server Express—which supports only one processor—you'll want to stick with the dual-proc box if at all possible. Remember, there is more going on in your system than SQL Server, so having that other proc available to perform external operation cuts down on lag on your SQL Server.

Perhaps the biggest issue of all though is memory. This is definitely one area that you don't want to short change. In addition, remember that if you are in a multiprocessor environment (and you should be), then you are going to have more things going on at once in memory. In these days of cheap memory, no SQL Server worth installing should ever be configured with less than 512MB of RAM—even in a development environment. Production servers should be equipped with no less than 1GB of RAM—quite likely more.

Things to think about when deciding how much RAM to use include:

- ❑ How many user connections will there be at one time (each one takes up space)? Each connection takes up about 24K of memory (it used to be even higher). This isn't really a killer since 1,000 users would only take up 24MB, but it's still something to think about.
- ❑ Will you be doing a lot of aggregations and/or sorts? This can be killers depending on the size of the data set you're working with in your query.
- ❑ How large is your largest database? If you have only one database, and it is only 1GB (and, actually, most databases are much smaller than people think), then having 4GB of RAM probably doesn't make much sense.
- ❑ The Workgroup edition of SQL Server 2005 only supports addressing of memory up to 3GB. If you need more than this, you'll need to go with at least the Standard edition.

In addition, once you're in operation—or when you get a fully populated test system up and running—you may want to take a look at your cache-hit ratio in perfmon. We'll talk about how this number is calculated a little bit later in the chapter. For now, it's sufficient to say that this can serve as something of a measurement for how often we are succeeding at getting things out of memory rather than off disk.

(memory is going to run much, much faster than disk). A low cache-hit ratio is usually a certain indication that more memory is needed. Keep in mind though, that a high ratio does not necessarily mean that you shouldn't add more memory. The read-ahead feature of SQL Server may create what is an artificially high cache-hit ratio and may disguise the need for additional memory.

OLTP vs. OLAP

The needs between these two systems are often at odds with each other. In any case, I'm going to keep my recommendation short here:

If you are running databases to support both of these kinds of needs, run them on different servers — it's just that simple.

On-Site vs. Off-Site

It used to be that anything that would be SQL Server based would be running on-site with those who were responsible for its care and upkeep. If the system went down, people were right there to worry about reloads and to troubleshoot.

In the Internet era, many installations are co-located with an Internet service provider (ISP). The ISP is responsible for making sure that the entire system is backed up — they will even restore according to your directions — but they do not take responsibility for your code. This can be very problematic when you run into a catastrophic bug in your system. While you can always connect remotely to work on it, you're going to run into several configuration and performance issues, including:

- ❑ **Security** — Remote access being open to you means that you're also making it somewhat more open to others who you may not be interested in having access. My two bits worth on this is to make sure that you have very tight routing and port restrictions in place. For those of you not all that network savvy (which includes me), this means that you restrict what IP addresses are allowed to be routed to the remote server.
- ❑ **Performance** — You're probably going to be used to the 100 Mbps to 1 Gbps network speeds that you have around the home office. Now you're communicating via virtual private network (VPN) over the Internet or, worse, dialup and you are starting to hate life (things are SLOW!).
- ❑ **Responsiveness** — It's a bit upsetting when you're running some e-commerce site or whatever and you can't get someone at your ISP to answer the phone, or they say that they will get on it right away and hours later you're still down. Make sure you investigate your remote hosting company very closely — don't assume that they'll still think you're important after the sale.
- ❑ **Hardware maintenance** — Many co-hosting facilities will not do hardware work for you. If you have a failure that requires more than a reloading, you may have to travel to the site yourself or call yet another party to do the maintenance — that means that your application will be offline for hours or possibly days.

If you're a small shop doing this with an Internet site, then off-site can actually be something of a saving grace. It's expensive, but you'll usually get lots of bandwidth plus someone to make sure that the back-ups actually get done — just make sure that you really check out your ISP. Many of them don't know anything about SQL Server, so make sure that expertise is there.

The Risks of Being Down

How long and how often can I afford to be down? This may seem like a silly question. When I ask it, I often get this incredulous look. For some installations, the answer is obvious—they can't afford to be down, period. This number is not, however, as high as it might seem. You see, the only true life-and-death kinds of applications are the ones that are in acute medical applications or are immediately tied to safety operations. Other installations may lose money—they may even cause bankruptcy if they go down—but that's not life and death either.

That being said, it's really not as black and white as all that. There is really something of a continuum in how critical downtime is. It ranges from the aforementioned medical applications at the high end to data-mining operations on old legacy systems at the low end (usually—for some companies, it may be all they have). The thing that pretty much everyone can agree on for every system is that downtime is highly undesirable.

So, the question becomes one of just how undesirable is it? How do we quantify that?

If you have a bunch of bean counters (I can get away with saying that since I was one) working for you, it shouldn't take you all that long to figure out that there are a lot of measurable costs to downtime. For example, if you have a bunch of employees sitting around saying that can't do anything until the system comes back up, then the number of affected employees times their hourly cost (remember, the cost of an employee is more than just his or her wages) equals the cost of the system being down from a productivity standpoint. But wait, there's more. If you're running something that has online sales—how many sales did you lose because you couldn't be properly responsive to your customers? Oops—more cost. If you're running a plant with your system, then how many goods couldn't be produced because the system was down—or, even if you could still build them, did you lose quality assurance or other information that might cost you down the line?

I think by now you should be able to both see and sell to your boss the notion that downtime is very expensive—how expensive depends on your specific situation. Now the thing to do is to determine just how much you're willing to spend to make sure that it doesn't happen.

Lost Data

There's probably no measuring this one. In some cases, you can quantify this by the amount of cost you're going to incur reconstructing the data. Sometimes you simply can't reconstruct it, in which case you'll probably never know for sure just how much it cost you.

Again, how much you want to prevent this should affect your budget for redundant systems as well as things like backup tape drives and off-site archival services.

Is Performance Everything?

More often than not, the answer is no. It's important, but just how important has something of diminishing returns to it. For example, if buying those extra 100 Mhz of CPU power is going to save you two seconds per transaction—that may be a big deal if you have 50 data entry clerks trying to enter as much as they can a day. Over the course of a day, seemingly small amounts of time saved can add up. If each of those 50 clerks are performing 500 transactions a day, then saving two seconds per transaction adds up to over 13 man hours (that's over one person working all day!). Saving that time may allow you to delay a little longer in adding staff. The savings in wages will probably easily pay for the extra power.

Chapter 23

The company next door may look at the situation a little differently though—they may only have one or two employees; furthermore, the process that they are working in might be one where they spend a lengthy period of time just filing out the form—the actual transaction that stores it isn’t that big of deal. In such a case, their extra dollars for the additional speed may not be worth it.

Driver Support

Let’s start off by cutting to the chase—I don’t at all recommend that you save a few dollars (or even a lot of dollars) when buying your server by purchasing it from some company like “Bob’s Pretty Fine Computers.” Remember, all those risks? Now, try introducing a strange mix of hardware and driver sets. Now imagine when you have a problem—you’re quickly going to find all those companies pointing the finger at each other saying, “It’s their fault!” Do you really want to be stuck in the middle?

What you want is the tried and true—the tested—the known. Servers—particularly data servers—are an area to stick with well-known, trusted names. I’m not advocating anyone in particular (no ads in this book!), but I’m talking very mainstream people like Dell, IBM, HP, etc. Note that, when I say well-known, trusted names, I mean names that are known in servers. Just because someone sells a billion desktops a year doesn’t mean they know anything about servers—it’s almost like apples and oranges. They are terribly different.

By staying with well-known equipment, in addition to making sure that you have proper support when something fails, it also means that you’re more likely to have that equipment survive upgrades well into the future. Each new version of the O/S only explicitly supports just so many pieces of equipment—you want to be sure that yours is one of them.

The Ideal System

Let me preface this by saying that there is no one ideal system. That being said, there is a general configuration (size excluded) that I and a very large number of other so called “experts” seem to almost universally push as where you’d like to be if you had the budget for it. What we’re talking about is drive arrangements here (the CPU and memory tends to be relative chicken feed budget- and setup-wise).

What you’d like to have is a mix of mirroring and RAID 5 or 10. You place the O/S and the logs on the mirrored drives. You place the data on the RAID 5/10 array. That way, the O/S and logs—which both tend to do a lot of serial operations—have a drive setup all of their own without being interfered with by the reads and writes of the actual data. The data has a multithread read/write arrangement for maximum performance, while maintaining a level of redundancy.

Troubleshooting

SQL Server offers a number of options to help with the prevention, detection, and measurement of long-running queries. The options range from a passive approach of measuring actual performance, so you know what’s doing what, to a more active approach of employing a query “governor” to automatically kill queries that run over a length of time you choose. These tools are very often ignored or used only sparingly—which is something of a tragedy—they can save hours of troubleshooting by often leading you right to the problem query and even the specific portion of your query that is creating the performance issues.

Tools to take a look at include:

- SHOWPLAN_TEXT|ALL and Graphical showplan—Looked at in this chapter
- STATISTICS IO—Also in this chapter
- Database Consistency Checker (DBCC)—Also in this chapter
- The Query Governor—Also in this chapter
- The sys.processes table function
- The SQL Server Profiler—Covered in this chapter

Many people are caught up in just using one of these, but the reality is that there is little to no (depending on which two you’re comparing) overlap between them. This means that developers and DBAs who try to rely on just one of them are actually missing out on a lot of potentially important information.

Also, keep in mind that many of these are still useful in some form even if you are writing in a client-side language and sending the queries to the server (no sprocs). You can either watch the query come through to your server using the SQL Server Profiler, or you could even test the query in QA before moving it back to your client code.

The Various Showplans and STATISTICS

SQL Server gives you a few different SHOWPLAN options. The information that they provide varies a bit depending on what option you choose, but this is one area where there is a fair amount of overlap between your options; however, each one definitely has its own unique thing that it brings to the picture. In addition, there are a number of options available to show query statistics.

Here are the options and what they do.

SHOWPLAN TEXT|ALL

When either of these two SHOWPLAN options (they are mutually exclusive) is executed, SQL Server changes what results you get for your query. Indeed, the NOEXEC option (which says to figure out the query plan to don’t actually do it) is put in place, and you receive no results other than those put out by the SHOWPLAN.

The syntax for turning the SHOWPLAN on and off is pretty straightforward:

```
SET SHOWPLAN TEXT|ALL ON|OFF
```

When you use the TEXT option, you get back the query plan along with the estimated costs of running that plan. Since the NOEXEC option automatically goes with SHOWPLAN, you won’t see any query results.

When you use the ALL option, you receive everything you received with the TEXT option, plus a slew of additional statistical information, including such things as:

- The actual physical and logical operations planned
- Estimated row counts

Chapter 23

- Estimated CPU usage
- Estimated I/O
- Average row size
- Whether the query will be run in parallel or not

Let's run a very brief query utilizing (one at a time) both of these options:

```
SET SHOWPLAN_TEXT ON
GO

SELECT *
FROM Sales.SalesOrderHeader
GO

SET SHOWPLAN_TEXT OFF
GO

SET SHOWPLAN_ALL ON
GO

SELECT *
FROM Sales.SalesOrderHeader
GO

SET SHOWPLAN_ALL OFF
GO
```

Notice that every statement is followed by a GO — thus making it part of its own batch. The batches that contain the actual query could have had an unlimited number of statements, but the batches setting the SHOWPLAN option have to be in a batch by themselves.

The SHOWPLAN_TEXT portion of the results should look something like this:

```
StmtText
-----
SELECT *
FROM Sales.SalesOrderHeader

(1 row(s) affected)

StmtText
-----
----- | --Compute Scalar(DEFINE:([AdventureWorks].....
| --Compute Scalar(DEFINE:([AdventureWorks]...
| --Clustered Index Scan(OBJECT:([AdventureWorks]...

(3 row(s) affected)
```

Unfortunately, the results are far too wide to fit all of it gracefully in the pages of this book, but there are a couple key things I want you to notice about what was produced.

- There are multiple steps displayed.
- At each step, what object is being addressed and what kind of operation is being supplied.

If we had been running a larger query—say something with several joins—then even more sub-processes would have been listed with indentations to indicate hierarchy.

I'm going to skip including the `ALL` results here since they simply will not fit in a book format (it's about 800 characters wide and won't fit in any readable form in a book—even if we flipped things sideways), but it included a host of other information. Which one of these to use is essentially dependent on just how much information you want to be flooded with. If you just want to know the basic plan—such as is it using a merge or hash join, you probably just want to use the `TEXT` option. If you really want to know where the costs are and such, then you want the `ALL` option.

Since the `SHOWPLAN` options imply the `NOEXEC`—that means nothing in your query is actually being executed. Before you do anything else, you need to set the option back to off—that even includes switching from one showplan option to the other (for example, `SET SHOWPLAN_ALL ON` wouldn't have any effect if you had already run `SET SHOWPLAN_TEXT ON` and hadn't yet turned it off).

I like to make sure that every script I run that has a `SET SHOWPLAN` statement in it has both the on and off within that same script. It goes along way toward keeping me from forgetting that I have it turned on and being confused when things aren't working the way I expect.

Graphical Showplan

The graphical showplan tool combines bits and pieces of the `SHOWPLAN_ALL` and wraps them up into a single graphical format. Graphical showplan is a Management Studio-only tool. It is selected through options in Management Studio rather than through T-SQL syntax—this means that it is only available when using Management Studio.

The graphical showplan comes in two versions: estimated and actual. The estimated version is more like the `SHOWPLAN` in T-SQL—it implies that the query plan is just developed but not actually executed. The actual essentially waits until the query is done and shows you the way the query was actually done in the end.

Why are these different? Well, SQL Server is smart enough to recognize when it starts down a given query plan based on an estimated cost and then finds the reality to be something other than what its estimates were based on. SQL Server uses statistics it keeps on tables and indexes to estimate cost. Those statistics can sometimes become skewed or downright out of date—the query optimizer will adjust on the fly if it starts down one path and finds something other than what it expected.

For most things we do, the estimated execution plan is just fine. We have two options to activate the graphical showplan option:

- Select the Display Estimated Execution Plan option from the Query menu
- Click on the Display Estimated Execution Plan button on the toolbar and in the Query menu (this option just shows us the plan with the `NOEXEC` option active)

Chapter 23

Personally, I like the option of having the graphical showplan in addition to my normal query run. While it means that I have to put the actual hit of the query on my system, it also means that the numbers I get are no longer just estimates but are based on the actual cost numbers. Indeed, if you run the showplan both ways and wind up with wildly different results, then you may want to take a look at the last time your statistics were updated on the tables on which the query is based. If necessary, you can then update them manually and try the process again.

The hierarchy of the different sub-processes is then shown graphically. In order to see the costs and other specifics about any sub-process, just hover your mouse over that part of the graphical showplan and a tooltip will come up with the information:

This arrangement, as shown in Figure 23-1, can often make it much easier to sort out the different pieces of the plan. The downside is that you can't print it out for reporting the way that you can with the text versions.

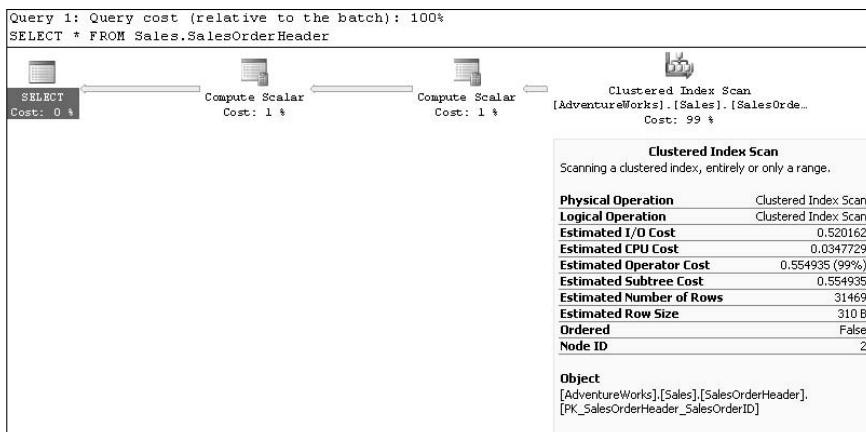


Figure 23-1

STATISTICS

In addition to using the graphical showplan with actual execution of the query, you have a couple of other options for retrieving the “real” information on the statistics of your query—using SQL Server Profiler (discussed later in this chapter) and turning on STATISTICS PROFILE.

STATISTICS actually has a couple of options that can be very handy in troubleshooting query performance, including those discussed the following sections.

SET STATISTICS IO ON|OFF

This one is a very commonly used tool to figure out where and how the query is performing. STATISTICS IO provides several key pieces of information regarding the actual work necessary to perform your query. Information provided includes:

- ❑ **Physical Reads**—This represents the actual physical pages read from disk. It is never any more than, and is usually smaller than, the number for logical reads. This one can be very misleading in the sense that it will usually change (be less than the first run) the second time that you run

your query. Any page that is already in the buffer cache will not have a physical read done on it, so, the second time you run the query in a reasonably short succession, the pages involved will, more than likely, still be in cache. In addition, this number will not be incremented if the page has already been read due to the read-ahead mechanism that is part of SQL Server. This means that your query may be responsible for loading the page physically into cache, but it still may not show up as part of the physical reads.

- ❑ **Logical Reads**—This is the number of times that the page was actually looked at—regardless of from where it came. That is, any page already in the memory cache will still create a logical read if the query makes use of it. Note that I said it is how many times the page was looked at—that means that you may have several logical reads for a single page if the page is needed several times (say for a nested loop that affects a page that has several rows on it).
- ❑ **Read Ahead Reads**—This is the number of pages that SQL Server read into the cache as a result of the read-ahead mechanism anticipating that they will be needed. The page may actually be used—or it may not. In either case, the read still counts as a read ahead. Read aheads are very similar to physical reads in the sense that they represent data being physically read from disk. The problem is that the number you get is based on the optimistic nature of the read-ahead mechanism and does not necessarily mean that all that work was actually put to use.
- ❑ **Scan Count**—The scan count represents the number of times that a table was accessed. This is somewhat different from Logical Reads, which was focused on page access. This is another situation where a nested loop is a good example. The outer table that is forming the basis for the condition on the query that is on the inside may only have a scan count of 1, where the inner loop table would have a scan count added for every time through the loop—that is, every record in the outer table.

Some of the same information that forms the basis for `STATISTICS IO` is the information that feeds your cache-hit ratio if you look in perfmon. The cache-hit ratio is based on the number of logical reads, less the physical reads, divided into the total actual reads (logical reads).

The thing to look for with `STATISTICS IO` is for any one table that seems disproportionately high in either physical or logical reads.

A very high physical read count could indicate that the data from the table is being pushed out of the buffer cache by other processes. If this is a table that you are going to be accessing with some regularity, then you may want to look at purchasing (or, if you're an ISV developing a SQL Server product, recommending) more memory for your system.

If the logical reads are very high, then the issue may be more one of proper indexing. I'll give an example here from a client I had some time back. A query was taking approximately 15 seconds to run on an otherwise unloaded system. Since the system was to be a true OLTP system, this was an unacceptable time for the user to have to wait for information (the query was actually a fairly simple lookup that happened to require a four-table join). In order to find the problem, I used what amounted to `STATISTICS IO`—it happened to be the old graphical version that came with 6.5, but the data was much the same. After running the query just once, I could see that the process was requiring less than 20 logical reads from three of the tables, but it was performing over 45,000 logical reads from the fourth table. This is what I liked about the old graphical version; it took about a half a second to see that the bar on one table stretched all the way across the screen when the others were just a few pixels! From there, I knew right where to focus—in about two minutes, I had an index built to support a foreign key (remember, they aren't built by default), and the response time dropped to less than a second. The entire troubleshooting process on this one took literally minutes. Not every performance troubleshooting effort is that easy, but using the right tools can often help a lot.

SET STATISTICS TIME ON|OFF

This one is amazingly little known. It shows the actual CPU time required to execute the query. Personally, I often use a simple `SELECT GETDATE()` before and after the query I'm testing—as we've done throughout most of the book—but this one can be handy because it separates out the time to parse and plan the query versus the time to actually execute the query. It's also nice to not have to figure things out for yourself (it will calculate the time in milliseconds—using `GETDATE()` you have to do that yourself).

Include Client Statistics

You also have the ability to show statistical information about your connection as part of your query run. To make use of this, just select Include Client Statistics from the Query menu. As long as that option is set, every execution you make will produce a Client Statistics tab in the results pane of the Query Window, as shown in Figure 23-2.

	Trial 1	Average
Client Execution Time	15:15:17	
Query Profile Statistics		
Number of INSERT, DELETE and UPDATE statements	0	→ 0.0000
Rows affected by INSERT, DELETE, or UPDATE statem...	0	→ 0.0000
Number of SELECT statements	1	→ 1.0000
Rows returned by SELECT statements	31469	→ 31469.0000
Number of transactions	0	→ 0.0000
Network Statistics		
Number of server roundtrips	1	→ 1.0000
TDS packets sent from client	1	→ 1.0000
TDS packets received from server	1476	→ 1476.0000
Bytes sent from client	106	→ 106.0000
Bytes received from server	6044308	→ 6044308.0000
Time Statistics		
Client processing time	1375	→ 1375.0000
Total execution time	1640	→ 1640.0000
Wait time on server replies	265	→ 265.0000

Figure 23-2

The Database Consistency Checker (DBCC)

The Database Consistency Checker (or DBCC) has a number of different options available to allow you to check the integrity and structural makeup of your database. This is far more the realm of the DBA than the developer, so I am, for the most part, considering the DBCC to be out of scope for this book. The most notable exceptions come in the maintenance of indexes—those were discussed back in Chapter 8.

The Query Governor

The query governor is a tool that's most easily found in Management Studio (you can also set it using `sp_configure`). This sets the maximum cost a query can be estimated to have and still have SQL Server execute the query—that is, if the estimated cost of running the query exceeds that allowed in the query governor, then the query will not be allowed to run.

To get to the query governor in Management Studio, right-click on your server and choose Properties. Then choose the Connections tab. The number you enter into the query governor field loosely (*very loosely*) equates to number of seconds. So, theoretically, if you set it to 180, that's about three minutes. Queries that are estimated to run longer than three minutes would not be allowed to run. This can be very handy for keeping just a few rather long queries from taking over your system.

Did you happen to notice that I slipped the word *theoretically* into the preceding paragraphs? There's a very important reason for that — the time is just an estimate. What really drives the limit is what the optimizer calculates as the estimated "cost" for that query. The problem is that the time is an estimate based on how cost equates to time. The values used are based on a single test box at Microsoft. You can expect that, as systems get faster, the equation between the query governor setting and the real-time factor will get less and less reliable.

The SQL Server Profiler

The true lifesaver among the tools provided with SQL Server, this one is about letting you "sniff out" what's really going on with the server.

Profiler can be started from the Start menu in Windows. You can also run it by selecting the Tools menu in Management Studio. When you first start it up, you can either load an existing profile template or create a new one.

Let's take a look at some of the key points of the main Profiler by walking through a brief example.

Start by choosing New→Trace from the File menu. Log in to the server you've been working with, and you should be presented with the dialog box in Figure 23-3.

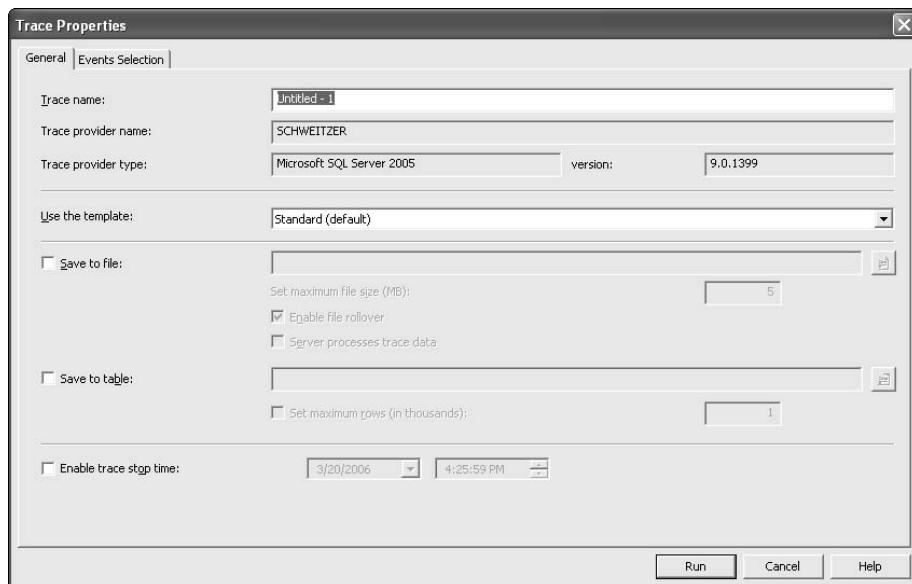


Figure 23-3

Chapter 23

The trace name is probably obvious enough, but the template information might not be. A template is a set of pre-established events, data columns, and filters that you want to see in a trace, and the templates provided with SQL Server are named for the kind of situation that you might want to use them in. Any templates that are stored in the default profiler template directory (which is under the tools subdirectory of wherever you installed SQL Server) are included in the Use the template dropdown box.

Pay particular attention to what template you choose — it decides exactly how much is available to you on the next tab. If you choose too restrictive of a template, you can select “Show all events” and “Show all columns” to expose all possible choices.

Next up, you can choose whether to capture the trace to a file on disk or a table in the database. If you save to a file, then that file will be available only to the system that you store it on (or anyone who has access to a network share if that's where you save it). If you save it to a table, then everyone who can connect to the server and has appropriate permissions will be able to examine the trace.

Last, but not least, on this dialog is the stop time feature — this allows you to leave a trace running (for example, for a workload file or some other long-running trace need) and have it shut down automatically at a later time.

Things get somewhat more interesting on the tab that comes next, as shown in Figure 23-4:

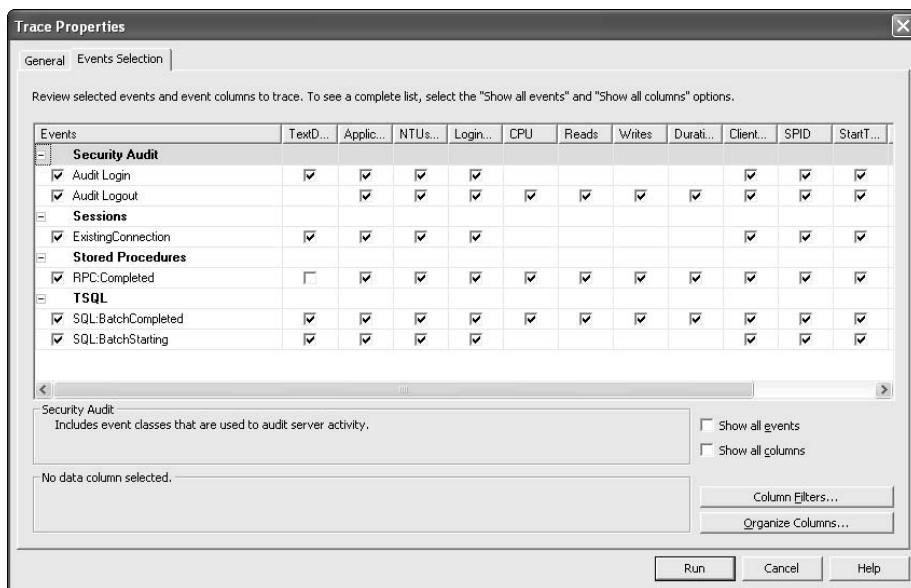


Figure 23-4

I've chosen the "blank" template here. This one is all about what events you are going to track, and, as you can see, there's quite a range. If, for example, you chose the Tuning trace template, then the initial setup is one that tracks what's needed for the Index Tuning Wizard plus a bit more. In addition, you use the table to select what information you want collected for each class of event.

The temptation here is just to select everything under the sun, so you'll be sure to have all the information. There are a couple of reasons not to do this. First, it means that a lot of additional text has to come back down the pipe to your server. Remember that SQL Server Profiler has to place some audits in the system, and this means that your system is having an additional burden placed on it whenever the Profiler is running—the bigger the trace, the bigger the burden. Second, it often means lower productivity for you since you have to wade through a huge morass of data—much of which you probably won't need.

I want to point out a couple of key fields here before we move on:

- TextData**—This is the actual text of the statement that the Profiler happens to have added to the trace at that moment in time.
- Application Name**—Another of those highly underutilized features. The application name is something you can set when you create the connection from the client. If you're using ADO.NET or some other data object model and underlying connection method, you can pass the application name as a parameter in your connection string. It can be quite handy for your DBAs when they are trying to troubleshoot problems in the system.
- NT User Name**—This one is what it sounds like—what's great about this is that it can provide a level of accountability.
- Login Name**—Same as NT User Name, only used when operating under SQL Server Security rather than Windows Security.
- CPU**—The actual CPU cycles used.
- Duration**—How long the query ran—including time waiting for locks and such (where the CPU may not have been doing anything, so doesn't reference that load).
- SPID (SQL Process ID)**—This one can be nice if your trace reveals something where you want to kill a process—this is the number you would use with your KILL statement.

Moving right along, let's take a look at what I consider to be one of the most important options—Column Filters:

This is the one that makes sure that, on a production or load test server, you don't get buried in several thousand pages of garbage just by opening a trace up for a few minutes.

With Column Filters, you can select from a number of different options to use to filter out data and limit the size of your result set. By default, Profiler automatically sets up to exclude its own activity in order to try to reduce the Profiler's impact on the end numbers. For the example in Figure 23-5, I'm adding in a Duration value where I've set the minimum to 3,000 milliseconds with no maximum.

Odds are that, if you run this with a query against the `Sales.SalesOrderHeaders` table, you're not going to see it appear in the trace. Why is that? Because that query will probably run very fast and not meet the criteria for being included in our trace—this is an example of how you might set up a trace to capture the query text and username of someone who has been running very long running queries on the system. Now try running something a little longer—such as a query that joins many large tables. There's a good chance that you'll now exceed the duration threshold, and your query will show up in the Profiler (if not, then try adjusting down the duration expectation that you set in Profiler).

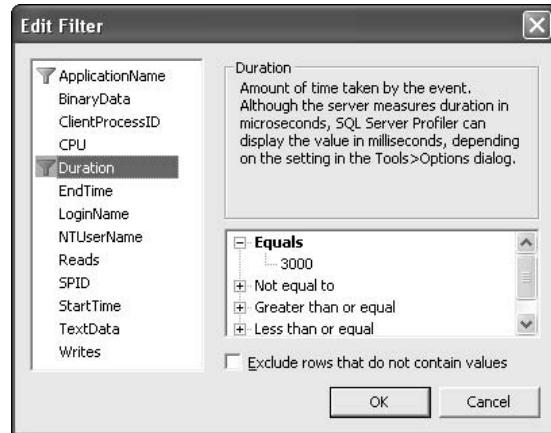


Figure 23-5

I can't say enough about how important this tool is in solving performance and other problems. Too many times to count have I been thinking that my sproc was running down one logic path only to find that a totally different branch was being executed. How did I originally find out? I watched it execute in Profiler.

The Performance Monitor (*Perfmon*)

When you install SQL Server on Windows, SQL Server adds several counters to the *Performance Monitor* (which is sometimes called *perfmon* because of the executable's filename — *perfmon.exe*). This can be an excellent tool for finding where problems are happening and even determining the nature of some problems.

The Performance Monitor can be accessed through the Administrative Tools menu in Windows labeled as Performance. SQL Server has a number of different Performance Objects, and, within each of these, you will find a series of counters related to that object. Historically, some of the important ones have included:

- ❑ **SQLServer Cache Manager: Buffer Hit Cache Ratio** — This is the number of pages that were able to be read from the buffer cache rather than having to issue a physical read from disk. The thing to watch out for here is that this number can be thrown off depending on how effective the read-ahead mechanism was — anything that the read-ahead mechanism got to and put in cache before the query actually needed it is counted as a buffer-cache hit — even though there really was a physical read related to the query. Still, this one is going to give you a decent idea of how efficient your memory usage is. You want to see really high numbers here (in the 90+ percent range) for maximum performance. Generally speaking, a low buffer hit cache ratio is indicative of needing more memory.
- ❑ **SQLServer General Statistics: User Connections** — Pretty much as it sounds, this is the number of user connections currently active in the system.

- ❑ **SQLServer Memory Manager: Total Server Memory**—The total amount of dynamic memory that the SQL Server is currently using. As you might expect, when this number is high relative to the amount of memory available in your system (remember to leave some for the O/S!), you need to seriously consider adding more RAM.
- ❑ **SQLServer SQL Statistics: SQL Compilations/sec**—This is telling you how often SQL Server needs to compile things (sprocs, triggers). Keep in mind that this number will also include recompiles (due to changes in index statistics or because a recompile was explicitly requested). When your server is first getting started, this number may spike for a bit, but it should become stable after your server has been running for a while at a constant set and rate of activities.
- ❑ **SQLServer Buffer Manager: Page Reads/sec**—The number of physical reads from disk for your server. You'd like to see a relatively low number here. Unfortunately, because the requirements and activities of each system are different, I can't give you a benchmark to work from here.
- ❑ **SQLServer Buffer Manager: Page Writes/sec**—The number of physical writes performed to disk for your server. Again, you'd like a low number here.

If you want to add or change any of these, just click on the plus (+) sign up on the tool bar. You'll be presented with a dialog, as shown in Figure 23-6, that lets you choose between all the different objects and counters available on your system (not just those related to SQL Server):

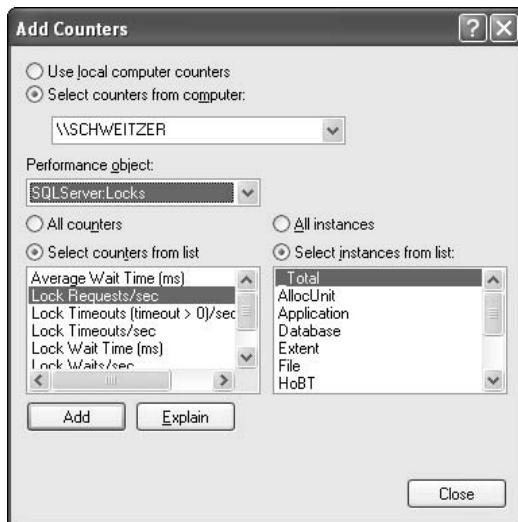


Figure 23-6

The big thing here is to realize that you can mix and match a wide variety of counters to be able to reach a better understanding of what's going on with your server and make the appropriate adjustments. Much of the time, this kind of a task is going to have more to do with the DBA than the developer, but many of these stats can be helpful to you when you are doing load testing for your application.

Summary

Performance could be, and should be, in a book by itself. There's simply just too much to cover and get acquainted with to do it all in one or even several chapters. The way I've tried to address this is by pointing out performance issues throughout the book, so you could take them on a piece at a time.

The biggest thing is to have a plan — a performance plan. Make performance an issue from the first stages of your project. Set benchmarks early, and continually measure your system against those benchmarks to know where you are improving and what problems you might need to address.

In this chapter, we've reviewed a number of the performance considerations touched on throughout the book, plus added several new tools and ideas to consider.

In the next chapter, we'll be taking a look at administration issues. As you've seen through some of the portions of this chapter, proper administration can also be a key ingredient to performance.

24

Administrator

So, at this point we've covered all of the core database topics and then some. We still have a chapter or two to clean up the edges around our development effort, but we've mostly covered everything—heh, NOT!!! For the developer, we like to think our job is done, but for the application we're building, it's just beginning. And so, it's time to talk a bit about maintenance and administration of the databases you develop.

As a developer, I can just hear it now: "Isn't that the database administrator's job?" If you did indeed say something like that, then step back, and smack yourself—*hard* (and no, I'm not kidding). If there is anything I hope to instill in you in your database development efforts, it's to avoid the "hey, I just build 'em—now it's your problem" attitude that is all too common out there.

A database-driven application is a wildly different animal than most standalone apps. Most standalone applications are either self-maintaining or deal with single files that are relatively easy for a user to copy somewhere for backup purposes. Likewise, they usually have no "maintenance" issues the way that a database does.

In this chapter, we're going to take a look at some of the tasks that are necessary to make sure that you end users can not only recover from problems and disasters but also perform some basic maintenance that will help things keep running smoothly.

Among the things we'll touch on are:

- Scheduling jobs
- Backing up and recovering
- Basic defragmenting and index rebuilding
- Setting alerts
- Archiving

While these are far from the only administration tasks available, these do represent something of "the minimum" you should expect to address in the deployment plans for your app.

Scheduling Jobs

Many of the tasks that we'll go over in the remainder of the chapter can be *scheduled*. Scheduling jobs allows you to run tasks that place a load on the system at off-peak hours. It also ensures that you don't forget to take care of things. From index rebuilds to backups, you'll hear of horror stories over and over about shops that "forgot" to do that, or thought they had set up a scheduled job but never checked on it.

If your background is in Windows Server, and you have scheduled other jobs using the Windows Scheduler service, you could utilize that scheduling engine to support SQL Server. Doing things all in the Windows Scheduler allows you to have everything in one place, but SQL Server has some more robust branching options.

There are basically two terms to think about: jobs and tasks.

- Tasks**—These are single processes that are to be executed, or batch of commands that are to be run. Tasks are not independent—they exist only as members of jobs.
- Jobs**—These are a grouping of one or more tasks that should be run together. You can, however, set up dependencies and branching depending on the success or failure of individual tasks (for example, task A runs if the previous task succeeds, but task B runs if the previous task fails).

Jobs can be scheduled based on:

- A daily, weekly, or monthly basis
- A specific time of the day
- A specific frequency (say, every 10 minutes, or every hour)
- When the CPU becomes idle for a period of time
- When the SQL Server Agent starts
- In response to an alert

Tasks are run by virtue of being part of a job and based on the branching rules you define for your job. Just because a job runs doesn't mean that all the tasks that are part of that job will run—some may be executed and others not depending on the success or failure of previous tasks in the job and what *branching rules* you have established. SQL Server not only allows one task to automatically fire when another finishes—but it also allows for doing something entirely different (such as running some sort of recovery task) if the current task fails.

In addition to branching you can, depending on what happens, also tell SQL Server to:

- Provide notification of the success or failure of a job to an operator. You're allowed to send a separate notification for a network message (which would pop-up on a user's screen as long as they are logged in), a pager, and an e-mail address to one operator each.
- Write the information to the event log.
- Automatically delete the job (to prevent executing it later and generally "clean up").

Let's take a quick look at how to create operators in Management Studio.

Creating an Operator

If you're going to make use of the notification features of the SQL Agent, then you must have an operator set up to define the specifics for who is notified. This side of things—the creation of operators—isn't typically done through any kind of automated process or as part of the developed code. These are usually created manually by the DBA. We'll go ahead and take a rather brief look at it here just to understand how it works in relation to the scheduling of tasks.

Creating an Operator Using Management Studio

To create an operator using Management Studio, you need to navigate to the SQL Server Agent node of the server for which you're creating the operator. Expand the SQL Server Agent node, right-click on the Operators member, and choose New Operator.

Be aware that, depending on your particular installation, the SQL Server Agent Service may not start automatically by default. If you run into any issues or if you notice the SQL Server Agent icon in the Management Studio has a little red square in it, then the service is probably set to manual or even disabled—you will probably want to change the service to start automatically. Regardless, make sure that it is running for the examples found in this chapter. You can do this by right-clicking the Agent node and selecting Manage Service.

You should be presented with the dialog box shown in Figure 24-1 (mine is partially filled in).

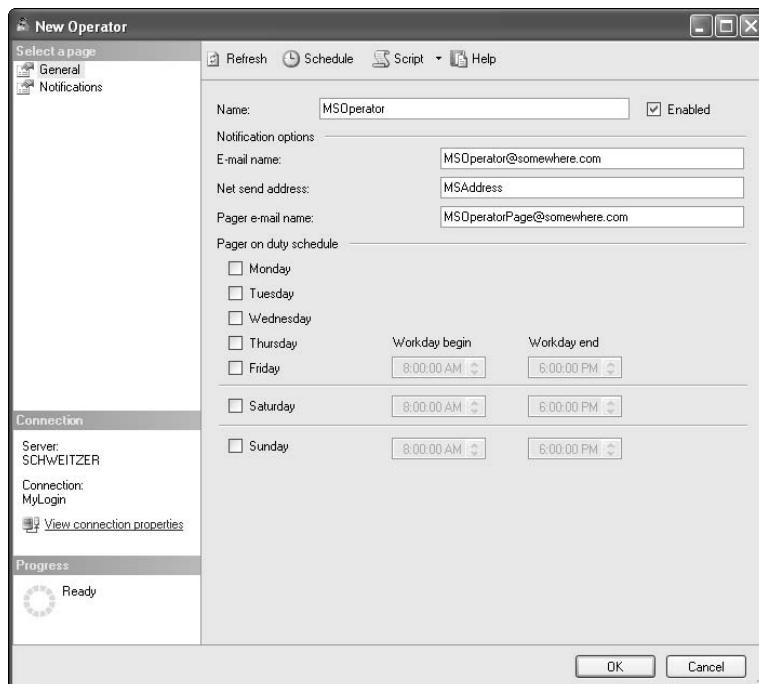


Figure 24-1

You can then fill out a schedule for what times this operator is to receive e-mail notifications for certain kinds of errors that we'll see on the Notifications tab.

Chapter 24

Speaking of that Notifications tab, go ahead and click over to that tab. It should appear as in Figure 24-2.

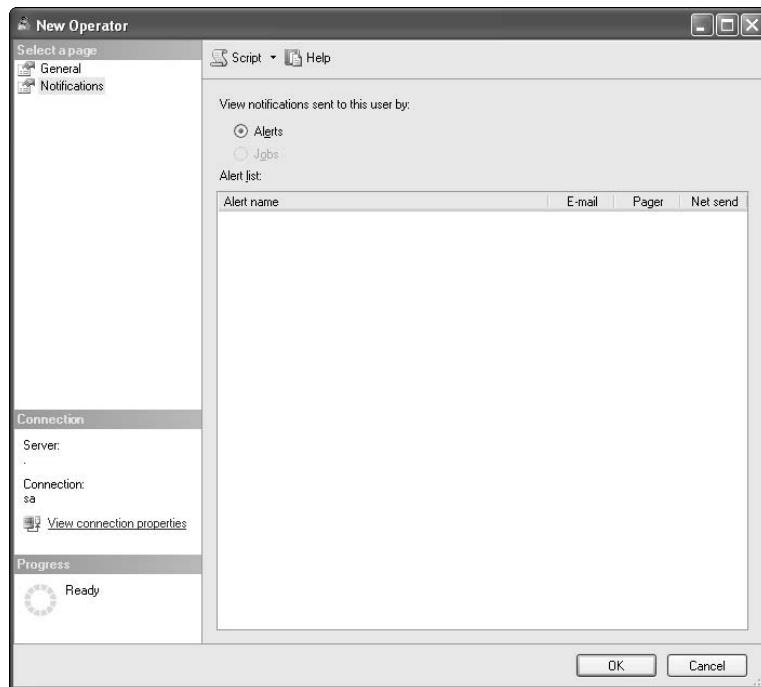


Figure 24-2

Until you have more alerts in your system (we'll get to those later in this chapter), this page may not make a lot of sense. What it is about is setting up what notifications you want this operator to receive depending on what defined alters get triggered. Again, hard to understand this concept before we've gotten to alerts, but suffice to say that alerts are triggered when certain things happen in your database, and this page defines which alerts this particular operator receives.

Creating an Operator Using T-SQL

If you do decide to create operators programmatically, you can make use of the `sp_add_operator` system sproc. Note that, since this is a SQL Server Agent sproc, you'll find it only in the `msdb` database.

After seeing all the different things you need to choose in Management Studio, it probably won't surprise you to find out that this sproc has a ton of different parameters. Fortunately, a number of them are optional, so you need to supply them only if you're going to make use of them. The syntax looks like this:

```
sp_add_operator [@name =] '<operator name>'  
[ ,[@enabled =] <0 for no, 1 for yes>]  
[ ,[@email_address =] '<email alias or address>']  
[ ,[@pager_address =] '<pager address>']  
[ ,[@weekday_pager_start_time =] <weekday pager start time>]  
[ ,[@weekday_pager_end_time =] <weekday pager end time>]  
[ ,[@saturday_pager_start_time =] <Saturday pager start time>]  
[ ,[@saturday_pager_end_time =] <Saturday pager end time>]
```

```
[, [@sunday_pager_start_time =] <Sunday pager start time>]
[, [@sunday_pager_end_time =] <Sunday pager end time>]
[, [@pager_days =] <pager days>]
[, [@netsend_address =] '<netsend address>']
[, [@category_name =] '<category name>']
```

Most of the parameters in this sproc are self-describing, but there are a few we need to take a closer look at:

- ❑ @enabled—This is a Boolean value and works just the way you would typically use a bit flag—0 means disable this operator and 1 means enable the operator.
- ❑ @email_address—This one is just a little tricky. In order to use e-mail with your SQL Server, you need to configure Database Mail to be operational using a specific mail server. This parameter assumes that whatever value you supply is an alias on that mail server. If you are providing the more classic e-mail address type (`somebody@SomeDomain.com`), then you need to enclose it in brackets—like `[somebody@SomeDomain.com]`. Note that the entire address—including the brackets—must still be enclosed in quotation marks.
- ❑ @pager_days—This is a number that indicates the days that the operator is available for pages. This is probably the toughest of all the parameters. This uses a single-byte bit-flag approach similar to what we saw with the `@@OPTIONS` global variable that we looked at earlier in the book (and is also described in the system functions appendix at the back of the book). You simply add the values together for all the values that you want to set as active days for this operator. The options are:

Value	Day of Week
Sunday	1
Monday	2
Tuesday	4
Wednesday	8
Thursday	16
Friday	32
Saturday	64

Okay, so let's go ahead and create our operator using `sp_add_operator`. We'll keep our use of parameters down, since many of them are redundant:

```
USE msdb
DECLARE @PageDays int

SELECT @PageDays = 2 + 8 + 32 -- Monday, Wednesday, and Friday

EXEC sp_add_operator @name = 'TSQLOperator',
    @enabled = 1,
    @pager_address = 'YourEmail@YourDomain.com',
    @weekday_pager_start_time = 080000,
    @weekday_pager_end_time = 170000,
    @pager_days = @PageDays
```

Chapter 24

If you go back into Management Studio and refresh your Operators list, you should see your new operator there.

There are three other sprocs (plus one to retrieve information) that you need to make use of in order to have power over your operator from T-SQL:

- ❑ `sp_help_operator`—Provides information on the current settings for the operator.
- ❑ `sp_update_operator`—Accepts all the same information as `sp_add_operator`; the new information completely replaces the old information.
- ❑ `sp_delete_operator`—Removes the specified operator from the system.
- ❑ `sp_add_notification`—Accepts an alert name, an operator name, and a method of notification (e-mail, pager, netsend). Adds a notification such that, if the alert is triggered, then the specified operator will be notified via the specified method.

Now that you've seen how to create operators, let's take a look at creating actual jobs and tasks.

Creating Jobs and Tasks

As I mentioned earlier, jobs are a collection of one or more tasks. A task is a logical unit of work, such as backing up one database or running a T-SQL script to meet a specific need such as rebuilding all your indexes.

Even though a job can contain several tasks, this is no guarantee that every task in a job will run. They will either run or not run depending on the success or failure of other tasks in the job and what you've defined as the response for each case of success or failure. For example, you might cancel the remainder of the job if one of the tasks fails.

Just like operators, jobs can be created in Management Studio as well as programmatic constructs.

Creating Jobs and Tasks Using the Management Studio

The SQL Server Management Studio makes it very easy to create scheduled jobs. Just navigate to the SQL Server Agent node of your server. Then right-click on the Jobs member and select New Job. You should get a multinode dialog box, shown in Figure 24-3, that will help you build the job one step at a time.

The name can be whatever you like as long as it adheres to the SQL Server rules for naming, as discussed early in this book.

Most of the rest of the information is, again, self-explanatory with the exception of Category—which is just one way of grouping together jobs. Many of your jobs that are specific to your application are going to be Uncategorized, although you will probably on occasion run into instances where you want to create Web Assistant, Database Maintenance, Full Text or Replication Jobs—those each go into their own category for easy identification.

We can then move on to Steps, as shown in Figure 24-4. This is the place where we tell SQL Server to start creating our new tasks that will be part of this job.

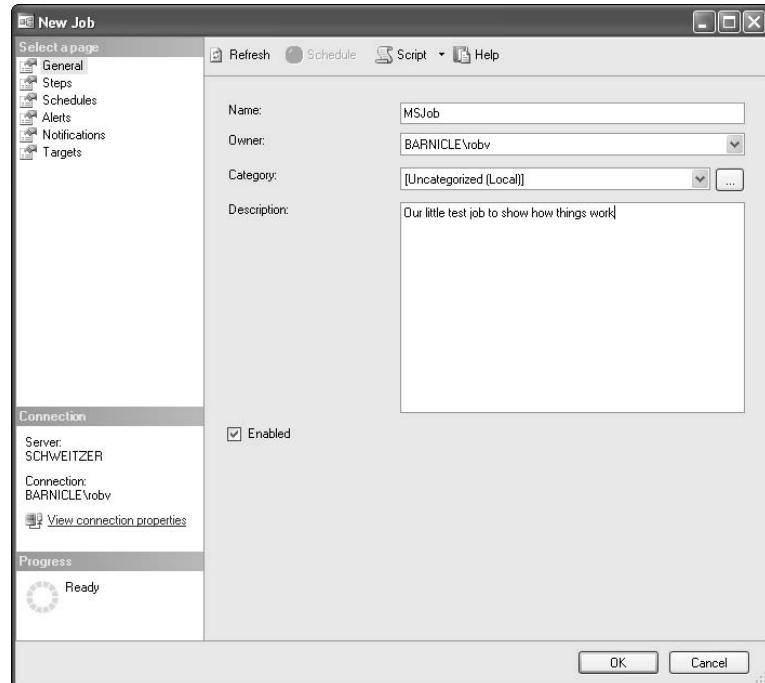


Figure 24-3

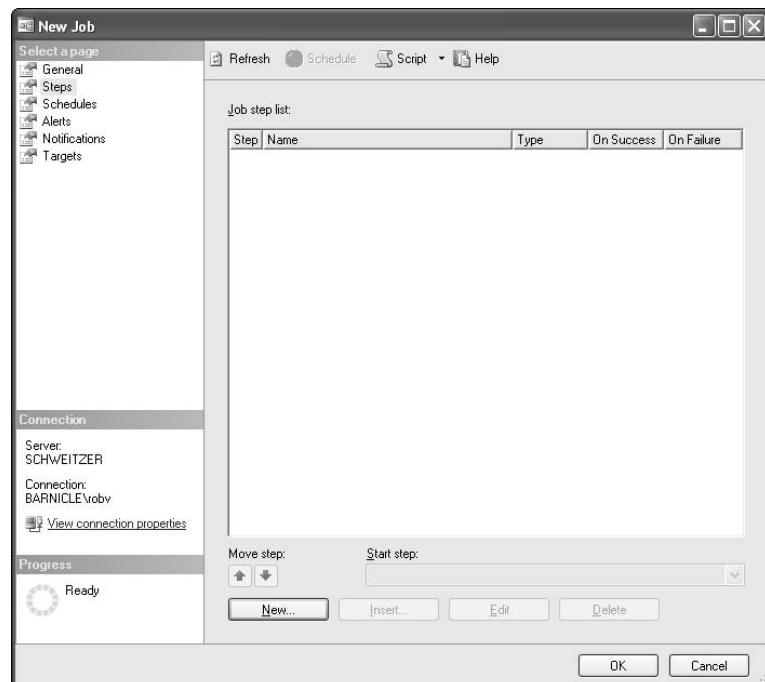


Figure 24-4

Chapter 24

To add a new step to our job, we just click on the New button and fill in the new dialog box, shown in Figure 24-5. We'll use a T-SQL statement to raise a bogus error just so we can see that things are really happening when we schedule this job. Note, however, that there is an Open button to the left of the command box—you can use this to import SQL Scripts that you have saved in files.

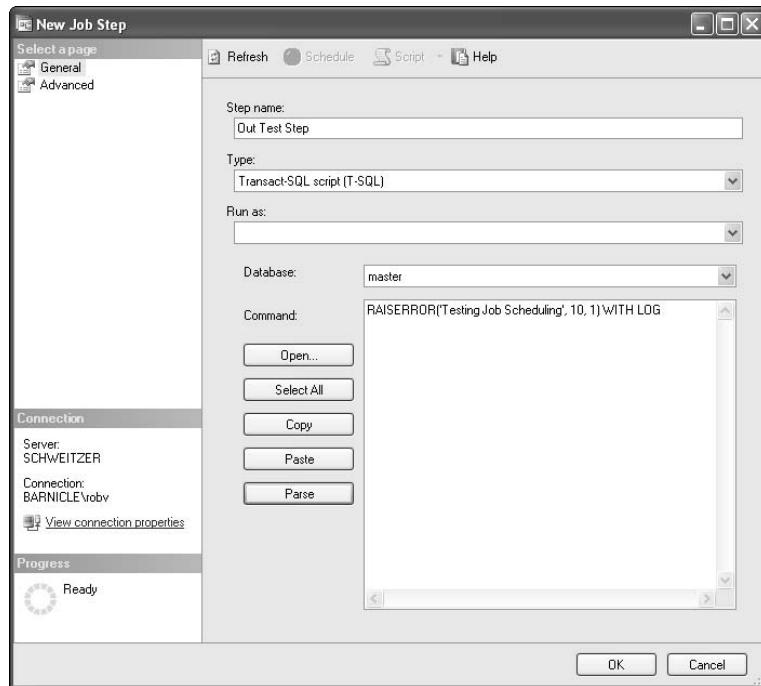


Figure 24-5

Let's go ahead and move on to the Advanced tab for this dialog, shown in Figure 24-6—it's here that we really start to see some of the cool functionality that our job scheduler offers.

Notice several things in this dialog:

- You can automatically set the job to retry at a specific interval if the task fails.
- You can choose what to do if the job succeeds or fails. For each result (success or failure), you can:
 - Quit reporting success
 - Quit reporting failure
 - Move on to the next step
- You can output results to a file. (This is very nice for auditing.)
- You can impersonate another user (for rights purposes). Note that you have to have the rights for that user. Because we're logged in as a sysadmin, we can run the job as the dbo or just about anyone. The average user would probably only have the guest account available (unless they were the dbo)—but, hey, in most cases a general user shouldn't be scheduling his or her own jobs this way anyway (let your client application provide that functionality).

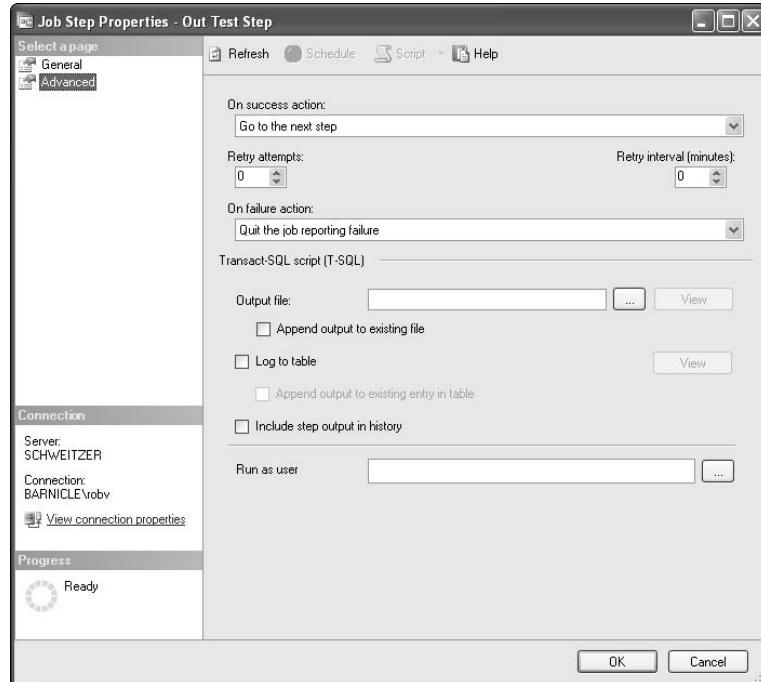


Figure 24-6

Okay, so there's little chance that our `RAISERROR` statement is going to fail, so we'll just take the default of "Quit the job reporting failure" on this one (we'll see other possibilities later in the chapter when we come to backups).

That moves us back to the main New Job dialog, and we're now ready to move on to the Schedules node, shown in Figure 24-7.

In this dialog, we can manage one or more scheduled times for this job to run. To actually create a new scheduled time for the job to run, we need to click on the New button. That brings up yet another dialog, shown in Figure 24-8.

I've largely filled this one out already (lest you get buried in a sea of screenshots), but it is from this dialog that we create a new schedule for this job. Recurrence and frequency are set here.

The frequency side of things can be a bit confusing because of the funny way that they've worded things. If you want something to run at multiple times every day, then you need to set the job to Occur Daily—every 1 day. This seems like it would run only once a day, but then you also have the option of setting whether it runs once or on an interval. In our case, we want to set our job to run every 5 minutes.

Now we're ready to move on to the next node of our job properties—alerts, shown in Figure 24-9.

From here, we can select what alerts we want to make depending on what happens. Choose Add and we get yet another rich dialog, shown in Figure 24-10.

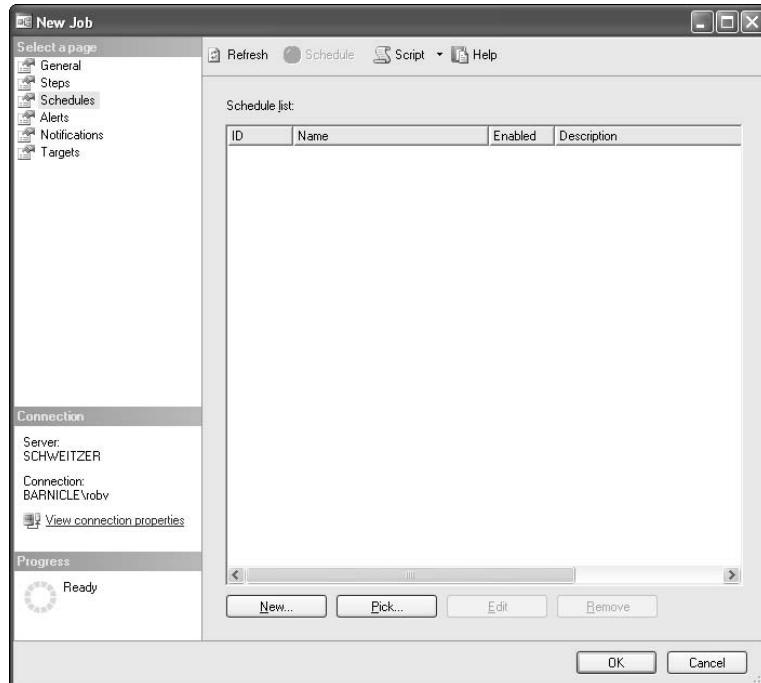


Figure 24-7

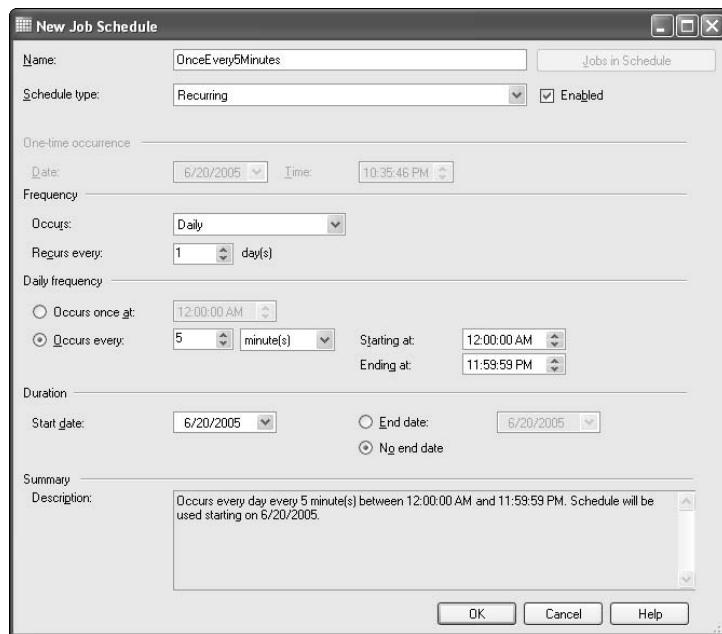


Figure 24-8

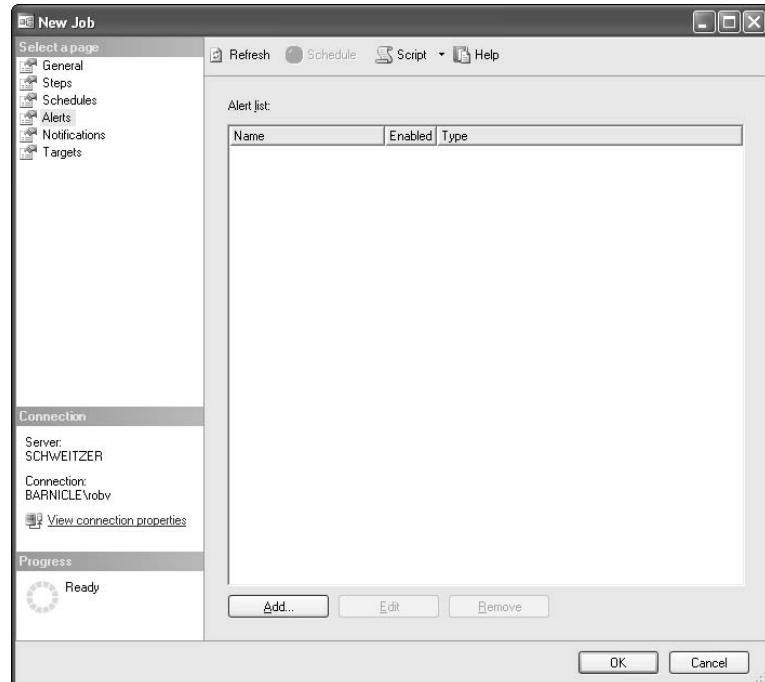


Figure 24-9

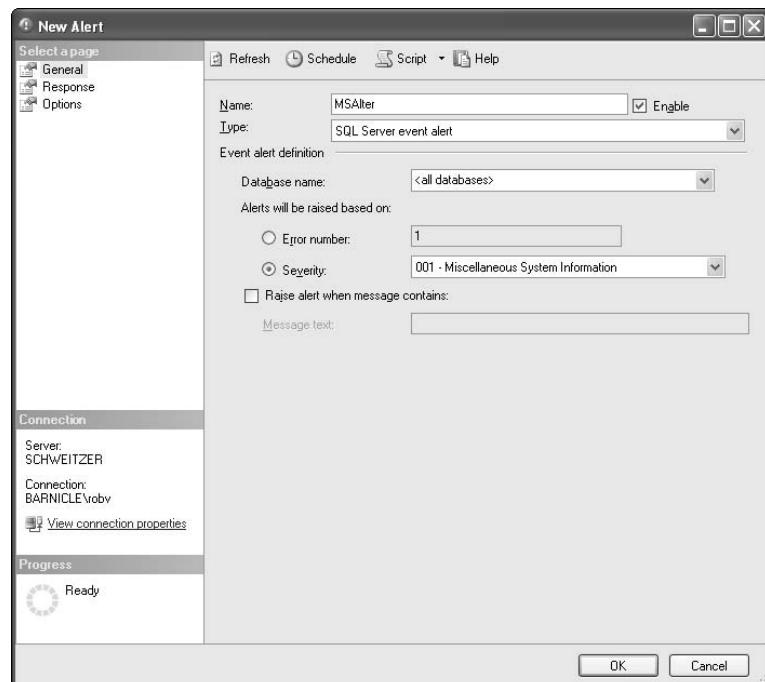


Figure 24-10

Chapter 24

Our first node—General—is going to let us fill out some of the basics. We can, for example, limit this notification to one particular database. We also define just how severe the condition needs to be before the alert will fire (in terms of severity of the error).

From there, it is on to the Response node (see Figure 24-11).

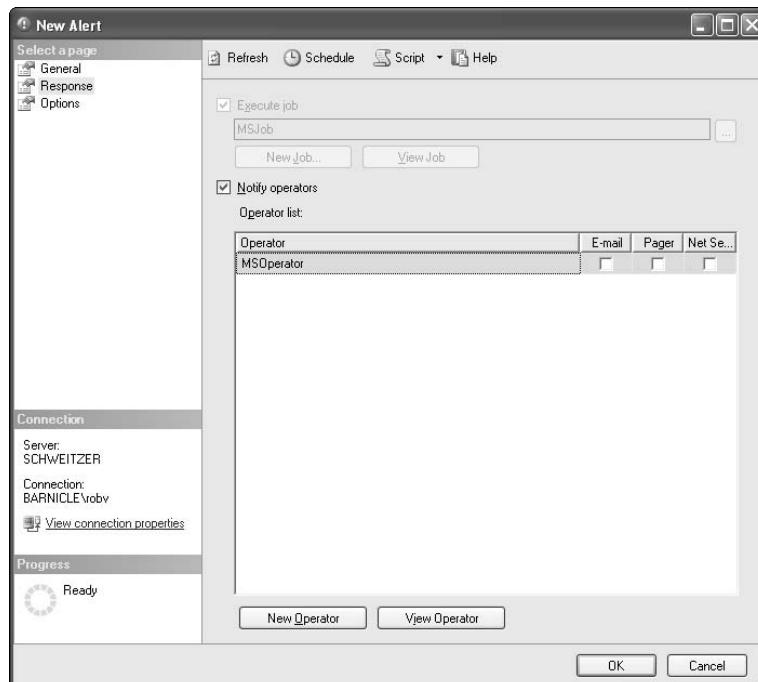


Figure 24-11

Notice that I was able to choose the operator that we created earlier in the chapter. It is through the definitions of these operators that the SQL Server Agent knows what e-mail address or netsend address to make the notification to. Also notice that we have control, on the right-hand side, over how our operator is notified.

Last, but not least, we have the Options node (see Figure 24-12).

Finally, we can go back to the Notifications node of the main New Job dialog (see Figure 24-13).

This window lets you bypass the older alerts model and define a response that is specific to this one job—we'll just stick with what we already have for now, but you could define specific additional notifications in this dialog.

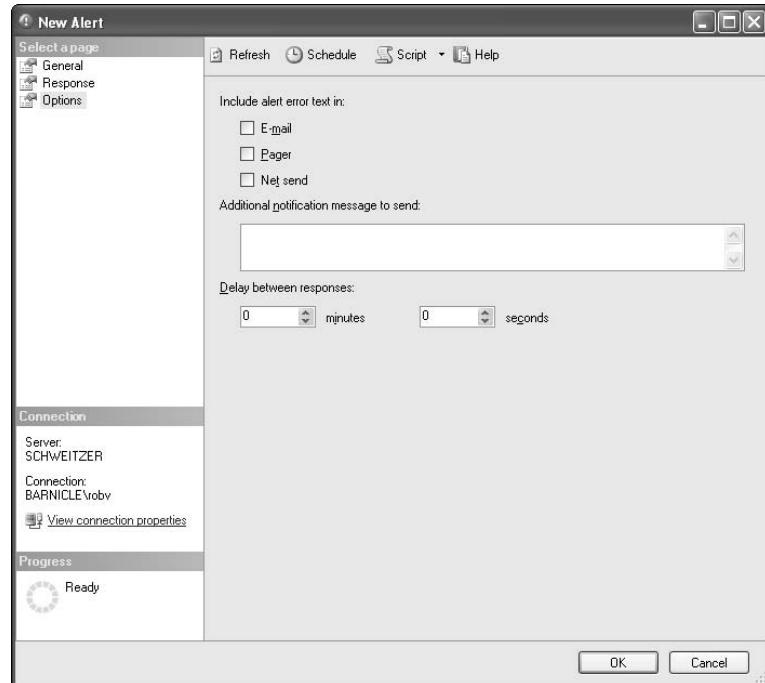


Figure 24-12

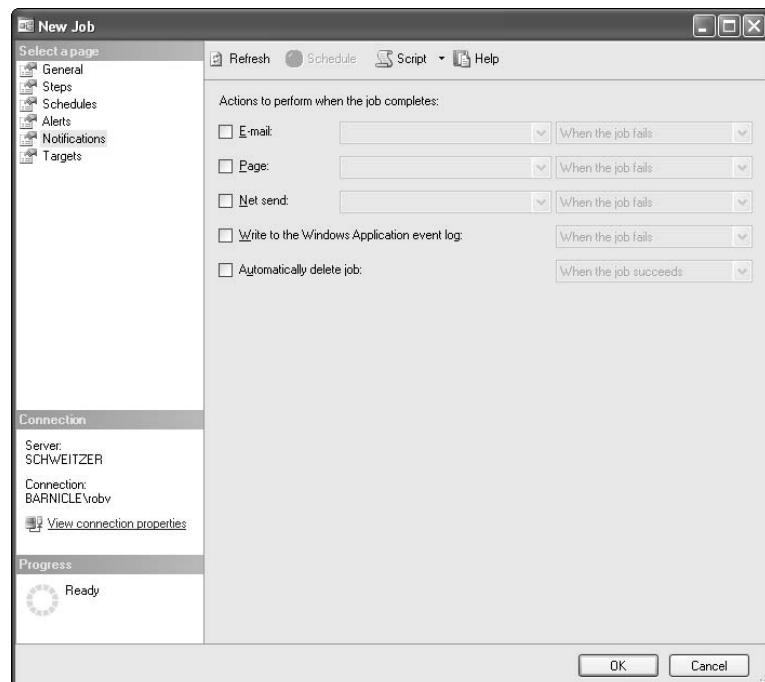


Figure 24-13

At this point, you are ready to say OK and exit the dialog. You'll need to wait a few minutes before the task will fire, but you should start to see log entries appear every five minutes in the Windows event log. You can look at this by navigating to Start→Programs→Administrative Tools→Event Viewer. You'll need to switch the view to use the Application log rather than the default System log.

Don't forget that, if you're going to be running scheduled tasks like this one, you need to have the SQL Server Agent running in order for them to be executed. You can check the status of the SQL Server Agent by running the SQL Server Configuration Manager and selecting the SQL Server Agent service, or by navigating to the SQL Server Agent node of the Object Explorer in Management Studio.

Also, don't forget to disable this job (right-click on the job in Management Studio after you've seen that it's working the way you expect). Otherwise, it will just continue to sit there and create entries in your Application Log—eventually, the Application Log will fill up and you can have problems with your system.

Creating Jobs and Tasks Using T-SQL

Before we get started, I want to point out that using T-SQL for this kind of stuff (creating scheduled jobs and tasks) is not usually the way things are done on a day-to-day basis. Most jobs wind up being scheduled by the DBA based on a specific need and a specific schedule that is required. If you're not in a situation where you need to script the installation of tasks, then you may want to just skip this section (it's a lot to learn if you aren't going to use it!). That being said, there can be times where your end users won't have a DBA handy (small shops, for example, often don't have anything even remotely resembling a DBA), so you'll want to script some jobs to help out unsophisticated users.

Automating the creation of certain jobs is a very frequent area of oversight in installation procedures—particularly for shrink-wrap software. If you're working in some form of consulting or private IS shop environment, then there's a good chance that you are going to need to take care of scheduling all the needed tasks when you do the install. With shrink-wrap software, however, you often aren't at all in control of the installation process—indeed, you may be hundreds or thousands of miles away from the install and may not even know that it's happening.

How then do you make sure that basic tasks (like backups, for example) get done? You can make it part of your installation process.

Jobs can be added to SQL Server using T-SQL by using three different stored procedures:

- ❑ `sp_add_job`—This creates the actual job.
- ❑ `sp_add_job_step`—This creates a task within the above job.
- ❑ `sp_add_jobschedule`—This determines when the job will run.

Each of these builds a piece of the overall execution of the scheduled task much as the different tabs in Management Studio did. The next sections take a look at each individually.

All jobs and tasks are stored in the `msdb` database. As such, you'll need to make sure that `msdb` is the current database (utilizing the `USE` command) when calling any of these sprocs.

sp_add_job

This one creates the top-level of a hierarchy and establishes who owns the job and how notifications should be handled. There are quite a few parameters, but most of them are fairly easy to figure out:

```
sp_add_job [@job_name =] '<job name>'  
[,@enabled =] <0 for no, 1 for yes>  
[,@description =] '<description of the job>'  
[,@start_step_id =] <ID of the step you want to start at>  
[,@category_name =] '<category>'  
[,@category_id =] <category ID>  
[,@owner_login_name =] '<login>'  
[,@notify_level_eventlog =] <eventlog level>  
[,@notify_level_email =] <email level>  
[,@notify_level_netsend =] <netsend level>  
[,@notify_level_page =] <page level>  
[,@notify_email_operator_name =] '<name of operator to email>'  
[,@notify_netsend_operator_name =] '<name of operator for network message>'  
[,@notify_page_operator_name =] '<name of operator to page>'  
[,@delete_level =] <delete level>  
[,@job_id =] <job id> OUTPUT
```

Again, most of the parameters here are self-describing, but let's touch on some of the more sticky ones.

- ❑ `@start_step_id`—This one is going to default to 1, and that's almost always going to be the place to leave it. We'll be adding steps shortly, but those steps will have identifiers to them, and this just lets the SQL Server Agent know where to begin the job.
- ❑ `@category_name`—This one equates directly with the category we saw in Management Studio. It will often be none (in which case, see `@category_ID`) but could be a Database Maintenance (another common choice), Full Text, Web Assistant, Replication or a category that you add yourself using `sp_add_category`.
- ❑ `@category_id`—This is just a way of providing a category without being dependent on a particular language. If you don't want to assign any particular category, then I recommend using this option instead of the name and supplying a value of either 0 (Uncategorized, but runs local) or 1 (Uncategorized Multi-Server).
- ❑ `@notify_level_eventlog`—For each type of notification, this determines under what condition the notification occurs. To use this sproc, though, we need to supply some constant values to indicate when we want the notification to happen. The constants are:

Constant Value	When the Notification Occurs
0	Never
1	When the task succeeds
2	When the task fails (this is the default)
3	Every time the task runs

Chapter 24

- ❑ @job_id—This is just a way of finding out what job ID was assigned to your newly created job—you'll need this value when you go to create job steps and the job schedule(s). The big things on this one is to remember to:
 - ❑ Receive the value into a variable so you can reuse it.
 - ❑ Remember that the variable needs to be of type `uniqueidentifier` rather than the types you might be more familiar with at this point.

Note that all the non-level “notify” parameters are expecting an operator name—you should create your operators before running this sproc.

So, let's create a job to test this process out. What we're going to do here is create a job that's nearly identical to the job we created Management Studio.

First, we need to create our top level job. All we're going to do for notifications is to send a message on failure to the Windows Event Log. If you have got Database Mail set up, then feel free to add in notification parameters for your operator.

```
USE msdb

DECLARE @JobID  uniqueidentifier

EXEC sp_add_job
    @job_name = 'TSQLCreatedTestJob',
    @enabled = 1,
    @notify_level_eventlog = 3,
    @job_id = @JobID OUTPUT

SELECT 'JobID is ' + CONVERT(varchar(128),@JobID)
```

Now, execute this, and you should wind up with something like this:

```
-----  
JobID is 83369994-6C5B-45FA-A702-3511214A2F8A  
(1 row(s) affected)
```

Note that your particular GUID will be different from the one I got here (remember that GUIDs are effectively guaranteed to be unique across time and space). You can either use this value or you can use the job name to refer to the job later (I happen to find this a lot easier, but it can create problems when dealing with multiple servers).

sp_add_jobserver

This is a quick-and-dirty one. We've now got ourselves a job, but we don't have anything assigned for it to run against. You see, you can create a job on one server but still run it against a completely different server if you choose.

In order to target a particular server, we'll use a sproc (in msdb still) called `sp_add_jobserver`. The syntax is the easiest by far of any we'll be looking at in this section, and looks like this:

```
sp_add_jobserver [@job_id =] <job id>|[@job_name =] '<job name>',
[@server_name =] '<server>'
```

Note that you supply either the job ID or the job name — not both.

So, to assign a target server for our job, we need to run a quick command:

```
USE msdb
EXEC sp_add_jobserver
    @job_name = 'TSQLCreatedTestJob',
    @server_name = "(local)"
```

Note that this will just point at the local server regardless of what that server is named. We could have also put the name of another valid SQL Server in to be targeted.

sp_add_jobstep

The second step in the process is to tell the job specifically what it is going to do. At the moment, all we have in our example is just the shell — the job doesn't have any tasks to perform, and that makes it a very useless job indeed. There is a flip side to this though — a step can't even be created without some job to assign it to.

The next step then is to run `sp_add_jobstep`. This is essentially adding a task to the job. If we had multiple steps we wanted the job to do, then we would run this particular sproc several times.

The syntax looks like this:

```
sp_add_jobstep [@job_id =] <job ID> | [@job_name =] '<job name>']
    [,[@step_id =] <step ID>]
    [,[@step_name =] '<step name>']
    [,[@subsystem =] '<subsystem>']
    [,[@command =] '<command>']
    [,[@additional_parameters =] '<parameters>']
    [,[@cmdexec_success_code =] <code>]
    [,[@on_success_action =] <success action>]
    [,[@on_success_step_id =] <success step ID>]
    [,[@on_fail_action =] <fail action>]
    [,[@on_fail_step_id =] <fail step ID>]
    [,[@server =] '<server>']
    [,[@database_name =] '<database>']
    [,[@database_user_name =] '<user>']
    [,[@retry_attempts =] <retry attempts>]
    [,[@retry_interval =] <retry interval>]
    [,[@os_run_priority =] <run priority>]
    [,[@output_file_name =] '<file name>']
    [,[@flags =] <flags>]
```

Not as many of the parameters are self-describing here, so let's look at the more confusing ones in the list:

Chapter 24

- ❑ @job_id vs. @job_name — This is actually a rather odd sproc in the sense that it expects you to enter one of the first two parameters, but not both. You can either attach this step to a job by its GUID (as you saved from the last sproc run) or by the job name.
- ❑ @step_id — All the steps in any job have an ID. SQL Server assigns these IDs automatically as you insert the steps. So why, if it does it automatically, do we have a parameter for it? That's in case we want to insert a step in the middle of a job. If there are already numbers 1–5 in the job, and we insert a new step and provide a step ID of 3, then our new step will be assigned to position number 3. The previous step 3 will be moved to position 4 with each succeeding step being incremented by 1 to make room for the previous step.
- ❑ @step_name — Is what it says — the name of that particular task. Just be aware that there is no default here — you must provide a step name.
- ❑ @subsystem — This ties in very closely to job categories and determines which subsystem within SQL Server (such as the replication engine, or the command line — the DOS prompt — or VB Script engine) is responsible for executing the script. The default is that you're running a set of T-SQL statements. The possible sub-systems are:

SubSystem	Description
ACTIVESCRIPTING	The scripting engine (VB Script).
CMDEXEC	Gives you the capability to execute compiled programs or batch files from a command (DOS) prompt.
DISTRIBUTION	The Replication Distribution Agent.
LOGREADER	Replication Log Reader Agent.
MERGE	The Replication Merge Agent.
SNAPSHOT	The Replication Snapshot Agent.
TSQL	A T-SQL batch. This is the default.

- ❑ @command — This is the actual command you're issuing to a specific sub-system. In our example, this is going to be the RAISERROR command just like we issued when using Management Studio, but it could be almost any T-SQL commands. What's cool here is that there are some system-supplied values you can use in your commands. You place these in the middle of your scripts as needed, and they are replaced at run time (we'll make use of this in our example). The possible system-supplied values are:

Tag	Description
[A-DBN]	Substitutes in the database name.
[A-SVR]	Substitutes the server name in the place of the tag.
[A-ERR]	Error number.
[A-SEV]	Error severity.
[A-MSG]	The message text from the error.

Tag	Description
[DATE]	Supplies the current date (in YYYYMMDD format).
[JOBID]	Supplies the current Job ID.
[MACH]	The current computer name.
[MSSA]	Master SQL Server Agent name.
[SQLDIR]	The directory in which SQL Server is installed (Usually C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL).
[STEPCT]	A count of the number of times this step has executed (excluding retries). You could use this one to keep count of the number of executions and force the termination of a multistep loop.
[STEPID]	Step ID.
[TIME]	The current time in HHMMSS format.
[STRTTM]	The start time for the job in HHMMSS format.
[STRTDT]	The start date for the job in YYYYMMDD format.

- @cmdexec_success_code—This is the value you expect to be returned by whatever command interpreter ran your job if the job ran successfully (applies only to command prompt sub-system). The default is zero.
- @on_success_action and @on_fail_action—This is where you say what to actually do at the success or failure of your step. Remember that at the job level we define what notifications we want to happen, but, at the step level, we can define how we want processing to continue (or end). For this parameter, you need to supply one of the following constant values:

Value	Description
1	Quit with success. This is the default for successful task executions.
2	Quit with failure. This is the default for failed tasks.
3	Go to the next step.
4	Go to a specific step as defined in <code>on_success_step_id</code> or <code>on_fail_step_id</code> .

- @on_success_step_id and @on_fail_step_id—What step you want to run next if you've selected option 4 in the preceding table.
- @server—The server the task is to be run against (you can run tasks on multiple target servers from a single master server).
- @database_name—The database to be set as current when the task runs.
- @retry_interval—This is set in minutes.

Chapter 24

- ❑ @os_run_priority—Ah, an undocumented feature. The default here is normal, but you can adjust how important Windows is going to think that your cmdExec (command line) scheduled task is. The possible values are:

Value	Priority
-15	Run at idle only
-1 thru -14	Increasingly below normal
0	Normal (this is the default)
1 thru 14	Increasingly above normal
15	Time critical

I just can't help but think of the old Lost in Space TV show here and think of the robot saying "DANGER Will Robinson—DANGER!" Don't take messing with these values lightly. If you're not familiar with the issues surrounding Windows thread priorities, I'd suggest staying as far away from this one as possible. Going with the higher values, in particular, can have a very detrimental impact on your system—including creating significant instabilities. When you say that this is the most important thing, remember that you are taking away some of the importance of things like operating system functions—not something that's smart to do. Stay clear of this unless you really know what you're doing.

- ❑ @flags—This one relates to the Output File parameter, and indicates whether to overwrite or append your output information to the existing file. The options are:

Value	Description
0	No option specified (currently, this means your file will be overwritten every time).
2	Append information to the existing file (if one exists).
4	Explicitly overwrite the file.

Okay, now that we've looked at the parameters, let's add a step to the job we created a short time ago:

```
sp_add_jobstep
    @job_name = 'TSQLCreatedTestJob',
    @step_name = 'This Is The Step',
    @command = 'RAISERROR
        (''RAISERROR (''TSQL Task is Job ID [JOBID] .'',10,1) WITH LOG'',10,1)
        WITH LOG',
    @database_name = 'AdventureWorks',
    @retry_attempts = 3 ,
    @retry_interval = 5
```

Technically speaking—our job should be able to be run at this point. The reason I say “technically speaking” is because we haven't scheduled the job, so the only way to run it is to manually tell the job to run. Let's take care of the scheduling issue, and then we'll be done.

sp_add_jobschedule

This is the last piece of the puzzle—we need to tell our job when to run. To do this, we'll make use of `sp_add_jobschedule`, which, like all the other sprocs we've worked on in this section, can only be found in the `msdb` database. Note that we could submit an entry from this sproc multiple times to create multiple schedules for our job. Keep in mind though that getting too many jobs scheduled can lead to a great deal of confusion, so schedule jobs wisely (for example, don't schedule one job for every day of the week when you can schedule a single job to run daily).

The syntax has some similarities to what we've already been working with, but adds some new pieces to the puzzle:

```
sp_add_jobschedule
    [@job_id =] <job ID>, | [@job_name =] '<job name>', [@name =] '<name>'
    [,[@enabled =] <0 for no, 1 for yes>]
    [,[@freq_type =] <frequency type>]
    [,[@freq_interval =] <frequency interval>]
    [,[@freq_subday_type =] <frequency subday type>]
    [,[@freq_subday_interval =] <frequency subday interval>]
    [,[@freq_relative_interval =] <frequency relative interval>]
    [,[@freq_recurrence_factor =] <frequency recurrence factor>]
    [,[@active_start_date =] <active start date>]
    [,[@active_end_date =] <active end date>]
    [,[@active_start_time =] <active start time>]
    [,[@active_end_time =] <active end time>]
```

Again, let's look at some of these parameters:

- ❑ `@freq_type`—Defines the nature of the intervals that are set up in the following parameters. This is another of those parameters that uses bit flags (although you should only use one at a time). Some of the choices are clear, but some aren't until you get to `@freq_interval` (which is next). Your choices are:

Value	Frequency
1	Once
4	Daily
8	Weekly
16	Monthly (fixed day)
32	Monthly (relative to <code>@freq_interval</code>)
64	Run at start of SQL Server Agent
128	Run when CPU is idle

- ❑ `@freq_interval`—Decides the exact days that the job is executed, but the nature of this value depends entirely on `@freq_type` (see the preceding point). This one can get kind of confusing; just keep in mind that it works with both `@freq_type` and `@frequency_relative_interval`. The interpretation works like this:

Chapter 24

freq_type Value	Matching freq_interval Values
1 (once)	Not Used
4 (daily)	Runs every x days where x is the value in the frequency interval
8 (weekly)	The frequency interval is one or more of the following: 1 (Sunday) 2 (Monday) 4 (Tuesday) 8 (Wednesday) 16 (Thursday) 32 (Friday) 64 (Saturday)
16 (monthly - fixed)	Runs on the exact day of the month specified in the frequency interval
32 (monthly - relative)	Runs on exactly one of the following: 1 (Sunday) 2 (Monday) 3 (Tuesday) 4 (Wednesday) 5 (Thursday) 6 (Friday) 7 (Saturday) 8 (Specific Day) 9 (Every Weekday) 10 (Every Weekend Day)
64 (Run at Agent startup)	Not Used
128 (Run at CPU idle)	Not Used

- ❑ @freq_subday_type—Specifies the units for @freq_subday_interval. If you’re running daily, then you can set a frequency to run within a given day. The possible values here are:

Value	Description
1	At the specified time
4	Every x minutes where x is the value of the frequency sub-day interval
8	Every x hours where x is the value of the frequency sub-day interval

- ❑ @freq_subday_interval—This is the number of @freq_subday_type periods to occur between each execution of the job (x in the above table).

- @freq_relative_interval — This is used only if the frequency type is monthly (relative) (32). If this is the case, then this value determines in which week a specific day of week job is run or flags things to be run on the last day of the month. The possible values are:

Value	Description
1	First Week
2	Second Week
4	Third Week
8	Fourth Week
16	Last Week or Day

- @freq_recurrence_factor — How many weeks or months between execution. The exact treatment depends on the frequency type and is applicable only if the type was weekly or monthly (fixed or relative). This is an integer value, and, for example, if your frequency type is 8 (weekly) and the frequency recurrence factor is 3, then the job would run on the specified day of the week every third week.

The default for each of these parameters is 0.

Okay, so let's move on to getting that job scheduled to run every five minutes as we did when using Management Studio.

```
sp_add_jobschedule
@job_name = 'TSQLCreatedTestJob',
@name = 'Every 5 Minutes',
@freq_type = 4,
@freq_interval = 1,
@freq_subday_type = 4,
@freq_subday_interval = 5,
@active_start_date = 20060101
```

Now, if you go and take a look at the job in Management Studio, you'll find that you have a job that is (other than the name) identical to the job we created directly in Management Studio. Our job has been fully implemented using T-SQL this time.

Maintaining and Deleting Jobs and Tasks

Maintaining jobs in Management Studio is pretty simple—just double-click on the job and edit it just as if you were creating a new job. Deleting jobs and tasks in Management Studio is simpler—just highlight the job and press the Delete button. After one confirmation, your job is gone.

Checking out what you have, editing it, and deleting it are all slightly trickier in T-SQL. The good news, however, is that maintaining jobs, tasks, and schedules works pretty much as creating did, and that deleting any of them is a snap.

Chapter 24

Editing and Deleting Jobs with T-SQL

To edit or delete each of the four steps we just covered for T-SQL, you just use (with one exception) the corresponding update sproc—the information provided to the update sproc completely replaces that of the original add (or prior updates)—or delete sproc. The parameters are the same as the add sproc for each:

If the Add Was	Then Update With	And Delete With
sp_add_job	sp_update_job	sp_delete_job
sp_add_jobserver	None (drop and add)	sp_delete_jobserver
sp_add_jobstep	sp_update_jobstep	sp_delete_jobstep
sp_add_jobschedule	sp_update_jobschedule	sp_delete_jobschedule

Backup and Recovery

No database-driven app should ever be deployed or sold to a customer without a mechanism for dealing with backup and recovery. As I've probably told people at least 1,000 times: You would truly be amazed at the percentage of database operations that I've gone into that do not have any kind of reliable backup. In a word: EEEeeeeek!

There is one simple rule to follow regarding backups—do it early and often. The follow up to this is to not just back up to a file on the same disk and forget it—you need to make sure that a copy moves to a completely separate place (ideally off-site) to be sure that it's safe. I've personally seen servers catch fire (the stench was terrible, as were all the freaked out staff). You don't want to find out that your backups went up in the same smoke that your original data did.

For apps being done by the relative beginner, then, you're probably going to stick with referring the customer or on-site administrator to SQL Server's own backup and recovery tools, but, even if you do, you should be prepared to support them as they come up to speed in its use. In addition, there is no excuse for not understanding what it is the customer needs to do.

Creating a Backup—a.k.a. “A Dump”

Creating a backup file of a given database is actually pretty easy. Simply navigate in the Object Explorer to the database you're interested in, and right-click.

Now choose Tasks and Back Up, as shown in Figure 24-14.

And you'll get a dialog that lets you define pretty much all of the backup process, as in Figure 24-15.

The first setting here is pretty self-explanatory—what database you want to back up. From there, however, things get a bit trickier.

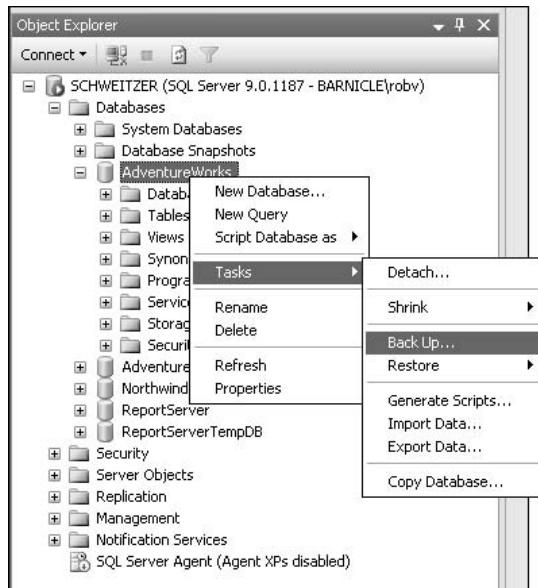


Figure 24-14

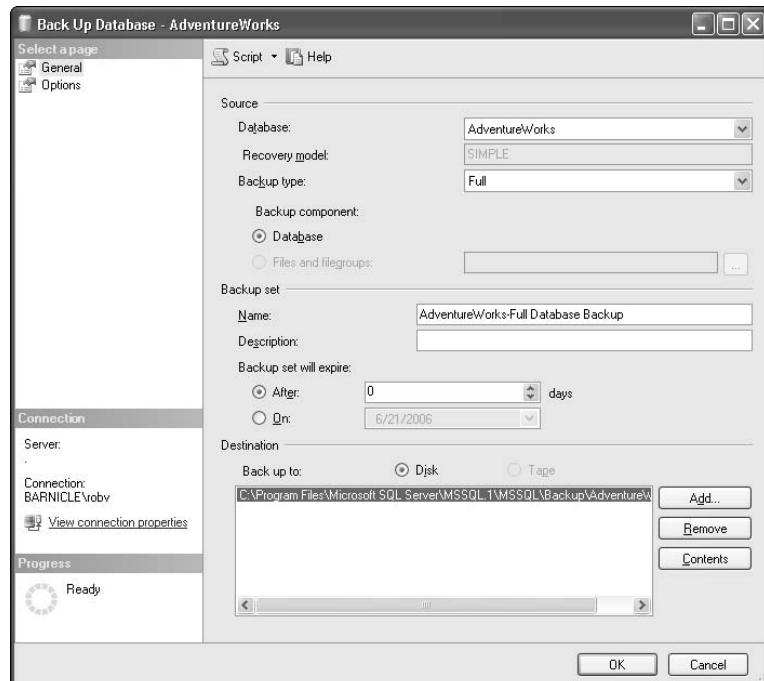


Figure 24-15

Getting into the items that may not yet make sense, first up is the Recovery Model. The Recovery Model field here is just notifying you of what the database you've selected for backup is set to—it is actually a database level setting. We're going to defer discussion of what this is for a bit—we'll get to it in the next section when we talk about backing up transaction logs.

Now, those are the simple parts, but let's break down some of the rest of the options that are available.

Backup Type

First of the choices to be made is the Backup Type. Depending on the recovery model for your database (again, be patient with me, we'll get there on what this is!), you'll have either two or three types of backups available:

- ❑ **Full**—This is just what it sounds like—a full backup of your actual database file as it is as of the last transaction that was committed prior to you issuing the Backup command.
- ❑ **Differential**—This might be referred to as a “backup *since*” backup. When you take a differential backup, it only writes out a copy of the extents (see Chapter 8 if you've forgotten!) that have changed since you did the last full backup. These typically run much faster than a Full backup and will take up less space. How much less? Well, that depends on how much your data actually changes. For very large databases where backups can take a very long time to run, it is very common to have a strategy where you take a full backup only once a week or even only once a month, and then take differential backups in between to save both space and time.
- ❑ **Transaction Log**—This is again just what it sounds like—a copy of the transaction log. This option will only show up if your database is set to Full or Bulk logging (this option is hidden if you are using simple logging). Again, full discussion of what these are is coming up shortly.

A subtopic of the Backup Type is the Backup Component, which applies only to Full and Differential backups.

For purposes of this book, we should pretty much just be focused on backing up the whole database. That said, you'll notice another option titled “Files and Filegroups.” Back in Chapter 4, we touched briefly on the idea of filegroups and individual files for data to be stored in. This option lets you select just one file or filegroup to participate in for this backup—I highly recommend avoiding this option until you have graduated to the “expert” class of SQL Server user.

Again, I want to stress avoiding this particular option until you've got yourself something just short of a doctorate in SQL Server backups. These are special use—designed to help with very large database installations (figure terabytes) that are in high-availability scenarios. There are major consistency issues to be considered when taking and restoring from this style of backup, and they are not for the faint of heart.

Backup Set

A *backup set* is basically a single name used to refer to one or more destinations for your backup.

SQL Server allows for the idea that your backup may be particularly large or that you may otherwise have reason to back up across multiple devices—be it drives or tapes. When you do this, however, you need to have all of the devices you used as a destination available in order to recover from any of them—that is, they are a “set.” The backup set essentially holds the definition of what destinations were involved in

your particular backup. In addition, a backup set contains some property information for your backup. You can, for example, identify an expiration date for the backup.

Destination

This is where your data is going to be backed up to. Here is where you define potentially several destinations to be utilized for one backup set. For most installations this will be a file location (which later will be moved to tape), but you can also define a backup device that would let you go directly to tape or similar backup device.

Options

In addition to those items we just covered from the General node of the dialog, you also have a node that lets you set other miscellaneous options. Most of these are fairly self-describing. Of particular note, however, is the Transaction Log area.

Schedule

With all this set up, wouldn't it be nice to set up a job to run this backup on a regular basis? Well, the Schedule button up at the top of the dialog is meant to facilitate your doing just that. Press it, and it will bring up the Job Schedule dialog you saw earlier in the chapter. You can then define a regular schedule to run the backup you just defined.

Creating a Device Using T-SQL

Creating a device using T-SQL is almost as easy and brings a touch of nostalgia compared to the older versions of SQL server when a backup was still called a *dump*. A dump is now a backup—it's that simple. To create the backup device, we use a legacy system sproc called `sp_addumpdevice`. The syntax looks like this:

```
sp_addumpdevice  
    [@devtype =] '<type>',  
    [@logicalname =] '<logical name>',  
    [@physicalname =] '<physical name>'
```

The type is going to be one of the following:

- Disk**—A local hard drive
- Tape**—A tape drive

The logical name is the name that you will use in the `BACKUP` and `RESTORE` statements that we will look at shortly.

The physical name is either the name of the tape device (must be one on the local server) or the physical path to the disk volume you're backing up to.

As an example, we'll create a backup device called `TSQLBackupDevice`:

```
EXEC sp_addumpdevice 'DISK', 'TSQLBackupDevice',  
    'C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\BACKUP\TSQLBackupDevice.bak'
```

And we're done!

Backing Up Using T-SQL

To back up the database or the log in T-SQL, we make use of the BACKUP command. The syntax for BACKUP works almost, but not quite, the same depending on whether you're backing up the database or the log. The syntax looks like this.

```
BACKUP DATABASE|LOG <database name>
{WITH
    NO_LOG|TRUNCATE_ONLY}
| TO <backup device> [,...n]
[WITH
[BLOCKSIZE = <block size>]
[[,] CHECKSUM | NO CHECKSUM ]
[[,] STOP_ON_ERROR | CONTINUE_AFTER_ERROR ]
[[,] DESCRIPTION = <description>]
[[,] DIFFERENTIAL]
[[,] EXPIREDATE = <expiration date> | RETAINDAYS = <days>]
[[,] PASSWORD = <password>]
[[,] FORMAT|NOFORMAT]
[[,] INIT|NOUNIT]
[[,] MEDIADESCRIPTION = <description>]
[[,] MEDIANAME = <media name>]
[[,] MEDIAPASSWORD = <media password>]
[[,] [NAME = <backup set name>]
[[,] REWIND|NOREWIND ]
[[,] NOSKIP|SKIP]
[[,] NOUNLOAD|UNLOAD]
[[,] RESTART]
[[,] STATS [= <percentage>]]
[[,] COPY_ONLY]
```

Let's look at some of the parameters:

- ❑ <backup device>—That's right; you can back up to more than one device. This creates what's called a *media set*. These can really speed up your backups if the media are spread over several disks, as it creates a parallel load situation—you're not bound by the I/O limitations of any of the individual devices. However, beware—you must have the entire media set intact to restore from this kind of backup.
- ❑ BLOCKSIZE—This is automatically determined in a hard drive backup, but, for tape, you need to provide the correct block size—contact your vendor for help on this one.
- ❑ DIFFERENTIAL—This is to perform a *differential backup*. A differential backup only backs up the data that is changed since your last full backup. Any log or other differential backup is ignored—any row/column changed, added, or deleted since the last full backup is included in the new backup. Differential backups have the advantage of being much faster to create than a full backup and much faster to restore than applying each individual log when restoring.
- ❑ EXPIREDATE/RETAINDAYS—You can have your backup media expire after a certain time. Doing so lets SQL Server know when it can overwrite the older media.
- ❑ FORMAT/NOFORMAT—Determines whether the media header (required for tapes) should be rewritten or not. Be aware that formatting affects the entire device—this means that formatting for one backup on a device destroys all the other backups on that device as well.

- INIT/NOINIT—Overwrites the device data but leaves the header intact.
- MEDIADESCRIPTION and MEDIANAME—Just describe and name the media—maximum of 255 characters for a description and 128 for a name.
- SKIP/NOSKIP—Decides whether or not to pay attention to the expiration information from previous backups on the tape. If SKIP is active, then the expiration is ignored so the tape can be overwritten.
- UNLOAD/NOUNLOAD—Used for tape only—determines whether to rewind and eject the tape (UNLOAD) or leave it in its current position (NOUNLOAD) after the backup is complete.
- RESTART—Picks up where a previously interrupted backup left off.
- STATS—Displays a progress bar indicating progress as the backup runs.
- COPY_ONLY—Creates a backup but does not affect any other backup sequence you have in any way. For example, logs are differential backups will continue as if the copy backup had never occurred.

Now let's try one out for a true backup:

```
BACKUP DATABASE AdventureWorks
TO TSQLBackupDevice
WITH
    DESCRIPTION = 'My what a nice backup!',
    STATS
```

We now have a backup of our AdventureWorks database.

SQL Server is even nice enough to provide progress messages as it processes the backup:

```
10 percent processed.
20 percent processed.
30 percent processed.
40 percent processed.
50 percent processed.
60 percent processed.
70 percent processed.
80 percent processed.
90 percent processed.
Processed 21816 pages for database 'AdventureWorks', file 'AdventureWorks_Data' on
file 1.
100 percent processed.
Processed 25 pages for database 'AdventureWorks', file 'AdventureWorks_Log' on
file 1.
BACKUP DATABASE successfully processed 21841 pages in 9.401 seconds (19.032
MB/sec).
```

It's that simple, so let's follow it up with a simple backup of the log:

```
BACKUP LOG AdventureWorks
TO TSQLBackupDevice
WITH
    DESCRIPTION = 'My what a nice backup of a log!',
    STATS
```

It's worth noting that you can't do a backup of a log while the database recovery model is set to Simple. To change this to a different recovery model, right-click on the AdventureWorks database, select Properties and the Options tab—in T-SQL, use the `sp_dboption` system sproc. If you think about it, this makes sense given that your log is always going to be essentially free of any committed transactions.

It's also worth noting that backups work just fine while there are users in your database. SQL Server is able to reconcile the changes that are being made by knowing the exact point in the log that the backup was begun, and using that as a reference point for the rest of the backup.

Recovery Models

Well, I spent most of the last section promising that we would discuss them, so it's time to ask: What is a recovery model?

Well, back in the chapter on transactions, we talked about the transaction log. In addition to keeping track of transactions to deal with transaction rollback and atomicity of data, transaction logs are also critical to being able to recover data right up to the point of system failure.

Imagine for a moment that you're running a bank. Let's say you've been taking deposits and withdrawals for the last six hours—the time since your last full backup was done. Now, if your system went down, I'm guessing you're not going to like the idea of going to last night's backup and losing all track of what money went out the door or came in during the interim. See where I'm going here? You really need every moment's worth of data.

Keeping the transaction log around gives us the ability to "roll forward" any transactions that happened since the last full or differential backup was done. Assuming both the data backup *and* the transaction logs are available, you should be able to recover right up to the point of failure.

The recovery model determines how long and what types of log records are kept—there are three options:

- ❑ **Full**—This is what it says—everything is logged. Under this model, you should have no data loss in the event of system failure, assuming you had a backup of the data available and have all transaction logs since that backup. If you are missing a log or have one that is damaged, then you'll be able to recover all data up through the last intact log you have available. Keep in mind, however, that as keeping everything suggests, this can take up a fair amount of space in a system that receives a lot of changes or new data.
- ❑ **Bulk-Logged**—This is like "Full recovery light." Under this option, regular transactions are logged just as they are with the Full recovery method, but bulk operations are not. The result is that, in the event of system failure, a restored backup will contain any changes to data pages that did not participate in bulk operations (bulk import of data or index creation for example), but any bulk operations must be redone. The good news on this one is that bulk operations perform *much* better. This performance comes with risk attached, so your mileage may vary. . . .

- ❑ **Simple**—Under this model, the transaction log essentially exists merely to support transactions as they happen. The transaction log is regularly truncated—with any completed or rolled back transactions essentially being removed from the log (not quite that simple, but that is the effect). This gives us a nice tight log that is smaller and often performs a bit better, but the log is of zero use for recovery from system failure.

For most installations, Full recovery is going to be what you want to have for a production-level database—end of story.

Recovery

This is something of the reverse of the backup side of things. You've done your backups religiously, and now you want to restore one—either for recovery purposes or merely to make a copy of a database somewhere.

Once you have a backup of your database, it's fairly easy to restore it to the original location. To get started—it works much as it did for backup: navigate to the database you want to restore to and right-click—then select Tasks→Restore, and up comes your Restore dialog, as in Figure 24-16.

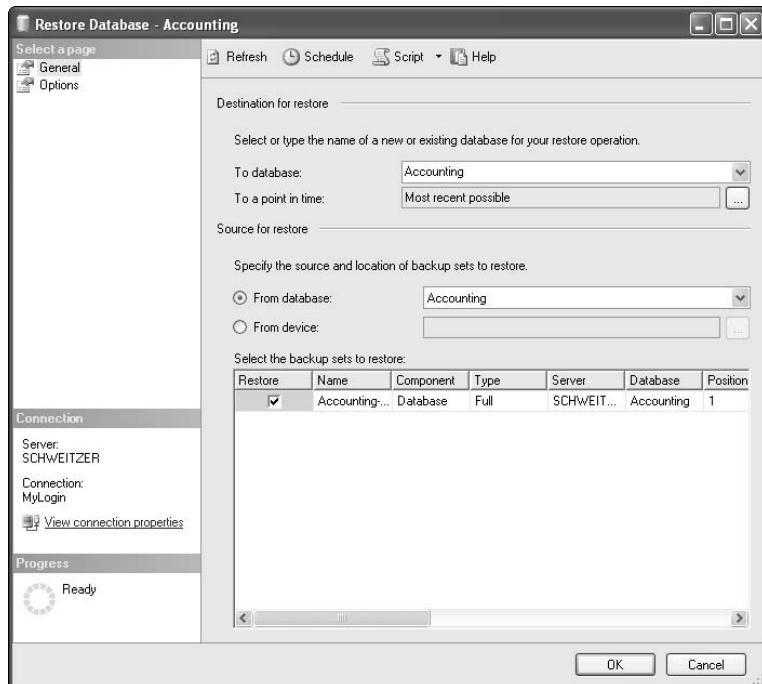


Figure 24-16

As long as what you're after is to take your old backup and slam it over the top of the database you made the backup of, this is pretty straightforward—simply say OK, and it should restore for you without issue.

Restoring to a Different Location

When things get tricky is when you want to change something about where you're restoring to. As part of the backup process, the backup knows what the name of the database was that was backed up, and, perhaps more important, it knows the path(s) to the physical files that it was supposed to be using.

Changing the destination database name is right there — no biggie — the problem is that changing the destination database name does nothing to change what physical files (the .MDF and .LDF files) it's going to try to store to. To deal with this, go to the options node of the Restore dialog.

Again, most of the options here are self-describing, but, in particular, notice the "Restore As" column. In this part of the dialog, you can replace every original file's destination, location, and name — providing you with the way to deal with restoring multiple copies of a database to the same server (perhaps for test purposes) or installing your database to a new volume or even a new system.

Recovery Status

This one is merely about the state you want to have the database in when you are done with this restore. This has particular relevance when you are restoring a database and still have logs to apply to the database later.

If you go with the default option (which translates to using the WITH RECOVERY option if you were using T-SQL), then the database will immediately be in a full online status when the restore operation is complete. If, for example, you wanted to restore logs after your initial restore was done, you would want to select one of the two other options. Both of these prevent updates happening to the database and leave it in a state where more recovery can be done — the difference is merely one of whether users are allowed to access the database in a "read-only" mode or whether the database should appear as still being offline.

The issue of availability is a larger one than you probably think it is. As big of a deal as I'm sure it already seems, it's really amazing how quickly users will find their way into your system when the restore operation suddenly marks the database as available. Quite often, even if you know that you will be "done" after the current restore is done, you'd like a chance to look over the database prior to actual users being in there. If this is the case, then be sure to use the NO RECOVERY method of restoring. You can later run a restore that is purely for a WITH RECOVERY option, and get the database fully back online once you're certain you have things just as you want them.

Restoring Data Using T-SQL

We use the RESTORE command to recover the data that we have in our backups. The syntax looks like this:

```
RESTORE DATABASE|LOG <database name>
    [FROM <backup_device> [,...n] ]
    [WITH
    [DBO_ONLY]
    [,] FILE = <file number>
    [,] MEDIANAME = <media name>
    [,] MOVE '<logical file name>' TO '<operating system file name>' [,...n]
    [,] {NORECOVERY|RECOVERY|STANDBY = <undo file name>}]
    [,] {NOUNLOAD|UNLOAD}
    [,] REPLACE
    [,] RESTART
    [,] STATS [= percentage]
```

```
[ [,] { STOPAT = { <date and time> }
      | STOPATMARK = { '<name of mark>' }
      [ AFTER <date and time> ]
      | STOPBEFOREMARK = { '<name of mark>' }
      [ AFTER <date and time> ]
]
```

Let's look at some of these options:

- DBO_ONLY—When the restore is done, the database will be set with the `dbo_only` database option turned on. This gives the dbo a chance to look around and test things out before allowing users back onto the system.

This is a biggie, and I very strongly recommend that you always use it. You would be amazed at how quickly users will be back on the system once it's backed up for even a moment. When a system is down, you'll find users very impatient to get back to work. They'll constantly be trying to log in, and they won't bother to ask if it's okay or not—they'll assume that when it's up, it's okay to go into it.

- FILE—You can back up multiple times to the same media. This option lets you select a specific version to restore. If this one isn't supplied, SQL Server will assume that you want to restore from the most recent backup.
- MOVE—Allows you to restore the database to a different physical file than the database was using when it was originally backed up.
- NORECOVERY/RECOVERY/STANDBY—RECOVERY and NORECOVERY are mutually exclusive. STANDBY works in conjunction with NORECOVERY. They work as follows:

Option	Description
NORECOVERY	Restores the database but keeps it marked as offline. Uncommitted transactions are left intact. This allows you to continue with the recovery process—for example, if you still have additional logs to apply.
RECOVERY	As soon as the restore command is done successfully, the database is marked as active again. Data can again be changed. Any uncommitted transactions are rolled back. This is the default if none of the options are specified.
STANDBY	STANDBY allows you to create an undo file so that the effects of a recovery can be undone. STANDBY allows you to bring the database up for read-only access before you have issued a RECOVERY (which means at least part of your data's been restored, but you aren't considering the restoration process complete yet). This allows users to make use of the system in a read-only mode while you verify the restoration process.

- REPLACE—Overrides the safety feature that prevents you from restoring over the top of an existing database.
- RESTART—Tells SQL Server to continue a previously interrupted restoration process.

Chapter 24

I'm going to go ahead and give you an example run of restoring the AdventureWorks database. Do not do this yourself unless you are absolutely certain that your backup was successful and is intact.

First, I'm going to drop the exiting AdventureWorks database:

```
USE master
DROP DATABASE AdventureWorks
```

Once that's done, I'll try to restore it using my RESTORE command:

```
RESTORE DATABASE AdventureWorks
    FROM TSQLBackupDevice
    WITH
        DBO_ONLY,
        NORECOVERY,
        STATS
```

I did my restore with NORECOVERY because I want to add another piece to the puzzle. My log will contain any transactions that happened between when my database or log was last backed up and when this log was backed up. I'm going to "apply" this log, and that should bring my database as up to date as I can make it:

```
RESTORE LOG AdventureWorks
    FROM TSQLBackupDevice
    WITH
        DBO_ONLY,
        NORECOVERY,
        STATS
```

Note that if I had several logs to apply from this one device, then I would have to name them as I wanted to apply them. They would also need to be applied in the order in which they were backed up.

Now, I could have turned everything on there, but I wanted to hold off for a bit before making the database active again. Even though I don't have any more logs to apply, I still need to re-run the RESTORE statement to make the database active again:

```
RESTORE LOG AdventureWorks WITH RECOVERY
```

I should now be able to test my database:

```
USE AdventureWorks
SELECT * FROM Region
```

And, sure enough, I get the results I'm looking for. Run a few SELECT statements to see that, indeed, our database was restored properly.

After you've checked things out, remember that I chose the DBO_ONLY option for all this. If we run sp_dboption, we'll see that no one else is able to get in:

```
EXEC sp_dboption
```

Look for the **dbo use only**:

```
Settable database options:  
-----  
ANSI null default  
ANSI nulls  
ANSI padding  
ANSI warnings  
arithabort  
auto create statistics  
auto update statistics  
autoclose  
autoshrink  
concat null yields null  
cursor close on commit  
db chaining  
dbo use only  
default to local cursor  
merge publish  
numeric roundabort  
offline  
published  
quoted identifier  
read only  
recursive triggers  
select into/bulkcopy  
single user  
subscribed  
torn page detection  
trunc. log on chkpt.
```

You must remember to turn that option off or your users won't be able to get into the system:

```
EXEC sp_dboption AdventureWorks, 'dbo use only', 'false'
```

You now have a restored an active database.

Index Maintenance

Back in Chapter 8, we talked about the issue of how indexes can become fragmented. This can become a major impediment to the performance of your database over time, and it's something that you need to have a strategy in place to deal with. Fortunately, SQL Server has commands that will reorganize your data and indexes to clean things up. Couple that with the job scheduling that we've already learned about, and you can automate routine defragmentation.

The commands that have to do with index defragmentation were altered fairly radically with this release of SQL Server. The workhorse of the old days was an option in what was sometimes known as the Database Consistency Checker—or DBCC. I see DBCC referred to these days as Database Console Command, but either way, what we're talking about is DBCC (specifically, DBCC INDEXDEFRAG and, to a lesser extent, DBCC DBREINDEX for our index needs). This has been replaced with the new ALTER INDEX command.

`ALTER INDEX` is the new workhorse of database maintenance. It is simultaneously much easier and slightly harder than `DBCC` used to be. Let's take a look at this one real quick, and then at how to get it scheduled.

ALTER INDEX

The command `ALTER INDEX` is somewhat deceptive in what it does. Up until now, `ALTER` commands have always been about changing the definition of our object. We `ALTER` tables to add or disable constraints and columns, for example. `ALTER INDEX` is different—it is all about maintenance and zero about structure. If you need to change the make-up of your index, you still need to either `DROP` and `CREATE` it, or you need to `CREATE` and use the `WITH DROP_EXISTING=ON` option.

The `ALTER INDEX` syntax looks like this:

```
ALTER INDEX { <name of index> | ALL }
    ON <table or view name>
    { REBUILD
        [ [ WITH ( <rebuild index option> [ ,...n ] ) ]
        | [ PARTITION = <partition number>
            [ WITH ( <partition rebuild index option>
                [ ,...n ] ) ] ] ]
        | DISABLE
        | REORGANIZE
            [ PARTITION = <partition number> ]
            [ WITH ( LOB_COMPACTION = { ON | OFF } ) ]
        | SET ( <set_index_option> [ ,...n ] )
    }
[ ; ]
```

A decent amount of this is fairly detailed “Realm of the advanced DBA” stuff—usually used on an ad hoc basis to deal with very specific problems—but there are some core elements here that should be part of our regular maintenance planning. We’ll start by looking at a couple of top parameters, and then look at the options that are part of our larger maintenance planning needs.

Index Name

You can name a specific index if you want to maintain one specific index, or use `ALL` to indicate that you want to perform this maintenance on every index associated with the named table.

Table or View Name

Pretty much just what it sounds like—the name of the specific object (table or view) that you want to perform the maintenance on. Note that it needs to be one specific table (you can’t feed it a list and say “do all of these please!”).

REBUILD

This is the “industrial strength” approach to fixing an index. If you run `ALTER INDEX` with this option, the old index is completely thrown away and reconstructed from scratch. The result is a truly optimized

index, where every page in both the leaf and non-leaf levels of the index have been reconstructed as you have defined them (either the defaults, or using switches to change things like the fill factor).

Careful on this one. As soon as you kick off a REBUILD, the index you are working on is essentially gone until the rebuild is complete. Any queries that relied on that index may become exceptionally slow (potentially by orders of magnitude). This is the sort of thing you want to test on an offline system first to have an idea how long it's going to take, and then schedule to run in off hours (preferably with someone monitoring it to be sure it's back online when peak hours come along).

This one can have major side effects while it runs, and thus it falls squarely in the domain of the database administrator in my not so humble opinion.

DISABLE

This one does what it says, only in somewhat drastic fashion. It would be nice if all this command did was take your index offline until you decided further what you want to do, but instead it essentially marks the index as unusable. Once an index has been disabled, it must be rebuilt (not reorganized, but rebuilt) before it will be active again.

This is one you're very, very rarely going to do yourself (you would more likely just drop the index)—it is far more likely to happen during a SQL Server upgrade or some other oddball situation.

Yet another BE CAREFUL!!! warning on this one. If you disable the clustered index for your table, it has the effect of disabling the table. The data will remain but will be inaccessible by all indexes (since they all depend on the clustered index) until you rebuild the clustered index.

REORGANIZE

BINGO!!! from the developer perspective. With REORGANIZE we hit much more of a happy medium in life. When you reorganize your index, you get a slightly less complete optimization than you get with a full rebuild, but one that occurs online (users can still utilize the index).

This should, if you're paying attention, bring about the question “What exactly do you mean by ‘slightly less complete’?” Well, REORGANIZE only works on the leaf level of your index—non-leaf levels of the index go untouched. This means that we’re not quite getting a full optimization, but, for the lion’s share of indexes, that is not where you real cost of fragmentation is (though it can happen and your mileage may vary).

Given its much lower impact on users, this is usually the tool you’ll want to use as part of your regular maintenance plan. Let’s take a look at running an index reorganization command.

To run this through its paces, we’re going to do a reorg on a table in the AdventureWorks database. The Production.TransactionHistory table is an excellent example of a table that is likely to have many rows inserted over time and then have rows purged back out of it as the transactions become old enough to delete. In this case, we’ll reorganize all the index on the table in one simple command:

```
ALTER INDEX ALL
ON Production.TransactionHistory
REORGANIZE;
```

The `ALTER INDEX` command sees that `ALL` was supplied instead of a specific index name, and looks up what indexes are available for our `Production.TransactionHistory` table (leaving out any that are disabled since a reorganization will do nothing for them). It then enumerates each index behind the scenes and performs the reorganization on each—reorganizing just the leaf level of each index (including reorganizing the actual data since the clustered index on this table will also be reorganized).

You should get back essentially nothing from the database—just a simple “Command(s) completed successfully.”

Archiving of Data

Ooh—here’s a tricky one. There are as many ways of archiving data as there are database engineers. If you’re building a OLAP database—for example, to utilize with Analysis Services—then that will often address what you need to know as far as archiving for long-term reporting goes. Regardless of how you’re making sure the data you need long-term is available, there will likely come a day when you need to deal with the issue of your data becoming simply too voluminous for your system to perform well.

As I said, there are just too many ways to go about archiving because every database is a little bit different. The key is to think about archiving needs at the time that you create your database. Realize that, as you start to delete records, you’re going to be hitting referential integrity constraints and/or orphaning records—design in a logical path to delete or move records at archive time. Here are some things to think about as you write your archive scripts:

- ❑ If you already have the data in an OLAP database, then you probably don’t need to worry about saving it anywhere else—talk to your boss and your attorney on that one.
- ❑ How often is the data really used? Is it worth keeping? Human beings are natural born pack rats in a larger size. Simply put, we hate giving things up—that includes our data. If you’re only worried about legal requirements, think about just saving a copy of never or rarely used data to tape (I’d suggest multiple backups for archive data) and reducing the amount of data you have online—your users will love you for it when they see improved performance.
- ❑ Don’t leave orphans. As you start deleting data, your referential integrity constraints should keep you from leaving that many orphans, but you’ll wind up with some where referential integrity didn’t apply. This situation can lead to serious system errors.
- ❑ Realize that your archive program will probably need a long time to run. The length of time it runs and the number of rows affected may create concurrency issues with the data your online users are trying to get at—plan on running it at a time where your system will have not be used.
- ❑ TEST! TEST! TEST!

Summary

Well, that gives you a few things to think about. It’s really easy as a developer to think about many administrative tasks and establish what the inaccurately named *Hitchhiker’s Guide to the Galaxy* trilogy called an “SEP” field. That’s something that makes things like administration seem invisible because it’s “somebody else’s problem.” Don’t go there!

A project I'm familiar with from several years ago is a wonderful example of taking responsibility for what can happen. A wonderful system was developed for a nonprofit group that operates in the north-western United States. After about eight months of operation, an emergency call was placed to the company that developed the software (it was a custom job). After some discussion, it was determined that the database had somehow become corrupted, and it was recommended to the customer that the database be restored from a backup. The response? "Backup?" The development company in question missed something very important — they knew they had an inexperienced customer that would have no administration staff, and who was going to tell the customer to do backups and help set it up if the development company didn't? I'm happy to say that the development company in question learned from that experience — and so should you.

Think about administration issues as you're doing your design and especially in your deployment plan. If you plan ahead to simplify the administration of your system, you'll find that your system is much more successful — that usually translates into rewards for the developer (i.e., you!).

25

SMO: SQL Management Objects

It's been a long road, and we're getting closer and closer to the end of our walk through SQL Server. It is, of course, no coincidence that our chat about how to manage your SQL Server programmatically has been held until very close to the end. Among other things, we needed to have a solid idea as to what objects we were managing and what administrative needs we had before we were ready to understand the SMO object model and talk about some of the reasons we might want to use SMO.

So, what exactly is SMO? Well, as the title of this chapter implies, SMO is an object model for managing SQL Server. Whereas connectivity models like ADO are all about accessing data, SMO is all about access the structure and health of your system.

In this chapter, we'll look at:

- The convoluted history of SQL Server management object models
- The basics of the SQL SMO object model
- A simple SMO example project

As with many of the SQL Server topics we cover in this book, SQL SMO can and will be a book unto itself (indeed, there are several such books already on the market), so please do not expect to come out of this chapter an expert. That said, hopefully, you will have the fundamentals down to at least the point to where you know what's possible and how much work is likely to be involved. From there, you can look for sources of more information as necessary.

The History of SQL Server Management Object Models

This is, to me—even as someone who genuinely loves the product—not an area where SQL Server shines. This is not to say that SMO is a bad thing but rather that it is a rather sordid history. The team has had a tough time picking a horse and sticking with it.

As I write this, I've been working with SQL Server for something over a decade. In that time, the methods of managing SQL Server have changed several different times. "A new release? A new management method!" could be the motto for SQL Server. In short, there are probably countries out there with shorter histories than SQL Server's management model efforts.

So, let's look at the highlights from the last couple of releases. These are some of the different models and technologies you may bump into as you work on legacy code out there.

SQL Distributed Management Objects

Distributed Management Objects, or DMO, is the relative "old dog" of the management models. When you think of the old Enterprise Manager from prior versions of SQL Server, most of its underlying functionality ended up in a DMO call. The DMO model supported COM, and could perform all the basic tasks you might want management-wise, such as:

- Start a backup
- Restore from backup
- Create a database
- Create jobs and other agent-related tasks
- Reverse engineer tables into SQL code

The list goes on.

So, what went wrong with DMO? Well, the object model was often deemed "clunky" at best. Indeed, parts of DMO often did not work well together, and the scripting engine was buggy at best. In short, most developers I know only used DMO after going through an electroshock therapy program to desensitize them to the pain of it (okay, it wasn't that bad, but not far from there).

SQL Namespaces

SQL Namespaces (SQL NS) is actually largely about providing UI level functionality. SQL NS encapsulates all of the functionality that you would find in the old Enterprise Manager—complete with the UI elements. You instantiate the UI objects, and those objects already utilizing SQL DMO underneath, and

remove that layer of programming from the equation. In short, if you needed to build a tool that already had the UI to do management tasks, then SQL NS was your tool. The problem? Well, put it this way—EM? They decided they needed to replace it. DMO? They decided they need to replace it, too. As you can guess, apparently not even Microsoft was all that impressed.

Now, lest I sound like all I am is a Microsoft basher or that I think EM was a bad product, I'll put it this way: EM was a fairly good "first shot at it." None of the RDBMS systems out there had anything remotely as powerful and useful as Enterprise Manager was when it first came out—it was a huge part of why SQL Server has been perceived as so much more usable than, say, Oracle. That usability, coupled with that used to be a very cheap price tag, is a big part of Microsoft's success with SQL Server.

EM did, however, have a number of flaws that became more and more obvious as the Windows era taught us what a Windows application should look and act like.

Windows Management Instrumentation

Windows Management Instrumentation (WMI) is very different from the other management objects we've talked about this far in the sense that it is not SQL Server specific, but, rather, an implementation of a management scripting model that was already taking hold to manage servers across Windows and beyond.

WMI is an implementation of the industry open standard Web-Based Enterprise Management (WBEM) initiative. WBEM goes well beyond Microsoft products, and the idea was that server administrators would be able to learn one core scripting model and manage all of their servers with it. Exchange, SQL Server, Windows O/S features, and more—it was all going to be managed using WMI (and, indeed, most of it can be).

Going into SQL Server 2000, the message was clear: WMI was the future. Many of the SQL Server stalwarts (like me) were told over and over again—DMO would be going away (well, that much turned out to be true), and we should do any new management in WMI (that much turned out to be not so true).

The reality is that WMI was never fully implemented for SQL Server, but it also will probably not totally go away any time soon. WMI is, as I've said, an industry standard, and many other Windows servers use WMI for configuration management. Having WMI available for the configuration fundamentals makes a lot of sense, and, for that space, it's likely here to stay (with no complaints from me).

It's worth noting that WMI is now implemented as a layer over SMO—go figure.

SMO

It's unclear to me exactly when Microsoft decided to make the move to SMO. What I can say is that they knew they had a problem: DMO was clearly at the end its useful life, and a complete rewrite of Enterprise Manager was already planned. At the same time, WMI was clearly not going to address everything that needed to be done (WMI is configuration oriented, but SQL Server needs more administrative love than WMI was likely to give in any kind of usable way).

So, as SQL Server 2000 was coming to market, .NET was already clearly on the horizon. What has become Visual Studio 2005 was already in heavy design. C# was already being sold as the programming language of the future. The decision was made to use Visual Studio plug-ins as the management center (indeed, you still see that for Reporting, Integration, and Analysis Services).

In the end, what we have in SMO as a very useful set of .NET assemblies. Management Studio has gone back to being its own thing (being too tied in to Visual Studio apparently didn't work out so well, but I like the decision to keep them separate), but it does utilize some of the Visual Studio notions. The services that require the notion of a designer use Business Intelligence Development Studio, which is still basically a set of projects and controls for Visual Studio (indeed, it says Visual Studio as you start it up).

My guess? Well, depending on how long it is before SQL Server goes to a new version again, I think it's safe to say you can count on SMO being the object model for no less than another 1–2 releases. There is no replacement on the horizon, and SMO looks very viable (no reason to replace it in the foreseeable future). In short, you should be able to count on it for at least 5–10 years, which is about as much as anyone can hope for anything in the software business.

The SMO Object Model

Server Management Objects, or SMO, replaces DMO. That said, SMO goes well beyond anything DMO was conceived to do. Beyond basic configuration or even statement execution, SMO has some truly advanced features such as:

- ❑ Event handling—SMO supports the notion of trapping events that are happening on the server and injecting code to handle the event situation.
- ❑ The ability to address types of objects in your server as collections (making it easy to enumerate them and provide consistent and complete treatment for all objects of that type).
- ❑ The ability to address all of the various server objects that are part of SQL Server in a relatively consistent manner.

Like all object models, SMO establishes something of a hierarchy among objects. Because SQL Server is such a complex product, there are many, many objects to consider. Figure 25-1 includes an example of the hierarchy of what I would consider to be "core" objects in SQL Server.

Note that this is not at all a comprehensive list! If you want a diagram with everything, check Books Online (they have one that isn't bad, though it's not great either—at least it's complete). This is my attempt at giving you something that is more readable and has all the core objects plus a few.

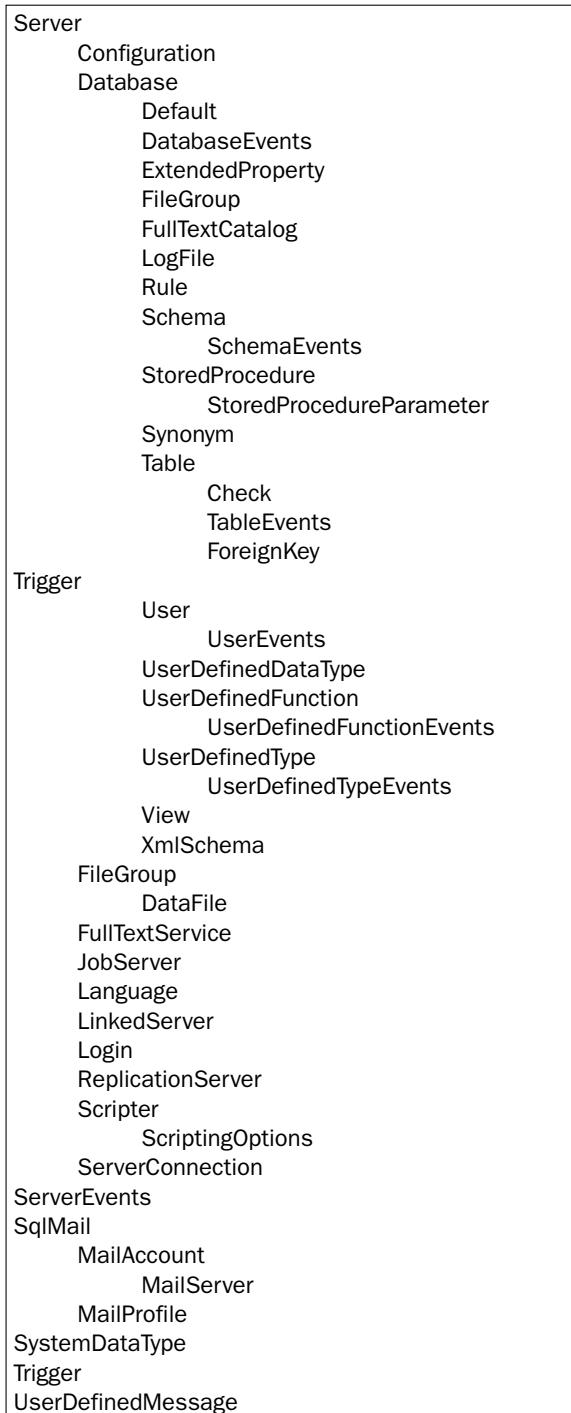


Figure 25-1

Walking through Some Examples

This may well be the messiest section in the entire book in terms of hearing me “talk” about things, as it includes a ton of Visual Studio stuff that goes well beyond what is built into the base SQL Server Business Intelligence Studio.

You must have some version of Visual Studio .NET in order to actually build these examples yourself. Not to fear, however, if you don’t—I do show all the lines of code here, so you can at least look them over.

Also, the examples below are done in C#, but the basic object references and method calls are the same—conversion to VB or C++ should be simple for those more comfortable in those languages.

What we’re going to be doing in this section is building up a little application that does a number of different “basics” that you might be interested in. Among the things that will happen at least once among all these various actions are:

- ❑ Creating a reference to a specific server, including a connection to a server using a trusted connection
- ❑ Creating an entirely new database
- ❑ Creating tables in a database
- ❑ Creating primary key constraints for those tables
- ❑ Creating a foreign key referencing from one table to another
- ❑ Dropping a database
- ❑ Backing up a database
- ❑ Scripting a database object

Each of these is a hyper-simplified version of what is required. Keep in mind that each of the objects I reference here has many more possible properties and methods to be set. For example, in the scripting example, we could play around with scripting options to change what generalize property commands do and do not appear in the script.

Getting Started

Start by creating a new Windows Application project in Visual Studio. I called mine `SQLSMOExample`. In order to make use of the SMO assemblies, you’ll need to set references in your project to at least four assemblies:

- ❑ `Microsoft.SqlServer.ConnectionInfo`
- ❑ `Microsoft.SqlServer.Smo`
- ❑ `Microsoft.SqlServer.SmoEnum`
- ❑ `Microsoft.SqlServer.SqlEnum`

Setting a reference is as easy as right-clicking “References” in the Solution Explorer and choosing “Add Reference....” Select the four assemblies above, and click OK.

For my example, all of my code is, for simplicity's sake, done in a Form called `frmMain`. In most cases, you would want to set up separate component files for your methods and just call them from a form as needed.

Declarations

We need to add declarations to a couple of the management libraries to make it simple to utilize those objects in our code.

```
using Microsoft.SqlServer.Management.Smo;  
using Microsoft.SqlServer.Management.Common;
```

This will allow us to reference several objects within these libraries without having to fully qualify them.

Basic Connection and Server References

There is a block of code you will see me reuse in every one of the methods we'll create in this chapter. The purpose of the code is to establish a connection and a server reference — everything we do will need these.

In practice, we would likely establish one or more connections that would be global to the application rather than a specific method, but, again, I am trying to keep the code blocks somewhat independent, so that you can look at them individually.

The connection and server reference code looks like this:

```
// Create the server and connect to it.  
ServerConnection cn = new ServerConnection();  
cn.LoginSecure = true;  
  
Server svr = new Server(cn);  
svr.ConnectionContext.Connect();
```

Creating a Database

Creating a database is pretty straightforward. In the implementation that follows, I create a `Database` object and immediately initialize with a reference to our `svr Server` object. Note, however, that all I am creating is a database definition object — the database itself is not actually created on the server until we call the `Create()` method of the database object. So, in short, we define the object, modify the various properties that define it, and then, and only then, do we call the `Create()` method to actually create the database on the server that is referenced in our `Server` object.

Drop a button onto the main form — I've called mine `btnCreateDB` — and you're ready to add some code. A simple method to create the database this might include:

```
private void btnCreateDB_Click(object sender, EventArgs e)  
{  
    // Create the server and connect to it.  
    ServerConnection cn = new ServerConnection();  
    cn.LoginSecure = true;  
  
    Server svr = new Server(cn);
```

```
        svr.ConnectionContext.Connect();

        Database db = new Database();

        db.Parent = svr;
        db.Name = "SMODatabase";
        db.Create();

        txtResult.Text = "Database Created";

        cn.Disconnect();

    }
```

I've established a generic database object. I then associated it with a specific server, gave the logic name for the database, and then created it.

The result is really nothing different than if we had connected to our database and issued the command:

```
CREATE DATABASE SMODatabase
```

We wind up with an empty database that is created completely with defaults. We could, however, have set things like the physical file location (including creating it with multiple file groups), default collation, growth and size properties—basically anything you normally think of as a property of the database.

Creating Tables

In this example, I'm going to add a pair of tables to our empty SMODatabase. We'll add `ParentTable` and `ChildTable`. `ChildTable` will have a foreign key to `ParentTable`. Both will have primary keys.

First, we'll need to set a reference to what database we want to create our tables in:

```
private void btnCreateTables_Click(object sender, EventArgs e)
{
    // Create the server and connect to it.
    ServerConnection cn = new ServerConnection();
    cn.LoginSecure = true;

    Server svr = new Server(cn);
    svr.ConnectionContext.Connect();

    // Get a reference to our test SMO Database
    Database db = svr.Databases["SMODatabase"];
```

Notice that this time I did not create the `Database` object as “new”—instead, I associated it with an existing database object from our referenced `Server` object.

From there, I create a new table object. Much as when we created the `Database` object, all we are doing is creating a object definition in your application—no table will be created in the database until after we've fully defined our `Table` object and called its `Create()` method.

```
// Create Table object, and begin defining said table
Table ParentTable = new Table(db, "ParentTable");
```

Now we're ready to start adding some meat to the definition of our table. Unlike a database, which has enough defaults that you really only need to specify a name to create one (the rest it will just be copied from the model database), tables require a lot of specification—specifically, it needs at least one column.

Let's add a column that will eventually serve as our primary key:

```
// Build up the table definition
Column ParentKey = new Column(ParentTable, "ParentKey");
ParentKey.DataType = DataType.Int;
ParentKey.Nullable = false;
ParentKey.Identity = true;
```

We've created a new column object. It has been templated from the `ParentTable` and named `ParentKey`. I've given it a data type of `int`, made it non-nullable, and defined it as an `IDENTITY` column.

Even though we've templated the column from the `ParentTable`, it is not yet associated directly with that table! The templating reference just helps establish what the initial property values are for the column (such as collation).

Now let's add another column called `ParentDescription`:

```
Column ParentDescription = new Column(ParentTable,
"ParentDescription");
ParentDescription.DataType = DataType.NVarCharMax;
ParentDescription.Nullable = false;
```

Again, the column is created, but not directly associated with the `Table` object yet—let's take care of that now:

```
// Now actually add them to the table definition
ParentTable.Columns.Add(ParentKey);
ParentTable.Columns.Add(ParentDescription);
```

It is not until we add them to the `Columns` collection of the `Table` object that they become directly associated with that table.

So, now we have a table object defined, and it has two columns associated with it. What we need now is a primary key.

```
// Add a Primary Key
Index PKParentKey = new Index(ParentTable, "PKParentKey");
PKParentKey.IndexKeyType = IndexKeyType.DriPrimaryKey;

PKParentKey.IndexedColumns.Add(new IndexedColumn(PKParentKey,
"ParentKey"));

ParentTableIndexes.Add(PKParentKey);
```

Chapter 25

Notice that we're defining the primary key as an index rather than as anything explicitly called a constraint. Instead, we define the index, and then tell the index (via its `IndexKeyType`) that it is a primary key. When the index is created, the constraint definition will also be added.

Much like our columns, the primary key is not directly associated with the table until we explicitly add it to the `Indexes` collection of our table.

With all that done, we're ready to create our table:

```
ParentTable.Create();
```

It is at this point that the table is physically created in the database.

Okay, with our parent table created, we're ready to add our child table. The code up through the creation of the primary key looks pretty much just as the `ParentTable` object did:

```
// Create Table object for child, and begin defining said table
Table ChildTable = new Table(db, "ChildTable");

// Build up the Child table definition
Column ChildParentKey = new Column(ChildTable, "ParentKey");
ChildParentKey.DataType = DataType.Int;
ChildParentKey.Nullable = false;

Column ChildKey = new Column(ChildTable, "ChildKey");
ChildKey.DataType = DataType.Int;
ChildKey.Nullable = false;

Column ChildDescription = new Column(ChildTable, "ChildDescription");
ChildDescription.DataType = DataType.NVarCharMax;
ChildDescription.Nullable = false;

// Now actually add them to the table definition
ChildTable.Columns.Add(ChildParentKey);
ChildTable.Columns.Add(ChildKey);
ChildTable.Columns.Add(ChildDescription);

// Add a Primary Key that is a composite key
Index PKChildKey = new Index(ChildTable, "PKChildKey");
PKChildKey.IndexKeyType = IndexKeyType.DriPrimaryKey;

PKChildKey.IndexedColumns.Add(new IndexedColumn(PKChildKey,
"ParentKey"));
PKChildKey.IndexedColumns.Add(new IndexedColumn(PKChildKey,
"ChildKey"));

ChildTable.Indexes.Add(PKChildKey);
```

But with `ChildTable`, we want to add a twist in the form of a foreign key. To do this, we create a `ForeignKey` object:

```
// Add a Foreign Key  
ForeignKey FKParent = new ForeignKey(ChildTable, "FKParent");
```

And then create `ForeignKeyColumn` objects to add to the `ForeignKey` object.

```
// The first "Parent Key" in the definition below is the name in the  
current table  
// The second is the name (of just the column) in the referenced table.  
ForeignKeyColumn FKParentParentKey = new ForeignKeyColumn(FKParent,  
"ParentKey", "ParentKey");  
  
FKParent.Columns.Add(FKParentParentKey);
```

Next, set a reference to a specific table:

```
FKParent.ReferencedTable = "ParentTable";  
  
// I could have also set a specific schema, but since the table was  
created using just a  
// default schema, I'm leaving the table reference to it default also.  
They would be  
// created using whatever the user's default schema is  
  
/*  
** Note that there are several other properties we could define here  
** such as CASCADE actions. We're going to keep it simple for now.  
*/
```

Then actually add the foreign key to and create the table:

```
ChildTable.ForeignKeys.Add(FKParent);  
  
ChildTable.Create();  
  
cn.Disconnect();  
  
txtResult.Text = "Tables Created";  
  
}
```

I recognize that this probably seems convoluted compared to just connecting and issuing a `CREATE TABLE` statement, but there are several advantages:

- ❑ If you are dynamically building a table, you can encapsulate the various parts of the table construction more easily than trying to do string manipulation.
- ❑ Changes to the properties of the various objects involved is far less sensitive to specific order of execution than trying to build a string would be.
- ❑ All the properties remain discreet, so they are easily addressed and edited without significant string manipulation

- ❑ It is the SMO way of doing things—if the other actions you’re taking are already in SMO, then doing things consistently in SMO is probably going to yield less confusion than if you mix string-based commands with SMO commands.

Dropping a Database

As with most drop situations, this one is pretty straightforward. We start with our now familiar server and connection info and then set a reference to what database we’re interested in:

```
private void btnDropDB_Click(object sender, EventArgs e)
{
    // Create the server and connect to it.
    ServerConnection cn = new ServerConnection();
    cn.LoginSecure = true;

    Server svr = new Server(cn);
    svr.ConnectionContext.Connect();

    Database db = svr.Databases["SMODatabase"];
}
```

Then just call the `Drop()` method and we’re done:

```
db.Drop();

txtResult.Text = "Database Dropped";

cn.Disconnect();

}
```

Note that we do not have any error trapping added here (there really isn’t anything different than other error-trapping issues in your language of choice). You may run into some issues dropping the database if you still have connections open to that database elsewhere in this or other applications (such as Management Studio).

Backing Up a Database

For this one, we’re actually going to switch over and use the AdventureWorks database just to give us something to make a meatier backup of.

As you might suspect from how many different objects we’ve seen so far, the `Backup` object is its own thing. It is considered a child of the `Server` object but has its own set of properties and methods.

To create a backup, you do the same server connection code that we’ve seen several times now:

```
private void btnBackupDB_Click(object sender, EventArgs e)
{
    // Create the server and connect to it.
    ServerConnection cn = new ServerConnection();
    cn.LoginSecure = true;

    Server svr = new Server(cn);
    svr.ConnectionContext.Connect();
```

We're then ready to create a new `Backup` object. Note that, unlike the `Database` object, which we associated with a server early on, we don't need to reference a specific server for our `Backup` object until we actually go to execute the backup.

```
// Create and define backup object
Backup bkp = new Backup();
bkp.Action = BackupActionType.Database;
bkp.Database = "AdventureWorks";
bkp.Devices.AddDevice(@"c:\SMOSMOSample.bak", DeviceType.File);
```

I've created the `Backup` object and told it what kind of a backup it should expect to do (A Database backup as opposed to, say, a Log backup). I've also set what database it's going to be backing up and defined a device for it to use.

Note that, while here I defined a file device and path on the fly, you could just as easily connect to the server and query what devices are already defined on the server and then select one of those for your backup. Similarly, the device could be of a different type—such as a tape.

Now we're ready to execute the backup. We have two different methods available for this:

- `SqlBackup`—This is a synchronous backup—your code will not gain control again until the backup is either complete or errors out.
- `SqlBackupAsync`—This tells the server to start the backup and then returns control to your application as soon as the server accepts the backup request as being valid (the backup will then run in the background). It's important to note that you do have the ability to receive notifications as the backup reaches completion points (you can define the granularity of those completion points).

I've chosen the asynchronous backup method in my example.

```
// Actually start the backup. Note that I've said to do this
// Asynchronously
// I could easily have made it synchronous by choosing SqlBackup
instead
bkp.SqlBackupAsync(svr);

cn.Disconnect();

}
```

After you've run this, go take a look for the `SQLSMOSample.bak` file in the root of your C: drive and it should be there!

Scripting

Perhaps one of the most compelling abilities that SMO offers the true developer crown is the ability to script out objects that are already in the database. Indeed, SMO can script out backups, reverse engineer tables, and even record the statements being sent to the server.

For our example, we're going to reverse engineer a script for the `HumanResources.Employee` table in the `AdventureWorks` database. We'll see just how easily even a relatively complex table definition can be scripted out for other use.

We start with the same server, connection, and database reference code we've used several times in this chapter:

```
private void btnScript_Click(object sender, EventArgs e)
{
    // Create the server and connect to it.
    ServerConnection cn = new ServerConnection();
    cn.LoginSecure = true;

    Server svr = new Server(cn);
    svr.ConnectionContext.Connect();

    // Now define the database we want to reference the table from.
    Database db = svr.Databases["AdventureWorks"];
```

Next, we set a reference to the table that we want to script out—we could just as easily be scripting out a different type of SQL Server object such as a stored procedure, a view, or even a database. Indeed, it can even be a server-level object such as a device or login.

```
// Get a reference to the table. Notice that schema is actually the
*2nd* parameter
// not the first.
Table Employee = db.Tables["Employee", "HumanResources"];
```

We're then ready to call the `Script()` method. The only real trick here is to realize that it returns not just a single string but rather a collection of strings. In order to receive this, we'll need to set up a variable of the proper `StringCollection` type, which is not defined in any of our using declarations—we will, therefore, need to fully qualify that variable declaration.

```
string          // Call the Script method. The issue with this is that it returns a
               // *collection* rather than a string. We'll enumerate it into a string
shortly.        System.Collections.Specialized.StringCollection script =
Employee.Script();
```

Okay, so we've received our script, but now we want to take a look. I'll define a holding variable and copy all of the separate strings into just one string to use in a `MessageBox`:

```

        string MyScript = "";

        foreach (string s in script)
        {
            MyScript = MyScript + s;
        }

        // Now show what we got out of it - very cool stuff.
        MessageBox.Show(MyScript);

        cn.Disconnect();
    }
}

```

Execute this, and you get a very usable script returned, as shown in Figure 25-2.

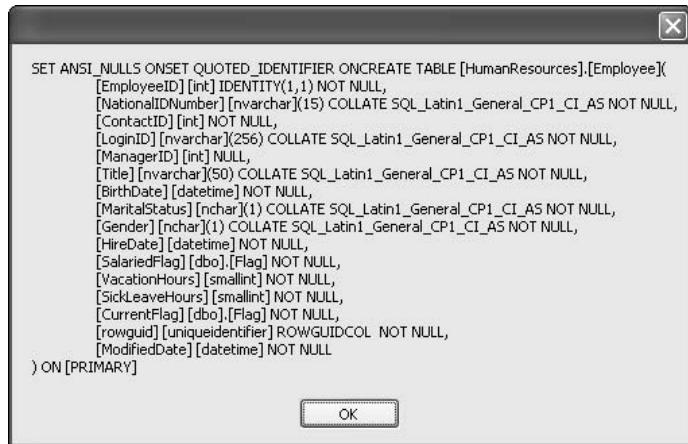


Figure 25-2

Pulling It All Together

Okay, we looked at the code in fragments, so I wanted to provide something of a reference section to show what all my code looked like pulled together:

How you choose to do your form is up to you, but mine looks like Figure 25-3.

Which buttons are which in the code should be self-descriptive based on the button names you'll see in the code. The very bottom box is a text box that I called `txtReturn` in the code.

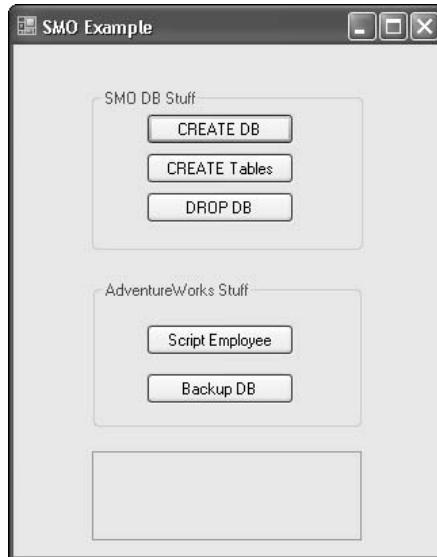


Figure 25-3

Following is my entire form code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using Microsoft.SqlServer.Management.Smo;
using Microsoft.SqlServer.Management.Common;

namespace SQLSMOSample
{
    public partial class frmMain : Form
    {
        public frmMain()
        {
            InitializeComponent();
        }

        private void btnBackupDB_Click(object sender, EventArgs e)
        {
            // Create the server and connect to it.
            ServerConnection cn = new ServerConnection();
            cn.LoginSecure = true;

            Server svr = new Server(cn);
            svr.ConnectionContext.Connect();

            // Create and define backup object
        }
    }
}
```

```
Backup bkp = new Backup();
bkp.Action = BackupActionType.Database;
bkp.Database = "AdventureWorks";
bkp.Devices.AddDevice(@"c:\SMOSample.bak", DeviceType.File);

    // Actually start the backup. Note that I've said to do this
Asynchronously
    // I could easily have made it synchronous by choosing SqlBackup
instead
    bkp.SqlBackupAsync(svr);
    cn.Disconnect();

}

private void btnCreateDB_Click(object sender, EventArgs e)
{
    // Create the server and connect to it.
    ServerConnection cn = new ServerConnection();
    cn.LoginSecure = true;

    Server svr = new Server(cn);
    svr.ConnectionContext.Connect();

    Database db = new Database();

    db.Parent = svr;
    db.Name = "SMODatabase";
    db.Create();

    txtResult.Text = "Database Created";

    cn.Disconnect();
}

private void btnDropDB_Click(object sender, EventArgs e)
{
    // Create the server and connect to it.
    ServerConnection cn = new ServerConnection();
    cn.LoginSecure = true;

    Server svr = new Server(cn);
    svr.ConnectionContext.Connect();

    Database db = svr.Databases["SMODatabase"];

    db.Drop();

    txtResult.Text = "Database Dropped";

    cn.Disconnect();
}

private void btnCreateTables_Click(object sender, EventArgs e)
{
```

Chapter 25

```
// Create the server and connect to it.  
ServerConnection cn = new ServerConnection();  
cn.LoginSecure = true;  
  
Server svr = new Server(cn);  
svr.ConnectionContext.Connect();  
  
// Get a reference to our test SMO Database  
Database db = svr.Databases["SMODatabase"];  
  
// Create Table object, and begin defining said table  
Table ParentTable = new Table(db, "ParentTable");  
  
// Build up the table definition  
Column ParentKey = new Column(ParentTable, "ParentKey");  
ParentKey.DataType = DataType.Int;  
ParentKey.Nullable = false;  
ParentKey.Identity = true;  
  
Column ParentDescription = new Column(ParentTable,  
"ParentDescription");  
ParentDescription.DataType = DataType.NVarCharMax;  
ParentDescription.Nullable = false;  
  
// Now actually add them to the table definition  
ParentTable.Columns.Add(ParentKey);  
ParentTable.Columns.Add(ParentDescription);  
  
// Add a Primary Key  
Index PKParentKey = new Index(ParentTable, "PKParentKey");  
PKParentKey.IndexKeyType = IndexKeyType.DriPrimaryKey;  
  
PKParentKey.IndexedColumns.Add(new IndexedColumn(PKParentKey,  
"ParentKey"));  
  
ParentTable.Indexes.Add(PKParentKey);  
  
ParentTable.Create();  
  
// Create Table object for child, and begin defining said table  
Table ChildTable = new Table(db, "ChildTable");  
  
// Build up the Child table definition  
Column ChildParentKey = new Column(ChildTable, "ParentKey");  
ChildParentKey.DataType = DataType.Int;  
ChildParentKey.Nullable = false;  
  
Column ChildKey = new Column(ChildTable, "ChildKey");  
ChildKey.DataType = DataType.Int;  
ChildKey.Nullable = false;  
  
Column ChildDescription = new Column(ChildTable, "ChildDescription");  
ChildDescription.DataType = DataType.NVarCharMax;
```

```
        ChildDescription.Nullable = false;

        // Now actually add them to the table definition
        ChildTable.Columns.Add(ChildParentKey);
        ChildTable.Columns.Add(ChildKey);
        ChildTable.Columns.Add(ChildDescription);

        // Add a Primary Key that is a composite key
        Index PKChildKey = new Index(ChildTable, "PKChildKey");
        PKChildKey.IndexKeyType = IndexKeyType.DriPrimaryKey;

        PKChildKey.IndexedColumns.Add(new IndexedColumn(PKChildKey,
    "ParentKey"));
        PKChildKey.IndexedColumns.Add(new IndexedColumn(PKChildKey,
    "ChildKey"));

        ChildTable.Indexes.Add(PKChildKey);

        // Add a Foreign Key
        ForeignKey FKParent = new ForeignKey(ChildTable, "FKParent");

        // The first "Parent Key" in the definition below is the name in the
        current table
        // The second is the name (of just the column) in the referenced table.
        ForeignKeyColumn FKParentParentKey = new ForeignKeyColumn(FKParent,
    "ParentKey", "ParentKey");

        FKParent.Columns.Add(FKParentParentKey);

        FKParent.ReferencedTable = "ParentTable";

        // I could have also set a specific schema, but since the table was
        created using just a
        // default schema, I'm leaving the table reference to it default also.
        They would be
        // created using whatever the user's default schema is

        /*
        ** Note that there are several other properties we could define here
        ** such as CASCADE actions. We're going to keep it simple for now.
        */

        ChildTable.ForeignKeys.Add(FKParent);

        ChildTable.Create();

        cn.Disconnect();

        txtResult.Text = "Tables Created";

    }

private void btnScript_Click(object sender, EventArgs e)
{
```

```
// Create the server and connect to it.  
ServerConnection cn = new ServerConnection();  
cn.LoginSecure = true;  
  
Server svr = new Server(cn);  
svr.ConnectionContext.Connect();  
  
// Now define the database we want to reference the table from.  
Database db = svr.Databases["AdventureWorks"];  
  
// Get a reference to the table. Notice that schema is actually the  
*2nd* parameter  
// not the first.  
Table Employee = db.Tables["Employee", "HumanResources"];  
  
// Call the Script method. The issue with this is that it returns a  
string  
// *collection* rather than a string. We'll enumerate it into a string  
shortly.  
System.Collections.Specialized.StringCollection script =  
Employee.Script();  
string MyScript = "";  
  
foreach (string s in script)  
{  
    MyScript = MyScript + s;  
}  
  
// Now show what we got out of it - very cool stuff.  
MessageBox.Show(MyScript);  
  
cn.Disconnect();  
  
}  
}  
}
```

Summary

Well, all I can say is “Wow!” Okay, so, in a way, this is nothing all that new — after all, DMO used to do a lot of this stuff (indeed, most everything we’ve looked at with actual code). SMO has, however, made things simpler. The “Wow!” is about thinking of the possibilities.

- ❑ Imagine issuing commands asynchronously.
- ❑ Imagine still being able to monitor the progress of those commands by receiving events as progress continues.
- ❑ Imagine being able to generate script code to support most anything you might want to do.
- ❑ Imagine being able to register event handlers on your SQL Server and being notified when custom events occur on the server.

The list goes on and on.

Most of the concepts in this chapter are nothing new. We've already looked at ways to create tables, as well as create, back up, and drop databases. The power, then, is in how discretely you can manage those tasks using SMO. We have the prospect for very robust event and error handling. We can far more easily receive configuration information about objects already in the server in a form that yields separate properties as opposed to trying to parse those values out of system stored procedures.

This chapter truly just scratches the surface of what you can do. If I've piqued your interest at all, I encourage you to consider the use of SMO in your design work, and, of course, go get a book specific to SMO if you need one (you probably will!).

A

System Functions

SQL Server includes a number of “System Functions” as well as more typical functions with the product. Some of these are used often and are fairly clear right from the beginning in terms of how to use them. Others, though, are both rarer in use and more cryptic in nature.

In this appendix, we’ll try to clarify the use of most of these functions in a short, concise manner.

Just as an FYI, in prior releases, many system functions were often referred to as “Global Variables.” This was a misnomer, and Microsoft has striven to fix it over the last few releases—changing the documentation to refer to them by the more proper “system function” name. Just keep the old terminology in mind in case any old fogies (such as myself) find themselves referring to them as Globals.

The T-SQL functions available in SQL Server 2005 fall into 11 categories:

- Legacy “system” functions
- Aggregate functions
- Cursor functions
- Date and time functions
- Mathematical functions
- Metadata functions
- Rowset functions
- Security functions
- String functions
- System functions
- Text and image functions

Legacy System Functions (a.k.a. Global Variables)

@@CONNECTIONS

Returns the number of connections attempted since the last time your SQL Server was started.

This one is the total of all connection *attempts* made since the last time your SQL Server was started. The key thing to remember here is that we are talking about attempts, not actual connections, and that we are talking about connections as opposed to users.

Every attempt made to create a connection increments this counter regardless of whether that connection was successful or not. The only catch with this is that the connection attempt has to have made it as far as the server. If the connection failed because of NetLib differences or some other network issue, then your SQL Server wouldn't even know that it needed to increase the count—it only counts if the server saw the connection attempt. Whether the attempt succeeded or failed does not matter.

It's also important to understand that we're talking about connections instead of login attempts. Depending on your application, you may create several connections to your server, but you'll probably only ask the user for information once. Indeed, even Query Analyzer does this. When you click for a new window, it automatically creates another connection based on the same login information.

This, like a number of other system functions, is better served by a system stored procedure, sp_monitor.

This procedure, in one command, produces the information from the number of connections, CPU busy, through to the total number of writes by SQL Server.

@@CPU_BUSY

Returns the time in milliseconds that the CPU has been actively doing work since SQL Server was last started. This number is based on the resolution of the system timer—which can vary—and can therefore vary in accuracy.

This is another of the “since the server started” kind of functions. This means that you can’t always count on the number going up as your application runs. It’s possible, based on this number, to figure out a CPU percentage that your SQL Server is taking up. Realistically though, I’d rather tap right into the Performance Monitor for that if I had some dire need for it. The bottom line is that this is one of those really cool things from a “gee, isn’t it swell to know that” point of view, but doesn’t have all that many practical uses in most applications.

@@CURSOR_ROWS

How many rows are currently in the last cursor set opened on the current connection. Note that this is for cursors, not temporary tables.

Keep in mind that this number is reset every time you open a new cursor. If you need to open more than one cursor at a time, and you need to know the number of rows in the first cursor, then you’ll need to move this value into a holding variable before opening subsequent cursors.

It's possible to use this to set up a counter to control your `WHILE` loop when dealing with cursors, but I strongly recommend against this practice—the value contained in `@@CURSOR_ROWS` can change depending on the cursor type and whether SQL Server is populating the cursor asynchronously or not. Using `@@FETCH_STATUS` is going to be far more reliable and at least as easy to use.

If the value returned is a negative number larger than `-1`, then you must be working with an asynchronous cursor, and the negative number is the number of records so far created in the cursor. If however, the value is `-1`, then the cursor is a dynamic cursor, in that the number of rows is constantly changing. A returned value of `0` informs you that either no cursor opened has been opened or the last cursor opened is no longer open. Finally, any positive number indicates the number of rows within the cursor.

To create an asynchronous cursor, set `sp_configure cursor threshold` to a value greater than `0`. Then, when the cursor exceeds this setting, the cursor is returned, while the remaining records are placed in to the cursor asynchronously.

@@DATEFIRST

Returns the numeric value that corresponds to the day of the week that the system considers the first day of the week.

The default in the United States is `7`, which equates to Sunday. The values convert as follows:

- `1` — Monday (the first day for most of the world)
- `2` — Tuesday
- `3` — Wednesday
- `4` — Thursday
- `5` — Friday
- `6` — Saturday
- `7` — Sunday

This can be really handy when dealing with localization issues, so you can properly layout any calendar or other day-of-week-dependent information you have.

Use the `SET DATEFIRST` function to alter this setting.

@@DBTS

Returns the last used timestamp for the current database.

At first look, this one seems to act an awful lot like `@@IDENTITY` in that it gives you the chance to get back the last value set by the system (this time, it's the last timestamp instead of the last identity value). The things to watch out for on this one include:

- The value changes based on any change in the database, not just the table you're working on.
- Any timestamp change in the database is reflected, not just those for the current connection.

Appendix A

Because you can't count on this value truly being the last one that you used (someone else may have done something that would change it), I personally find very little practical use for this one.

@@ERROR

Returns the error code for the last T-SQL statement that ran on the current connection. If there is no error, then the value will be zero.

If you're going to be writing stored procedures or triggers, this is a bread-and-butter kind of system function—you pretty much can't live without it.

The thing to remember with @@ERROR is that its lifespan is just one statement. This means that, if you want to use it to check for an error after a given statement, then you either need to make your test the very next statement, or you need to move it into a holding variable.

A listing of all the system errors can be viewed by using the sysmessages system table in the master database.

To create your own custom errors, use sp_addmessage.

@@FETCH_STATUS

Returns an indicator of the status of the last cursor FETCH operation.

If you're using cursors, you're going to be using @@FETCH_STATUS. This one is how you know the success or failure of your attempt to navigate to a record in your cursor. It will return a constant depending on whether SQL Server succeeded in your last FETCH operation or not, and, if the FETCH failed, why. The constants are:

- 0 — Success
- 1 — Failed. Usually because you are beyond either the beginning or end of the cursorset.
- 2 — Failed. The row you were fetching wasn't found, usually because it was deleted between the time when the cursorset was created and when you navigated to the current row. Should only occur in scrollable, non-dynamic cursors.

For purposes of readability, I often will set up some constants prior to using @@FETCH_STATUS.

For example:

```
DECLARE @NOTFOUND int  
DECLARE @BEGINEND int  
  
SELECT @NOTFOUND = -2  
SELECT @BEGINEND = -1
```

I can then use these in my conditional in the WHILE statement of my cursor loop instead of just the row integer. This can make the code quite a bit more readable.

@@IDENTITY

Returns the last identity value created by the current connection.

If you're using identity columns and then referencing them as a foreign key in another table, you'll find yourself using this one all the time. You can create the parent record (usually the one with the identity you need to retrieve), then select `@@IDENTITY` to know what value you need to relate child records to.

If you perform inserts into multiple tables with identity values, remember that the value in `@@IDENTITY` will only be for the *last* identity value inserted — anything before that will have been lost, unless you move the value into a holding variable after each insert. Also, if the last column you inserted into didn't have an identity column, then `@@IDENTITY` will be set to `NULL`.

@@IDLE

Returns the time in milliseconds (based on the resolution of the system timer) that SQL Server has been idle since it was last started.

You can think of this one as being something of the inverse of `@@CPU_BUSY`. Essentially, it tells you how much time your SQL Server has spent doing nothing. If anyone finds a programmatic use for this one, send me an e-mail—I'd love to hear about it (I can't think of one).

@@IO_BUSY

Returns the time in milliseconds (based on the resolution of the system timer) that SQL Server has spent doing input and output operations since it was last started. This value is reset every time SQL Server is started.

This one doesn't really have any rocket science to it, and it is another one of those that I find falls into the "no real programmatic use" category.

@@LANGID and @@LANGUAGE

Respectively return the ID and the name of the language currently in use.

These can be handy for figuring out if your product has been installed in a localization situation or not, and if so what language is the default.

For a full listing of the languages currently supported by SQL Server, use the system stored procedure, `sp_helplanguage`.

@@LOCK_TIMEOUT

Returns the current amount of time in milliseconds before the system will time out waiting for a blocked resource.

If a resource (a page, a row, a table, whatever) is blocked, your process will stop and wait for the block to clear. This determines just how long your process will wait before the statement is canceled.

Appendix A

The default time to wait is 0 (which equates to indefinitely) unless someone has changed it at the system level (using `sp_configure`). Regardless of how the system default is set, you will get a value of -1 from this global unless you have manually set the value for the current connection using `SET LOCK_TIMEOUT`.

`@@MAX_CONNECTIONS`

Returns the maximum number of simultaneous user connections allowed on your SQL Server.

Don't mistake this one to mean the same thing as you would see under the Maximum Connections property in the Management Console. This one is based on licensing and will show a very high number if you have selected "per seat" licensing.

Note that the actual number of user connections allowed also depends on the version of SQL Server you are using and the limits of your application(s) and hardware.

`@@MAX_PRECISION`

Returns the level of precision currently set for decimal and numeric data types.

The default is 38 places, but the value can be changed by using the `/p` option when you start your SQL Server. The `/p` can be added by starting SQL Server from a command line or by adding it to the Startup parameters for the MSSQLServer service in the Windows 2000, 2003, XP Services applet.

`@@NETLEVEL`

Returns the current nesting level for nested stored procedures.

The first stored procedure (sproc) to run has an `#TLEVEL` of 0. If that sproc calls another, then the second sproc is said to be nested in the first sproc (and `#TLEVEL` is incremented to a value of 1). Likewise, the second sproc may call a third, and so on up to maximum of 32 levels deep. If you go past the level of 32 levels deep, not only will the transaction be terminated, but you should revisit the design of your application.

`@@OPTIONS`

Returns information about options that have been applied using the `SET` command.

Since you only get one value back, but can have many options set, SQL Server uses binary flags to indicate what values are set. In order to test whether the option you are interested in is set, you must use the option value together with a bitwise operator. For example:

```
IF (@@OPTIONS & 2)
```

If this evaluates to `True`, then you would know that `IMPLICIT_TRANSACTIONS` had been turned on for the current connection. The values are:

Bit	SET Option	Description
1	DISABLE_DEF_CNST_CHK	Interim vs. deferred constraint checking.
2	IMPLICIT_TRANSACTIONS	A transaction is started implicitly when a statement is executed.
4	CURSOR_CLOSEON_COMMIT	Controls behavior of cursors after a COMMIT operation has been performed.
8	ANSI_WARNINGS	Warns of truncation and NULL in aggregates.
16	ANSI_PADDING	Controls padding of fixed-length variables.
32	ANSI_NULLS	Determines handling of nulls when using equality operators.
64	ARITHABORT	Terminates a query when an overflow or divide-by-zero error occurs during query execution.
128	ARITHIGNORE	Returns NULL when an overflow or divide-by-zero error occurs during a query.
256	QUOTED_IDENTIFIER	Differentiates between single and double quotation marks when evaluating an expression.
512	NOCOUNT	Turns off the row(s) affected message returned at the end of each statement.
1024	ANSI_NULL_DFLT_ON	Alters the session's behavior to use ANSI compatibility for nullability. Columns created with new tables or added to old tables without explicit null option settings are defined to allow nulls. Mutually exclusive with ANSI_NULL_DFLT_OFF.
2048	ANSI_NULL_DFLT_OFF	Alters the session's behavior not to use ANSI compatibility for nullability. New columns defined without explicit nullability are defined not to allow nulls. Mutually exclusive with ANSI_NULL_DFLT_ON.
4096	CONCAT_NULL_YIELDS_NULL	Returns a NULL when concatenating a NULL with a string.
8192	NUMERIC_ROUNDABORT	Generates an error when a loss of precision occurs in an expression.

@@PACK_RECEIVED and @@PACK_SENT

Respectively return the number of input packets read/written from/to the network by SQL Server since it was last started.

Primarily, these are a network troubleshooting tools.

Appendix A

@@PACKET_ERRORS

Returns the number of network packet errors that have occurred on connections to your SQL Server since the last time the SQL Server was started.

Primarily a network troubleshooting tool.

@@PROCID

Returns the stored procedure ID of the currently running procedure.

Primarily a troubleshooting tool when a process is running and using up a large amount of resources. Is used mainly as a DBA function.

@@REMSERVER

Returns the value of the server (as it appears in the login record) that called the stored procedure.

Used only in stored procedures. This one is handy when you want the sproc to behave differently depending on what remote server (often a geographic location) the sproc was called from.

@@ROWCOUNT

Returns the number of rows affected by the last statement.

One of the most used globals, my most common use for this one is to check for nonruntime errors—that is, items that are logically errors to your program but that SQL Server isn't going to see any problem with. An example is a situation where you are performing an update based on a condition, but you find that it affects zero rows. Odds are that, if your client submitted a modification for a particular row, then it was expecting that row to match the criteria given—zero rows affected is indicative of something being wrong.

However, if you test this system function on any statement that does not return rows, then you will also return a value of 0.

@@SERVERNAME

Returns the name of the local server that the script is running from.

If you have multiple instances of SQL Server installed (a good example would be a web hosting service which uses a separate SQL Server installation for each client), then @@SERVERNAME returns the following local server name information if the local server name has not been changed since setup:

Instance	Server Information
Default instance	<servername>
Named instance	<servername\instancename>

Instance	Server Information
Virtual server— default instance	<virtualservername>
Virtual server— named instance	<virtualservername\instancename>

@@SERVICENAME

Returns the name of the registry key under which SQL Server is running.

Only returns something under Windows 2000/2003/XP, and (under either of these) should always return `MSSQLService` unless you've been playing games in the registry.

@@SPID

Returns the server process ID (SPID) of the current user process.

This equates to the same process ID that you see if you run `sp_who`. What's nice is that you can tell the SPID for your current connection, which can be used by the DBA to monitor, and if necessary terminate, that task.

@@TEXTSIZE

Returns the current value of the `TEXTSIZE` option of the `SET` statement, which specifies the maximum length, in bytes, returned by a `SELECT` statement when dealing with text or image data.

The default is 4096 bytes (4KB). You can change this value by using the `SET TEXTSIZE` statement.

@@TIMETICKS

Returns the number of microseconds per tick. This varies by machines and is another of those that falls under the category of “no real programmatic use.”

@@TOTAL_ERRORS

Returns the number of disk read/write errors encountered by the SQL Server since it was last started.

Don't confuse this with runtime errors or as having any relation to `@@ERROR`. This is about problems with physical I/O. This one is another of those of the “no real programmatic use” variety. The primary use here would be more along the lines of system diagnostic scripts. Generally speaking, I would use Performance Monitor for this instead.

@@TOTAL_READ and @@TOTAL_WRITE

Respectively return the total number of disk reads/writes by SQL Server since it was last started.

The names here are a little misleading, as these do not include any reads from cache — they are only physical I/O.

Appendix A

@@TRANCOUNT

Returns the number of active transactions—essentially the transaction nesting level—for the current connection.

This is a very big one when you are doing transactioning. I'm not normally a big fan of nested transactions, but there are times where they are difficult to avoid. As such, it can be important to know just where you are in the transaction-nesting side of things (for example, you may have logic that only starts a transaction if you're not already in one).

If you're not in a transaction, then @@TRANCOUNT is 0. From there, let's look at a brief example:

```
SELECT @@TRANCOUNT As TransactionNestLevel      --This will be zero at this point

BEGIN TRAN
SELECT @@TRANCOUNT As TransactionNestLevel      --This will be one at this point
    BEGIN TRAN
        SELECT @@TRANCOUNT As TransactionNestLevel  --This will be two at this point
    COMMIT TRAN
SELECT @@TRANCOUNT As TransactionNestLevel      --This will be back to one
                                                --at this point
ROLLBACK TRAN
SELECT @@TRANCOUNT As TransactionNestLevel      --This will be back to zero
                                                --at this point
```

Note that, in this example, the @@TRANCOUNT at the end would also have reached zero if we had a COMMIT as our last statement.

@@VERSION

Returns the current version of SQL Server as well as the processor type and OS architecture.

For example:

```
SELECT @@VERSION
```

gives:

```
-----
Microsoft SQL Server 2005 - 9.00.1116 (Intel X86)
Apr 9 2005 20:56:37
Copyright (c) 1988-2004 Microsoft Corporation
Beta Edition on Windows NT 5.1 (Build 2600: Service Pack 2)
```

(1 row(s) affected)

Unfortunately, this doesn't return the information into any kind of structured field arrangement, so you have to parse it if you want to use it to test for specific information.

Consider using the xp_msver system sproc instead—it returns information in such a way that you can more easily retrieve specific information from the results.

Aggregate Functions

Aggregate functions are applied to sets of records rather than a single record. The information in the multiple records is processed in a particular manner and then is displayed in a single record answer. Aggregate functions are often used in conjunction with the GROUP BY clause.

The aggregate functions are:

- AVG
- CHECKSUM
- CHECKSUM_AGG
- COUNT
- COUNT_BIG
- GROUPING
- MAX
- MIN
- STDEV
- STDEVP
- SUM
- VAR
- VARP

In most aggregate functions, the ALL or DISTINCT keywords can be used. The ALL argument is the default and will apply the function to all the values in the expression, even if a value appears numerous times. The DISTINCT argument means that a value will only be included in the function once, even if it occurs several times.

Aggregate functions cannot be nested. The expression cannot be a subquery.

AVG

AVG returns the average of the values in expression. The syntax is as follows:

```
AVG( [ALL | DISTINCT] <expression>)
```

The expression must contain numeric values. NULL values are ignored.

COUNT

COUNT returns the number of items in expression. The data type returned is of type int. The syntax is as follows:

Appendix A

```
COUNT  
(  
    [ALL | DISTINCT] <expression> | *  
)
```

The expression cannot be of the uniqueidentifier, text, image, or ntext data types. The * argument returns the number of rows in the table; it does not eliminate duplicate or NULL values.

COUNT_BIG

COUNT_BIG returns the number of items in a group. This is very similar to the COUNT function described above, with the exception that the return value has a data type of bigint. The syntax is as follows:

```
COUNT_BIG  
(  
    [ALL | DISTINCT ] <expression> | *  
)
```

GROUPING

GROUPING adds an extra column to the output of a SELECT statement. The GROUPING function is used in conjunction with CUBE or ROLLUP to distinguish between normal NULL values and those added as a result of CUBE and ROLLUP operations. Its syntax is:

```
GROUPING (<column_name>)
```

GROUPING is only used in the SELECT list. Its argument is a column that is used in the GROUP BY clause and that is to be checked for NULL values.

MAX

The MAX function returns the maximum value from expression. The syntax is as follows:

```
MAX([ALL | DISTINCT] <expression>)
```

MAX ignores any NULL values.

MIN

The MIN function returns the smallest value from expression. The syntax is as follows:

```
MIN([ALL | DISTINCT] <expression>)
```

MIN ignores NULL values.

STDEV

The STDEV function returns the standard deviation of all values in expression. The syntax is as follows:

STDEV(<expression>)

STDEV ignores NULL values.

STDEVP

The STDEVP function returns the standard deviation for the population of all values in expression. The syntax is as follows:

STDEVP(<expression>)

STDEVP ignores NULL values.

SUM

The SUM function will return the total of all values in expression. The syntax is as follows:

SUM([ALL | DISTINCT] <expression>)

SUM ignores NULL values.

VAR

The VAR function returns the variance of all values in expression. The syntax is as follows:

VAR(<expression>)

VAR ignores NULL values.

VARP

The VARP function returns the variance for the population of all values in expression. The syntax is as follows:

VARP(<expression>)

VARP ignores NULL values.

Cursor Functions

There is only one cursor function (CURSOR_STATUS), and it provides information about cursors.

CURSOR_STATUS

The CURSOR_STATUS function allows the caller of a stored procedure to determine if that procedure has returned a cursor and result set. The syntax is as follows:

Appendix A

```
CURSOR_STATUS
(
    {'<local>', '<cursor_name>'}
    | {'<global>', '<cursor_name>'}
    | {'<variable>', '<cursor_variable>'}
)
```

local, global, and variable all specify constants that indicate the source of the cursor. Local equates to a local cursor name, global to a global cursor name, and variable to a local variable.

If you are using the `cursor_name` form then there are four possible return values:

- 1 — The cursor is open. If the cursor is dynamic, its result set has zero or more rows. If the cursor is not dynamic, it has one or more rows.
- 0 — The result set of the cursor is empty.
- 1 — The cursor is closed.
- 3 — A cursor of `cursor_name` does not exist.

If you are using the `cursor_variable` form, there are five possible return values:

- 1 — The cursor is open. If the cursor is dynamic, its result set has zero or more rows. If the cursor is not dynamic, it has one or more rows.
- 0 — The result set is empty.
- 1 — The cursor is closed.
- 2 — There is no cursor assigned to the `cursor_variable`.
- 3 — The variable with name `cursor_variable` does not exist, or if it does exist, has not had a cursor allocated to it yet.

Date and Time Functions

The date and time functions perform operations on values that have `datetime` and `smalldatetime` data types or that are character data types in a date form. They are:

- DATEADD
- DATEDIFF
- DATENAME
- DATEPART
- DAY
- GETDATE
- GETUTCDATE
- MONTH
- YEAR

SQL Server recognizes eleven “dateparts” and their abbreviations, as shown in the following table:

Datepart	Abbreviations
year	YY, YYYY
quarter	qq, q
month	mm, m
dayofyear	dy, y
day	dd, d
week	wk, ww
weekday	dw
hour	hh
minute	mi, n
second	ss, s
millisecond	ms

DATEADD

The DATEADD function adds an interval to a date and returns a new date. The syntax is as follows:

```
DATEADD(<datepart>, <number>, <date>)
```

The `datepart` argument specifies the time scale of the interval (day, week, month, etc.) and may be any of the dateparts recognized by SQL Server. The `number` argument is the number of dateparts that should be added to the `date`.

DATEDIFF

The DATEDIFF function returns the difference between two specified dates in a specified unit of time (for example: hours, days, weeks). The syntax is as follows:

```
DATEDIFF(<datepart>, <startdate>, <enddate>)
```

The `datepart` argument may be any of the dateparts recognized by SQL Server and specifies the unit of time to be used.

DATENAME

The DATENAME function returns a string representing the name of the specified `datepart` (for example: 1999, Thursday, July) of the specified `date`. The syntax is as follows:

```
DATENAME(<datepart>, <date>)
```

Appendix A

DATEPART

The DATEPART function returns an integer that represents the specified datepart of the specified date. The syntax is as follows:

```
DATEPART(<datepart>, <date>)
```

The DAY function is equivalent to DATEPART(dd, <date>); MONTH is equivalent to DATEPART(mm, <date>); YEAR is equivalent to DATEPART(yy, <date>).

DAY

The DAY function returns an integer representing the day part of the specified date. The syntax is as follows:

```
DAY(<date>)
```

The DAY function is equivalent to DATEPART(dd, <date>).

GETDATE

The GETDATE function returns the current system date and time. The syntax is as follows:

```
GETDATE()
```

GETUTCDATE

The GETUTCDATE function returns the current UTC (Universal Time Coordinate) time. In other words, this returns Greenwich Mean Time. The value is derived by taking the local time from the server, and the local time zone, and calculating GMT from this. Daylight saving is included. GETUTCDATE cannot be called from a user-defined function. The syntax is as follows:

```
GETUTCDATE()
```

MONTH

The MONTH function returns an integer that represents the month part of the specified date. The syntax is as follows:

```
MONTH(<date>)
```

The MONTH function is equivalent to DATEPART(mm, <date>).

YEAR

The YEAR function returns an integer that represents the year part of the specified date. The syntax is as follows:

YEAR(<date>)

The YEAR function is equivalent to DATEPART(yy, <date>).

Mathematical Functions

The mathematical functions perform calculations. They are:

- ABS
- ACOS
- ASIN
- ATAN
- ATN2
- CEILING
- COS
- COT
- DEGREES
- EXP
- FLOOR
- LOG
- LOG10
- PI
- POWER
- RADIANS
- RAND
- ROUND
- SIGN
- SIN
- SQRT
- SQUARE
- TAN

ABS

The ABS function returns the positive, absolute value of numeric expression. The syntax is as follows:

ABS(<numeric expression>)

Appendix A

ACOS

The ACOS function returns the angle in radians for which the cosine is the expression (in other words, it returns the arccosine of expression). The syntax is as follows:

ACOS (<expression>)

The value of expression must be between -1 and 1 and be of the float data type.

ASIN

The ASIN function returns the angle in radians for which the sine is the expression (in other words, it returns the arcsine of expression). The syntax is as follows:

ASIN (<expression>)

The value of expression must be between -1 and 1 and be of the float data type.

ATAN

The ATAN function returns the angle in radians for which the tangent is expression (in other words, it returns the arctangent of expression). The syntax is as follows:

ATAN (<expression>)

The expression must be of the float data type.

ATN2

The ATN2 function returns the angle in radians for which the tangent is between the two expressions provided (in other words, it returns the arctangent of the two expressions). The syntax is as follows:

ATN2 (<expression1>, <expression2>)

Both expression1 and expression2 must be of the float data type.

CEILING

The CEILING function returns the smallest integer that is equal to or greater than the specified expression. The syntax is as follows:

CEILING (<expression>)

COS

The COS function returns the cosine of the angle specified in expression. The syntax is as follows:

`COS (<expression>)`

The angle given should be in radians and expression must be of the `float` data type.

COT

The `COT` function returns the cotangent of the angle specified in expression. The syntax is as follows:

`COT (<expression>)`

The angle given should be in radians and expression must be of the `float` data type.

DEGREES

The `DEGREES` function takes an angle given in radians (`expression`) and returns the angle in degrees. The syntax is as follows:

`DEGREES (<expression>)`

EXP

The `EXP` function returns the exponential value of the value given in expression. The syntax is as follows:

`EXP (<expression>)`

The expression must be of the `float` data type.

FLOOR

The `FLOOR` function returns the largest integer that is equal to or less than the value specified in expression. The syntax is as follows:

`FLOOR (<expression>)`

LOG

The `LOG` function returns the natural logarithm of the value specified in expression. The syntax is as follows:

`LOG (<expression>)`

The expression must be of the `float` data type.

LOG10

The `LOG10` function returns the base10 logarithm of the value specified in expression. The syntax is as follows:

Appendix A

`LOG10(<expression>)`

The expression must be of the float data type.

PI

The PI function returns the value of the constant. The syntax is as follows:

`PI()`

POWER

The POWER function raises the value of the specified expression to the specified power. The syntax is as follows:

`POWER(<expression>, <power>)`

RADIANS

The RADIANS function returns an angle in radians corresponding to the angle in degrees specified in expression. The syntax is as follows:

`RADIANS(<expression>)`

RAND

The RAND function returns a random value between 0 and 1. The syntax is as follows:

`RAND([<seed>])`

The seed value is an integer expression, which specifies the start value.

ROUND

The ROUND function takes a number specified in expression and rounds it to the specified length:

`ROUND(<expression>, <length> [, <function>])`

The length parameter specifies the precision to which expression should be rounded. The length parameter should be of the tinyint, smallint, or int data type. The optional function parameter can be used to specify whether the number should be rounded or truncated. If a function value is omitted or is equal to 0 (the default), the value in expression will be rounded. If any value other than 0 is provided, the value in expression will be truncated.

SIGN

The SIGN function returns the sign of the expression. The possible return values are +1 for a positive number, 0 for zero, and -1 for a negative number. The syntax is as follows:

```
SIGN(<expression>)
```

SIN

The SIN function returns the sine of an angle. The syntax is as follows:

```
SIN(<angle>)
```

The angle should be in radians and must be of the float data type. The return value will also be of the float data type.

SQRT

The SQRT function returns the square root of the value given in expression. The syntax is as follows:

```
SQRT(<expression>)
```

The expression must be of the float data type.

SQUARE

The SQUARE function returns the square of the value given in expression. The syntax is as follows:

```
SQUARE(<expression>)
```

The expression must be of the float data type.

TAN

The TAN function returns the tangent of the value specified in expression. The syntax is as follows:

```
TAN(<expression>)
```

The expression parameter specifies the number of radians and must be of the float or real data type.

Metadata Functions

The metadata functions provide information about the database and database objects. They are:

- COL_LENGTH
- COL_NAME
- COLUMNPROPERTY
- DATABASEPROPERTY
- DATABASEPROPERTYEX
- DB_ID

Appendix A

- DB_NAME
- FILE_ID
- FILE_NAME
- FILEGROUP_ID
- FILEGROUP_NAME
- FILEGROUPPROPERTY
- FILEPROPERTY
- FULLTEXTCATALOGPROPERTY
- FULLTEXTSERVICEPROPERTY
- INDEX_COL
- INDEXKEY_PROPERTY
- INDEXPROPERTY
- OBJECT_ID
- OBJECT_NAME
- OBJECTPROPERTY
- OBJECTPROPERTYEX
- SCHEMA_ID
- SCHEMA_NAME
- SQL_VARIANT_PROPERTY
- TYPE_ID
- TYPE_NAME
- TYPEPROPERTY

COL_LENGTH

The **COL_LENGTH** function returns the defined length of a column. The syntax is as follows:

```
COL_LENGTH('<table>', '<column>')
```

The **column** parameter specifies the name of the column for which the length is to be determined. The **table** parameter specifies the name of the table that contains that column.

COL_NAME

The **COL_NAME** function takes a table ID number and a column ID number and returns the name of the database column. The syntax is as follows:

```
COL_NAME(<table_id>, <column_id>)
```

The `column_id` parameter specifies the ID number of the column. The `table_id` parameter specifies the ID number of the table that contains that column.

COLUMNPROPERTY

The `COLUMNPROPERTY` function returns data about a column or procedure parameter. The syntax is as follows:

```
COLUMNPROPERTY(<id>, <column>, <property>)
```

The `id` parameter specifies the ID of the table/procedure. The `column` parameter specifies the name of the column/parameter. The `property` parameter specifies the data that should be returned for the column or procedure parameter. The `property` parameter can be one of the following values:

- `AllowsNull`—Allows NULL values.
- `IsComputed`—The column is a computed column.
- `IsCursorType`—The procedure is of type CURSOR.
- `IsFullTextIndexed`—The column has been full-text indexed.
- `IsIdentity`—The column is an IDENTITY column.
- `IsIdNotForRepl`—The column checks for IDENTITY NOT FOR REPLICATION.
- `IsOutParam`—The procedure parameter is an output parameter.
- `IsRowGuidCol`—The column is a ROWGUIDCOL column.
- `Precision`—The precision for the data type of the column or parameter.
- `Scale`—The scale for the data type of the column or parameter.
- `UseAnsiTrim`—The ANSI padding setting was ON when the table was created.

The return value from this function will be 1 for `True`, 0 for `False`, and `NULL` if the input was not valid—except for `Precision` (where the precision for the data type will be returned) and `Scale` (where the scale will be returned).

DATABASEPROPERTY

The `DATABASEPROPERTY` function returns the setting for the specified database and property name. The syntax is as follows:

```
DATABASEPROPERTY('<database>', '<property>')
```

The `database` parameter specifies the name of the database for which data on the named property will be returned. The `property` parameter contains the name of a database property and can be one of the following values:

- `IsAnsiNullDefault`—The database follows the ANSI-92 standard for NULL values.
- `IsAnsiNullsEnabled`—All comparisons made with a NULL cannot be evaluated.

Appendix A

- ❑ `IsAnsiWarningsEnabled`—Warning messages are issued when standard error conditions occur.
- ❑ `IsAutoClose`—The database frees resources after the last user has exited.
- ❑ `IsAutoShrink`—Database files can be shrunk automatically and periodically.
- ❑ `IsAutoUpdateStatistics`—The autoupdate statistics option has been enabled.
- ❑ `IsBulkCopy`—The database allows nonlogged operations (such as those performed with the bulk copy program).
- ❑ `IsCloseCursorsOnCommitEnabled`—Any cursors that are open when a transaction is committed will be closed.
- ❑ `IsDboOnly`—The database is only accessible to the `dbo`.
- ❑ `IsDetached`—The database was detached by a detach operation.
- ❑ `IsEmergencyMode`—The database is in emergency mode.
- ❑ `IsFulltextEnabled`—The database has been full-text enabled.
- ❑ `IsInLoad`—The database is loading.
- ❑ `IsInRecovery`—The database is recovering.
- ❑ `IsInStandby`—The database is read-only and restore log is allowed.
- ❑ `IsLocalCursorsDefault`—Cursor declarations default to `LOCAL`.
- ❑ `IsNotRecovered`—The database failed to recover.
- ❑ `IsNullConcat`—Concatenating to a `NULL` results in a `NULL`.
- ❑ `IsOffline`—The database is offline.
- ❑ `IsQuotedIdentifiersEnabled`—Identifiers can be delimited by double quotation marks.
- ❑ `IsReadOnly`—The database is in a read-only mode.
- ❑ `IsRecursiveTriggersEnabled`—The recursive firing of triggers is enabled.
- ❑ `Is ShutDown`—The database encountered a problem during startup.
- ❑ `IsSingleUser`—The database is in single-user mode.
- ❑ `IsSuspect`—The database is suspect.
- ❑ `IsTruncLog`—The database truncates its logon checkpoints.
- ❑ `Version`—The internal version number of the SQL Server code with which the database was created.

The return value from this function will be 1 for true, 0 for false, and `NULL` if the input was not valid—except for `Version` (where the function will return the version number if the database is open and `NULL` if the database is closed).

DATABASEPROPERTYEX

The DATABASEPROPERTYEX function is basically a superset of DATABASEPROPERTY, and also returns the setting for the specified database and property name. The syntax is pretty much just the same as DATABASEPROPERTY and is as follows:

```
DATABASEPROPERTYEX('<database>', '<property>')
```

DATABASEPROPERTYEX just has a few more properties available, including:

- ❑ Collation—Returns the default collation for the database (remember, collations can also be overridden at the column level)
- ❑ ComparisonStyle—Indicates the Windows comparison style (for example, case sensitivity) of the particular collation
- ❑ IsAnsiPaddingEnabled—Whether strings are padded to the same length before comparison or insert
- ❑ IsArithmaticAbortEnabled—Whether queries are terminated when a major arithmetic (such as a data overflow) occurs

The database parameter specifies the name of the database for which data on the named property will be returned. The property parameter contains the name of a database property and can be one of the following values.

DB_ID

The DB_ID function returns the database ID number. The syntax is as follows:

```
DB_ID(['<database_name>'])
```

The optional database_name parameter specifies which database's ID number is required. If the database_name is not given, the current database will be used instead.

DB_NAME

The DB_NAME function returns the name of the database that has the specified ID number. The syntax is as follows:

```
DB_NAME([<database_id>])
```

The optional database_id parameter specifies which database's name is to be returned. If no database_id is given, the name of the current database will be returned.

FILE_ID

The FILE_ID function returns the file ID number for the specified file name in the current database. The syntax is as follows:

```
FILE_ID('<file_name>')
```

Appendix A

The `file_name` parameter specifies the name of the file for which the ID is required.

FILE_NAME

The `FILE_NAME` function returns the file name for the file with the specified file ID number. The syntax is as follows:

```
FILE_NAME(<file_id>)
```

The `file_id` parameter specifies the ID number of the file for which the name is required.

FILEGROUP_ID

The `FILEGROUP_ID` function returns the filegroup ID number for the specified filegroup name. The syntax is as follows:

```
FILEGROUP_ID('<filegroup_name>')
```

The `filegroup_name` parameter specifies the filegroup name of the required filegroup ID.

FILEGROUP_NAME

The `FILEGROUP_NAME` function returns the filegroup name for the specified filegroup ID number. The syntax is as follows:

```
FILEGROUP_NAME(<filegroup_id>)
```

The `filegroup_id` parameter specifies the filegroup ID of the required filegroup name.

FILEGROUPPROPERTY

The `FILEGROUPPROPERTY` returns the setting of a specified filegroup property, given the filegroup and property name. The syntax is as follows:

```
FILEGROUPPROPERTY(<filegroup_name>, <property>)
```

The `filegroup_name` parameter specifies the name of the filegroup that contains the property being queried. The `property` parameter specifies the property being queried and can be one of the following values:

- `IsReadOnly`—The filegroup name is read-only.
- `IsUserDefinedFG`—The filegroup name is a user-defined filegroup.
- `IsDefault`—The filegroup name is the default filegroup.

The return value from this function will be 1 for `True`, 0 for `False`, and `NULL` if the input was not valid.

FILEPROPERTY

The FILEPROPERTY function returns the setting of a specified filename property, given the filename and property name. The syntax is as follows:

```
FILEPROPERTY(<file_name>, <property>)
```

The `file_name` parameter specifies the name of the filegroup that contains the property being queried. The `property` parameter specifies the property being queried and can be one of the following values:

- `IsReadOnly`—The file is read-only.
- `IsPrimaryFile`—The file is the primary file.
- `IsLogFile`—The file is a log file.
- `SpaceUsed`—The amount of space used by the specified file.

The return value from this function will be 1 for `True`, 0 for `False`, and `NULL` if the input was not valid, except for `SpaceUsed` (which will return the number of pages allocated in the file).

FULLTEXTCATALOGPROPERTY

The FULLTEXTCATALOGPROPERTY function returns data about the full-text catalog properties. The syntax is as follows:

```
FULLTEXTCATALOGPROPERTY(<catalog_name>, <property>)
```

The `catalog_name` parameter specifies the name of the full-text catalog. The `property` parameter specifies the property that is being queried. The properties that can be queried are:

- `PopulateStatus`—For which the possible return values are: 0 (idle), 1 (population in progress), 2 (paused), 3 (throttled), 4 (recovering), 5 (shutdown), 6 (incremental population in progress), 7 (updating index)
- `ItemCount`—Returns the number of full-text indexed items currently in the full-text catalog
- `IndexSize`—Returns the size of the full-text index in megabytes
- `UniqueKeyCount`—Returns the number of unique words that make up the full-text index in this catalog
- `LogSize`—Returns the size (in bytes) of the combined set of error logs associated with a full-text catalog
- `PopulateCompletionAge`—Returns the difference (in seconds) between the completion of the last full-text index population and 01/01/1990 00:00:00

FULLTEXTSERVICEPROPERTY

The FULLTEXTSERVICEPROPERTY function returns data about the full-text service-level properties. The syntax is as follows:

Appendix A

`FULLTEXTSERVICEPROPERTY(<property>)`

The `property` parameter specifies the name of the service-level property that is to be queried. The `property` parameter may be one of the following values:

- `ResourceUsage`—Returns a value from 1 (background) to 5 (dedicated)
- `ConnectTimeOut`—Returns the number of seconds that the Search Service will wait for all connections to SQL Server for full-text index population before timing out
- `IsFulltextInstalled`—Returns 1 if Full-Text Service is installed on the computer and a 0 otherwise

`INDEX_COL`

The `INDEX_COL` function returns the indexed column name. The syntax is as follows:

`INDEX_COL('<table>', <index_id>, <key_id>)`

The `table` parameter specifies the name of the table, `index_id` specifies the ID of the index, and `key_id` specifies the ID of the key.

`INDEXKEY_PROPERTY`

This function returns information about the index key.

`INDEXKEY_PROPERTY(<table_id>, <index_id>, <key_id>, <property>)`

The `table_id` parameter is the numerical ID of data type `int`, which defines the table you wish to inspect. Use `OBJECT_ID` to find the numerical `table_id`. `index_id` specifies the ID of the index, and is also of data type `int`. `key_id` specifies the index column position of the key; for example, with a key of three columns, setting this value to 2 will determine that you are wishing to inspect the middle column. Finally, the `property` is the character string identifier of one of two properties you wish to find the setting of. The two possible values are `ColumnId`, which will return the physical column ID, and `IsDescending`, which returns the order that the column is sorted (1 is for descending and 0 is ascending).

`INDEXPROPERTY`

The `INDEXPROPERTY` function returns the setting of a specified index property, given the table ID, index name, and property name. The syntax is as follows:

`INDEXPROPERTY(<table_ID>, <index>, <property>)`

The `property` parameter specifies the property of the index that is to be queried. The `property` parameter can be one of these possible values:

- `IndexDepth`—The depth of the index.
- `IsAutoStatistic`—The index was created by the autocreate statistics option of `sp_dboption`.

- IsClustered—The index is clustered.
- IsStatistics—The index was created by the CREATE STATISTICS statement or by the auto create statistics option of sp_dboption.
- IsUnique—The index is unique.
- IndexFillFactor—The index specifies its own fill factor.
- IsPadIndex—The index specifies space to leave open on each interior node.
- IsFulltextKey—The index is the full-text key for a table.
- IsHypothetical—The index is hypothetical and cannot be used directly as a data access path.

The return value from this function will be 1 for True, 0 for False, and NULL if the input was not valid, except for IndexDepth (which will return the number of levels the index has) and IndexFillFactor (which will return the fill factor used when the index was created or last rebuilt).

OBJECT_ID

The OBJECT_ID function returns the specified database object's ID number. The syntax is as follows:

```
OBJECT_ID('<object>')
```

OBJECT_NAME

The OBJECT_NAME function returns the name of the specified database object. The syntax is as follows:

```
OBJECT_NAME(<object_id>)
```

OBJECTPROPERTY

The OBJECTPROPERTY function returns data about objects in the current database. The syntax is as follows:

```
OBJECTPROPERTY(<id>, <property>)
```

The id parameter specifies the ID of the object required. The property parameter specifies the information required on the object. The following property values are allowed:

CnstIsClustKey	CnstIsColumn
CnstIsDeleteCascade	CnstIsDisabled
CnstIsNonclustKey	CnstIsNotRepl
CnstIsNotTrusted	CnstIsUpdateCascade
ExecIsAfterTrigger	ExecIsAnsiNullsOn
ExecIsDeleteTrigger	ExecIsFirstDeleteTrigger
ExecIsFirstInsertTrigger	ExecIsFirstUpdateTrigger

Appendix A

ExecIsInsertTrigger	ExecIsInsteadOfTrigger
ExecIsLastDeleteTrigger	ExecIsLastInsertTrigger
ExecIsLastUpdateTrigger	ExecIsQuotedIdentOn
ExecIsStartup	ExecIsTriggerDisabled
ExecIsTriggerNotForRepl	ExecIsUpdateTrigger
HasAfterTrigger	HasDeleteTrigger
HasInsertTrigger	HasInsteadOfTrigger
HasUpdateTrigger	IsAnsiNullsOn
IsCheckCnst	IsConstraint
IsDefault	IsDefaultCnst
IsDeterministic	IsExecuted
IsExtendedProc	IsForeignKey
IsIndexed	IsIndexable
IsInlineFunction	IsMSShipped
IsPrimaryKey	IsProcedure
IsQuotedIdentOn	IsQueue
IsReplProc	IsRule
IsScalarFunction	IsSchemaBound
IsSystemTable	IsTable
IsTableFunction	IsTrigger
IsUniqueCnst	IsUserTable
IsView	OwnerId
TableDeleteTrigger	TableDeleteTriggerCount
TableFullTextBackgroundUpdateIndexOn	TableFulltextCatalogId
TableFullTextChangeTrackingOn	TableFulltextDocsProcessed
TableFulltextFailCount	TableFulltextItemCount
TableFulltextKeyColumn	TableFulltextPendingChanges
TableFulltextPopulateStatus	TableHasActiveFulltextIndex
TableHasCheckCnst	TableHasClustIndex
TableHasDefaultCnst	TableHasDeleteTrigger
TableHasForeignKey	TableHasForeignRef
TableHasIdentity	TableHasIndex
TableHasInsertTrigger	TableHasNonclustIndex

TableHasPrimaryKey	TableHasRowGuidCol
TableHasTextImage	TableHasTimestamp
TableHasUniqueCnst	TableHasUpdateTrigger
TableInsertTrigger	TableInsertTriggerCount
TableIsFake	TableIsLockedOnBulkLoad
TableIsPinned	TableTextInRowLimit
TableUpdateTrigger	TableUpdateTriggerCount

The return value from this function will be 1 for True, 0 for False, and NULL if the input was not valid, except for:

- ❑ OwnerId—Returns the database user ID of the owner of that object—note that this is different from the SchemaID of the object and will likely not be that useful in SQL Server 2005 and beyond.
- ❑ TableDeleteTrigger, TableInsertTrigger, TableUpdateTrigger—Return the ID of the first trigger with the specified type. Zero is returned if no trigger of that type exists.
- ❑ TableDeleteTriggerCount, TableInsertTriggerCount, TableUpdateTriggerCount—Return the number of the specified type of trigger that exists for the table in question.
- ❑ TableFulltextCatalogId—Returns the ID of the full-text catalog if there is one, and zero if no full-text catalog exists for that table.
- ❑ TableFulltextKeyColumn—Returns the ColumnID of the column being utilized as the unique index for that full-text index.
- ❑ TableFulltextPendingChanges—The number of entries that have changed since the last full-text analysis was run for this table. Change tracking must be enabled for this function to return useful results.
- ❑ TableFulltextPopulateStatus—This one has multiple possible return values:
 - ❑ 0—Indicates that the full-text process is currently idle.
 - ❑ 1—A full population run is currently in progress.
 - ❑ 2—An incremental population is currently running.
 - ❑ 3—Changes are currently being analyzed and added to the full-text catalog.
 - ❑ 4—Some form of background update (such as that done by the automatic change tracking mechanism) is currently running.
 - ❑ 5—A full-text operation is in progress, but has either been throttled (to allow other system requests to perform as needed) or has been paused.

You can use the feedback from this option to make decisions about what other full-text-related options are appropriate (to check whether a population is in progress so you know whether other functions, such as TableFulltextDocsProcessed, are valid).

- ❑ TableFulltextDocsProcessed—Valid only while full-text indexing is actually running, this returns the number of rows processed since the full-text index processing task started. A zero result indicates that full-text indexing is not currently running (a null result means full-text indexing is not configured for this table).

Appendix A

- ❑ `TableFulltextFailCount` — Valid only while full-text indexing is actually running, this returns the number of rows that full-text indexing has, for some reason, skipped (no indication of reason). As with `TableFulltextDocsProcessed`, a zero result indicates the table is not currently being analyzed for full text, and a null indicates that full text is not configured for this table.
- ❑ `TableIsPinned` — This is left in for backward compatibility only and will always return “0” in SQL Server 2005 and beyond.

OBJECTPROPERTYEX

`OBJECTPROPERTYEX` is an extended version of the `OBJECTPROPERTY` function.

```
OBJECTPROPERTYEX(<id>, <property>)
```

Like `OBJECTPROPERTY`, the `id` parameter specifies the ID of the object required. The `property` parameter specifies the information required on the object. `OBJECTPROPERTYEX` supports all the same property values as `OBJECTPROPERTY` but adds the following property values as additional options:

- ❑ `BaseType` — Returns the base data type of an object.
- ❑ `IsPrecise` — Indicates that your object does not contain any imprecise computations. For example an int or decimal is precise, but a float is not — computations that utilize imprecise data types must be assumed to return imprecise results. Note that you can specifically mark any .NET assemblies you produce as being precise or not.
- ❑ `IsSystemVerified` — Indicates whether the `IsPrecise` and `IsDeterministic` properties can be verified by SQL Server itself (as opposed to just having been set by the user).
- ❑ `SchemaId` — Just what it sounds like — Returns the internal system ID for a given object. You can then use `SCHEMA_NAME` to put a more user-friendly name on the schema ID.
- ❑ `SystemDataAccess` — Indicates whether the object in question relies on any system table data.
- ❑ `User DataAccess` — Indicates whether the object in question utilizes any of the user tables or system user data.

SCHEMA_ID

Given a schema name, returns the internal system ID for that schema. Utilizes the syntax:

```
SCHEMA_ID( <schema name> )
```

SCHEMA_NAME

Given an internal schema system ID, returns the user-friendly name for that schema. The syntax is:

```
SCHEMA_NAME( <schema id> )
```

SQL_VARIANT_PROPERTY

`SQL_VARIANT_PROPERTY` is a powerful function and returns information about a `sql_variant`. This information could be from `BaseType`, `Precision`, `Scale`, `TotalBytes`, `Collation`, `MaxLength`. The syntax is:

`SQL_VARIANT_PROPERTY (expression, property)`

Expression is an expression of type `sql_variant`. Property can be any one of the following values:

Value	Description	Base Type of <code>sql_variant</code> Returned
BaseType	Data types include: <code>char</code> , <code>int</code> , <code>money</code> , <code>nchar</code> , <code>ntext</code> , <code>numeric</code> , <code>nvarchar</code> , <code>real</code> , <code>smalldatetime</code> , <code>smallint</code> , <code>smallmoney</code> , <code>text</code> , <code>timestamp</code> , <code>tinyint</code> , <code>uniqueidentifier</code> , <code>varbinary</code> , <code>varchar</code>	sysname
Precision	The precision of the numeric base data type: datetime = 23 smalldatetime = 16 float = 53 real = 24	int
	<code>decimal (p,s)</code> and <code>numeric (p,s) = p</code> money = 19 smallmoney = 10 int = 10 smallint = 5 tinyint = 3 bit = 1	
	All other types = 0	
Scale	The number of digits to the right of the decimal point of the numeric base data type: <code>decimal (p,s)</code> and <code>numeric (p,s) = s</code>	int
	money and <code>smallmoney = 4</code> datetime = 3	
	All other types = 0	
TotalBytes	The number of bytes required to hold both the metadata and data of the value. If the value is greater than 900, index creation will fail.	int
Collation	The collation of the particular <code>sql_variant</code> value.	sysname
MaxLength	The maximum data type length, in bytes.	int

Appendix A

TYPEPROPERTY

The TYPEPROPERTY function returns information about a data type. The syntax is as follows:

```
TYPEPROPERTY(<type>, <property>)
```

The `type` parameter specifies the name of the data type. The `property` parameter specifies the property of the data type that is to be queried; it can be one of the following values:

- Precision—Returns the number of digits/characters
- Scale—Returns the number of decimal places
- AllowsNull—Returns 1 for True and 0 for False
- UsesAnsiTrim—Returns 1 for True and 0 for False

Rowset Functions

The rowset functions return an object that can be used in place of a table reference in a T-SQL statement. The rowset functions are:

- CONTAINSTABLE
- FREETEXTTABLE
- OPENDATASOURCE
- OPENQUERY
- OPENROWSET
- OPENXML

CONTAINSTABLE

The CONTAINSTABLE function is used in full-text queries. Please refer to Chapter 21 for an example of its usage. The syntax is as follows:

```
CONTAINSTABLE (<table>, {<column> | *}, '<contains_search_condition>')
```

FREETEXTTABLE

The FREETEXTTABLE function is used in full-text queries. Please refer to Chapter 21 for an example of its usage. The syntax is as follows:

```
FREETEXTTABLE (<table>, {<column> | *}, '<freetext_string>')
```

OPENDATASOURCE

The OPENDATASOURCE function provides ad hoc connection information. The syntax is as follows:

```
OPENDATASOURCE (<provider_name>, <init_string>)
```

The provider_name is the name registered as the ProgID of the OLE DB provider used to access the data source. The init_string should be familiar to VB programmers, as this is the initialization string to the OLE DB provider. For example, the init_string could look like:

```
"User Id=wonderison;Password=JuniorBlues;DataSource=MyServerName"
```

OPENQUERY

The OPENQUERY function executes the specified pass-through query on the specified linked_server. The syntax is as follows:

```
OPENQUERY(<linked_server>, '<query>')
```

OPENROWSET

The OPENROWSET function accesses remote data from an OLE DB data source. The syntax is as follows:

```
OPENROWSET('<provider_name>'  
{  
    '<datasource>'; '<user_id>'; '<password>'  
    | '<provider_string>'  
,  
    {  
        [<catalog.>] [<schema.>]<object>  
        | '<query>'  
    })
```

The provider_name parameter is a string representing the friendly name of the OLE DB provider as specified in the registry. The data_source parameter is a string corresponding to the required OLE DB data source. The user_id parameter is a relevant username to be passed to the OLE DB provider. The password parameter is the password associated with the user_id.

The provider_string parameter is a provider-specific connection string and is used in place of the datasource, user_id, and password combination.

The catalog parameter is the name of catalog/database that contains the required object. The schema parameter is the name of the schema or object owner of the required object. The object parameter is the object name.

The query parameter is a string that is executed by the provider and is used instead of a combination of catalog, schema, and object.

OPENXML

By passing in an XML document as a parameter, or by retrieving an XML document and defining the document within a variable, OPENXML allows you to inspect the structure and return data, as if the XML document were a table. The syntax is as follows:

```
OPENXML(<idoc_int> [in],<rowpattern> nvarchar[in], [<flags> byte[in]])  
[WITH (<SchemaDeclaration> | <TableName>)]
```

Appendix A

The `idoc_int` parameter is the variable defined using the `sp_xml_preparedocument` system sproc. `Rowpattern` is the node definition. The `flags` parameter specifies the mapping between the XML document and the rowset to return within the `SELECT` statement. `SchemaDeclaration` defines the XML schema for the XML document, if there is a table defined within the database that follows the XML schema, then `TableName` can be used instead.

Before being able to use the XML document, it must be prepared by using the `sp_xml_preparedocument` system procedure.

Security Functions

The security functions return information about users and roles. They are:

- ❑ HAS_DBACCESS
- ❑ IS_MEMBER
- ❑ IS_SRVROLEMEMBER
- ❑ SUSER_ID
- ❑ SUSER_NAME
- ❑ SUSER_SID
- ❑ USER
- ❑ USER_ID

HAS_DBACCESS

The `HAS_DBACCESS` function is used to determine whether the user that is logged in has access to the database being used. A return value of 1 means the user does have access, and a return value of 0 means that he or she does not. A NULL return value means the `database_name` supplied was invalid. The syntax is as follows:

```
HAS_DBACCESS ('<database_name>')
```

IS_MEMBER

The `IS_MEMBER` function returns whether the current user is a member of the specified Windows NT group/SQL Server role. The syntax is as follows:

```
IS_MEMBER ({'<group>' | '<role>'})
```

The `group` parameter specifies the name of the NT group and must be in the form `domain\group`. The `role` parameter specifies the name of the SQL Server role. The role can be a database fixed role or a user-defined role but cannot be a server role.

This function will return a 1 if the current user is a member of the specified group or role, a 0 if the current user is not a member of the specified group or role, and NULL if the specified group or role is invalid.

IS_SRVROLEMEMBER

The **IS_SRVROLEMEMBER** function returns whether a user is a member of the specified server role. The syntax is as follows:

```
IS_SRVROLEMEMBER ('<role>' [, '<login>'])
```

The optional **login** parameter is the name of the login account to check—the default is the current user. The **role** parameter specifies the server role and must be one of the following possible values:

- sysadmin
- dbcreator
- diskadmin
- processadmin
- serveradmin
- setupadmin
- securityadmin

This function returns a 1 if the specified login account is a member of the specified role, a 0 if the login is not a member of the role, and a NULL if the role or login is invalid.

SUSER_ID

The **SUSER_ID** function returns the specified user's login ID number. The syntax is as follows:

```
SUSER_ID(['<login>'])
```

The **login** parameter is the specified user's login ID name. If no value for **login** is provided, the default of the current user will be used instead.

The SUSER_ID system function is included in SQL Server 2000 for backward compatibility, so if possible you should use SUSER_SID instead.

SUSER_NAME

The **SUSER_NAME** function returns the specified user's login ID name. The syntax is as follows:

```
SUSER_NAME([<server_user_id>])
```

The **server_user_id** parameter is the specified user's login ID number. If no value for **server_user_id** is provided, the default of the current user will be used instead.

The SUSER_NAME system function is included in SQL Server 2000 for backward compatibility only, so if possible you should use SUSER_SNAME instead.

Appendix A

SUSER_SID

The SUSER_SID function returns the security identification number (SID) for the specified user. The syntax is as follows:

```
SUSER_SID(['<login>'])
```

The `login` parameter is the user's login name. If no value for `login` is provided, the current user will be used instead.

SUSER_SNAME

The SUSER_SNAME function returns the login ID name for the specified security identification number (SID). The syntax is as follows:

```
SUSER_SNAME(<server_user_sid>)
```

The `server_user_sid` parameter is the user's SID. If no value for the `server_user_sid` is provided, the current user's will be used instead.

USER

The USER function allows a system-supplied value for the current user's database username to be inserted into a table if no default has been supplied. The syntax is as follows:

```
USER
```

USER_ID

The USER_ID function returns the specified user's database ID number. The syntax is as follows:

```
USER_ID(['<user>'])
```

The `user` parameter is the username to be used. If no value for `user` is provided, the current user is used.

String Functions

The string functions perform actions on string values and return strings or numeric values. The string functions are:

- ASCII
- CHAR
- CHARINDEX
- DIFFERENCE
- LEFT

- LEN
- LOWER
- LTRIM
- NCHAR
- PATINDEX
- QUOTENAME
- REPLACE
- REPLICATE
- REVERSE
- RIGHT
- RTRIM
- SOUNDEX
- SPACE
- STR
- STUFF
- SUBSTRING
- UNICODE
- UPPER

ASCII

The ASCII function returns the ASCII code value of the leftmost character in `character_expression`.
The syntax is as follows:

```
ASCII(<character_expression>)
```

CHAR

The CHAR function converts an ASCII code (specified in expression) into a string. The syntax is as follows:

```
CHAR(<expression>)
```

The expression can be any integer between 0 and 255.

CHARINDEX

The CHARINDEX function returns the starting position of an expression in a character_string. The syntax is as follows:

```
CHARINDEX(<expression>, <character_string> [, <start_location>])
```

Appendix A

The `expression` parameter is the string, which is to be found. The `character_string` is the string to be searched, usually a column. The `start_location` is the character position to begin the search, if this is anything other than a positive number, the search will begin at the start of `character_string`.

DIFFERENCE

The `DIFFERENCE` function returns the difference between the `SOUNDEX` values of two expressions as an integer. The syntax is as follows:

```
DIFFERENCE(<expression1>, <expression2>)
```

This function returns an integer value between 0 and 4. If the two expressions sound identical (for example, blue and blew) a value of 4 will be returned. If there is no similarity, a value of 0 is returned.

LEFT

The `LEFT` function returns the leftmost part of an expression, starting a specified number of characters from the left. The syntax is as follows:

```
LEFT(<expression>, <integer>)
```

The `expression` parameter contains the character data from which the leftmost section will be extracted. The `integer` parameter specifies the number of characters from the left to begin—it must be a positive integer.

LEN

The `LEN` function returns the number of characters in the specified `expression`. The syntax is as follows:

```
LEN(<expression>)
```

LOWER

The `LOWER` function converts any uppercase characters in the `expression` into lowercase characters. The syntax is as follows:

```
LOWER(<expression>)
```

LTRIM

The `LTRIM` function removes any leading blanks from a `character_expression`. The syntax is as follows:

```
LTRIM(<character_expression>)
```

NCHAR

The `NCHAR` function returns the Unicode character that has the specified `integer_code`. The syntax is as follows:

```
NCHAR(<integer_code>)
```

The `integer_code` parameter must be a positive whole number from 0 to 65,535.

PATINDEX

The `PATINDEX` function returns the starting position of the first occurrence of a pattern in a specified expression or zero if the pattern was not found. The syntax is as follows:

```
PATINDEX('<%pattern%>', <expression>)
```

The `pattern` parameter is a string that will be searched for. Wildcard characters can be used, but the % characters must surround the pattern. The `expression` parameter is character data in which the pattern is being searched for—usually a column.

QUOTENAME

The `QUOTENAME` function returns a Unicode string with delimiters added to make the specified string a valid SQL Server delimited identifier. The syntax is as follows:

```
QUOTENAME('<character_string>[, '<quote_character>'])
```

The `character_string` parameter is Unicode string. The `quote_character` parameter is a one-character string that will be used as a delimiter. The `quote_character` parameter can be a single quotation mark ('), a left or a right bracket ([]), or a double quotation mark (")—the default is for brackets to be used.

REPLACE

The `REPLACE` function replaces all instances of second specified string in the first specified string with a third specified string. The syntax is as follows:

```
REPLACE('<string_expression1>', '<string_expression2>', '<string_expression3>')
```

The `string_expression1` parameter is the expression in which to search. The `string_expression2` parameter is the expression to search for in `string_expression1`. The `string_expression3` parameter is the expression with which to replace all instances of `string_expression2`.

REPLICATE

The `REPLICATE` function repeats a `character_expression` a specified number of times. The syntax is as follows:

```
REPLICATE(<character_expression>, <integer>)
```

REVERSE

The `REVERSE` function returns the reverse of the specified `character_expression`. The syntax is as follows:

```
REVERSE(<character_expression>)
```

Appendix A

RIGHT

The **RIGHT** function returns the rightmost part of the specified `character_expression`, starting a specified number of characters (given by `integer`) from the right. The syntax is as follows:

```
RIGHT(<character_expression>, <integer>)
```

The `integer` parameter must be a positive whole number.

RTRIM

The **RTRIM** function removes all the trailing blanks from a specified `character_expression`. The syntax is as follows:

```
RTRIM(<character_expression>)
```

SOUNDEX

The **SOUNDEX** function returns a four-character (SOUNDEX) code, which can be used to evaluate the similarity of two strings. The syntax is as follows:

```
SOUNDEX(<character_expression>)
```

SPACE

The **SPACE** function returns a string of repeated spaces, the length of which is indicated by `integer`. The syntax is as follows:

```
SPACE(<integer>)
```

STR

The **STR** function converts numeric data into character data. The syntax is as follows:

```
STR(<numeric_expression>[, <length>[, <decimal>]])
```

The `numeric_expression` parameter is a numeric expression with a decimal point. The `length` parameter is the total length including decimal point, digits, and spaces. The `decimal` parameter is the number of places to the right of the decimal point.

STUFF

The **STUFF** function deletes a specified length of characters and inserts another set of characters in their place. The syntax is as follows:

```
STUFF(<expression>, <start>, <length>, <characters>)
```

The expression parameter is the string of characters in which some will be deleted and new ones added. The start parameter specifies where to begin deletion and insertion of characters. The length parameter specifies the number of characters to delete. The characters parameter specifies the new set of characters to be inserted into the expression.

SUBSTRING

The SUBSTRING function returns part of an expression. The syntax is as follows:

```
SUBSTRING(<expression>, <start>, <length>)
```

The expression parameter specifies the data from which the substring will be taken, and can be a character string, binary string, text, or an expression that includes a table. The start parameter is an integer that specifies where to begin the substring. The length parameter specifies how long the substring is.

UNICODE

The UNICODE function returns the Unicode number that represents the first character in character_expression. The syntax is as follows:

```
UNICODE('<character_expression>')
```

UPPER

The UPPER function converts all the lowercase characters in character_expression into uppercase characters. The syntax is as follows:

```
UPPER(<character_expression>)
```

System Functions

The system functions can be used to return information about values, objects and settings with SQL Server. The functions are as follows:

- APP_NAME
- CASE
- CAST and CONVERT
- COALESCE
- COLLATIONPROPERTY
- CURRENT_TIMESTAMP
- CURRENT_USER
- DATALENGTH
- FORMATMESSAGE
- GETANSINULL

Appendix A

- HOST_ID
- HOST_NAME
- IDENT_CURRENT
- IDENT_INCR
- IDENT_SEED
- IDENTITY
- ISDATE
- ISNULL
- ISNUMERIC
- NEWID
- NULLIF
- PARSENAME
- PERMISSIONS
- ROWCOUNT_BIG
- SCOPE_IDENTITY
- SERVERPROPERTY
- SESSION_USER
- SESSIONPROPERTY
- STATS_DATE
- SYSTEM_USER
- USER_NAME

APP_NAME

The APP_NAME function returns the application name for the current session if one has been set by the application as an nvarchar type. It has the following syntax:

```
APP_NAME()
```

CASE

The CASE function evaluates a list of conditions and returns one of multiple possible results. It also has two formats:

- The simple CASE function compares an expression to a set of simple expressions to determine the result.
- The searched CASE function evaluates a set of Boolean expressions to determine the result.

Both formats support an optional ELSE argument.

Simple CASE function:

```
CASE <input_expression>
    WHEN <when_expression> THEN <result_expression>
    ELSE <else_result_expression>
END
```

Searched CASE function:

```
CASE
    WHEN <Boolean_expression> THEN <result_expression>
    ELSE <else_result_expression>
END
```

CAST and CONVERT

These two functions provide similar functionality in that they both convert one data type into another type.

Using CAST:

```
CAST(<expression> AS <data_type>)
```

Using CONVERT:

```
CONVERT (<data_type>[(<length>)], <expression> [, <style>])
```

Where style refers to the style of date format when converting to a character data type.

COALESCE

The COALESCE function is passed an undefined number of arguments and it tests for the first non-null expression among them. The syntax is as follows:

```
COALESCE(<expression> [,...n])
```

If all arguments are NULL then COALESCE returns NULL.

COLLATIONPROPERTY

The COLLATIONPROPERTY function returns the property of a given collation. The syntax is as follows:

```
COLLATIONPROPERTY(<collation_name>, <property>)
```

The `collation_name` parameter is the name of the collation you wish to use, and `property` is the property of the collation you wish to determine. This can be one of three values:

Appendix A

Property Name	Description
CodePage	The non-Unicode code page of the collation.
LCID	The Windows LCID of the collation. Returns NULL for SQL collations.
ComparisonStyle	The Windows comparison style of the collation. Returns NULL for binary or SQL collations.

CURRENT_TIMESTAMP

The CURRENT_TIMESTAMP function simply returns the current date and time as a datetime type. It is equivalent to GETDATE(). The syntax is as follows:

```
CURRENT_TIMESTAMP
```

CURRENT_USER

The CURRENT_USER function simply returns the current user as a sysname type. It is equivalent to USER_NAME(). The syntax is as follows:

```
CURRENT_USER
```

DATALENGTH

The DATALENGTH function returns the number of bytes used to represent expression as an integer. It is especially useful with varchar, varbinary, text, image, nvarchar, and ntext data types because these data types can store variable-length data. The syntax is as follows:

```
DATALENGTH(<expression>)
```

FORMATMESSAGE

The FORMATMESSAGE function uses existing messages in sysmessages to construct a message. The syntax is as follows:

```
FORMATMESSAGE(<msg_number>, <param_value>[,...n])
```

Where msg_number is the ID of the message in sysmessages.

FORMATMESSAGE looks up the message in the current language of the user. If there is no localized version of the message, the U.S. English version is used.

GETANSINULL

The GETANSINULL function returns the default nullability for a database as an integer. The syntax is as follows:

```
GETANSINULL(['<database>'])
```

The `database` parameter is the name of the database for which to return nullability information.

When the nullability of the given database allows `NULL` values and the column or data type nullability is not explicitly defined, `GETANSINULL` returns 1. This is the ANSI `NUL` default.

HOST_ID

The `HOST_ID` function returns the ID of the workstation. The syntax is as follows:

```
HOST_ID()
```

HOST_NAME

The `HOST_NAME` function returns the name of the workstation. The syntax is as follows:

```
HOST_NAME()
```

IDENT_CURRENT

The `IDENT_CURRENT` function returns the last identity value created for a table, within any session or scope of that table. This is exactly like `@@IDENTITY` and `SCOPE_IDENTITY`; however, this has no limit to the scope of its search to return the value.

The syntax is as follows:

```
IDENT_CURRENT('<table_name>')
```

The `table_name` is the table for which you wish to find the current identity.

IDENT_INCR

The `IDENT_INCR` function returns the increment value specified during the creation of an identity column in a table or view that has an identity column. The syntax is as follows:

```
IDENT_INCR('<table_or_view>')
```

The `table_or_view` parameter is an expression specifying the table or view to check for a valid identity increment value.

IDENT_SEED

The `IDENT_SEED` function returns the seed value specified during the creation of an identity column in a table or a view that has an identity column. The syntax is as follows:

```
IDENT_SEED('<table_or_view>')
```

The `table_or_view` parameter is an expression specifying the table or view to check for a valid identity increment value.

Appendix A

IDENTITY

The **IDENTITY** function is used to insert an identity column into a new table. It is used only with a **SELECT** statement with an **INTO** table clause. The syntax is as follows:

```
IDENTITY(<data_type>[, <seed>, <increment>]) AS <column_name>
```

Where:

- ❑ **data_type** is the data type of the identity column.
- ❑ **seed** is the value to be assigned to the first row in the table. Each subsequent row is assigned the next identity value, which is equal to the last **IDENTITY** value plus the **increment** value. If neither **seed** nor **increment** is specified, both default to 1.
- ❑ **increment** is the increment to add to the **seed** value for successive rows in the table.
- ❑ **column_name** is the name of the column that is to be inserted into the new table.

ISDATE

The **ISDATE** function determines whether an input expression is a valid date. The syntax is as follows:

```
ISDATE(<expression>)
```

ISNULL

The **ISNULL** function checks an expression for a **NULL** value and replaces it with a specified replacement value. The syntax is as follows:

```
ISNULL(<check_expression>, <replacement_value>)
```

ISNUMERIC

The **ISNUMERIC** function determines whether an expression is a valid numeric type. The syntax is as follows:

```
ISNUMERIC(<expression>)
```

NEWID

The **NEWID** function creates a unique value of type **uniqueidentifier**. The syntax is as follows:

```
NEWID()
```

NULLIF

The **NULLIF** function compares two expressions and returns a **NULL** value. The syntax is as follows:

```
NULLIF(<expression1>, <expression2>)
```

PARSENAME

The PARSENAME function returns the specified part of an object name. The syntax is as follows:

```
PARSENAME ('<object_name>', <object_piece>)
```

The `object_name` parameter specifies the object name from the part that is to be retrieved. The `object_piece` parameter specifies the part of the object to return. The `object_piece` parameter takes one of these possible values:

- 1 — Object name
- 2 — Owner name
- 3 — Database name
- 4 — Server name

PERMISSIONS

The PERMISSIONS function returns a value containing a bitmap, which indicates the statement, object, or column permissions for the current user. The syntax is as follows:

```
PERMISSIONS ([<objectid> [, '<column>']] )
```

The `object_id` parameter specifies the ID of an object. The optional `column` parameter specifies the name of the column for which permission information is being returned.

ROWCOUNT_BIG

The ROWCOUNT_BIG function is very similar to @@ROWCOUNT in that it returns the number of rows from the last statement. However, the value returned is of a data type of bigint. The syntax is as follows:

```
ROWCOUNT_BIG()
```

SCOPE_IDENTITY

The SCOPE_IDENTITY function returns the last value inserted into an identity column in the same scope (that is, within the same sproc, trigger, function, or batch). This is similar to IDENT_CURRENT, discussed above, although that was not limited to identity insertions made in the same scope.

This function returns a sql_variant data type, and the syntax is as follows:

```
SCOPE_IDENTITY()
```

SERVERTPROPERTY

The SERVERPROPERTY function returns information about the server you are running on. The syntax is as follows:

```
SERVERPROPERTY ('<propertyname>')
```

Appendix A

The possible values for `propertyname` are:

Property Name	Values Returned
Collation	The name of the default collation for the server.
Edition	The edition of the SQL Server instance installed on the server. Returns one of the following nvarchar results: 'Desktop Engine' 'Developer Edition' 'Enterprise Edition' 'Enterprise Evaluation Edition' 'Personal Edition' 'Standard Edition'
Engine Edition	The engine edition of the SQL Server instance installed on the server: 1—Personal or Desktop Engine 2—Standard 3—Enterprise (returned for Enterprise, Enterprise Evaluation, and Developer)
InstanceName	The name of the instance to which the user is connected.
IsClustered	Will determine if the server instance is configured in a failover cluster: 1—Clustered 0—Not clustered NULL—Invalid input or error
IsFullText Installed	To determine if the full-text component is installed with the current instance of SQL Server: 1—Full-text is installed. 0—Full-text is not installed. NULL—Invalid input or error
IsIntegrated SecurityOnly	To determine if the server is in integrated security mode: 1—Integrated security 0—Not integrated security NULL—Invalid input or error
IsSingleUser	To determine if the server is a single-user installation: 1—Single user 0—Not single user NULL—Invalid input or error
IsSync WithBackup	To determine if the database is either a published database or a distribution database, and can be restored without disrupting the current transactional replication: 1—True 0—False

Property Name	Values Returned
LicenseType	What type of license is installed for this instance of SQL Server: PER_SEAT — Per-seat mode PER_PROCESSOR — Per-processor mode DISABLED — Licensing is disabled
MachineName	Returns the Windows NT computer name on which the server instance is running. For a clustered instance (an instance of SQL Server running on a virtual server on Microsoft Cluster Server), it returns the name of the virtual server.
NumLicenses	Number of client licenses registered for this instance of SQL Server, if in per-seat mode. Number of processors licensed for this instance of SQL Server, if in per-processor mode.
ProcessID	Process ID of the SQL Server service. (The ProcessID is useful in identifying which sqllservr.exe belongs to this instance.)
ProductVersion	Very much like Visual Basic projects, in that the version details of the instance of SQL Server, are returned, in the form of 'major.minor.build'.
ProductLevel	Returns the value of the version of the SQL Server instance currently running. Returns: 'RTM' — Shipping version 'SPn' — Service pack version 'Bn' — Beta version
ServerName	Both the Windows NT server and instance information associated with a specified instance of SQL Server.

The SERVERPROPERTY function is very useful for multi-sited corporations where developers need to find out information from a server.

SESSION_USER

The SESSION_USER function allows a system-supplied value for the current session's username to be inserted into a table if no default value has been specified. The syntax is as follows:

```
SESSION_USER
```

SESSIONPROPERTY

The SESSIONPROPERTY function is used to return the SET options for a session. The syntax is as follows:

```
SESSIONPROPERTY (<option>)
```

Appendix A

This function is useful when there are stored procedures that are altering session properties in specific scenarios. This function should rarely be used as you should not alter too many of the SET options during runtime.

STATS_DATE

The STATS_DATE function returns the date that the statistics for the specified index were last updated. The syntax is as follows:

```
STATS_DATE(<table_id>, <index_id>)
```

SYSTEM_USER

The SYSTEM_USER function allows a system-supplied value for the current system username to be inserted into a table if no default value has been specified. The syntax is as follows:

```
SYSTEM_USER
```

USER_NAME

The USER_NAME returns a database username. The syntax is as follows:

```
USER_NAME([<id>])
```

The id parameter specifies the ID number of the required username, if no value is given the current user is assumed.

Text and Image Functions

The text and image functions perform operations on text or image data. They are:

- ❑ PATINDEX (This was covered in the “String Functions” section earlier in the appendix.)
- ❑ TEXTPTR
- ❑ TEXTVALID

TEXTPTR

The TEXTPTR function checks the value of the text pointer that corresponds to a text, ntext, or image column and returns a varbinary value. The text pointer should be checked to ensure that it points to the first text page before running READTEXT, WRITETEXT, and UPDATE statements. The syntax is as follows:

```
TEXTPTR(<column>)
```

TEXTVALID

The TEXTVALID function checks whether a specified text pointer is valid. The syntax is as follows:

```
TEXTVALID('<table.column>', <text_ptr>)
```

The `table.column` parameter specifies the name of the table and column to be used. The `text_ptr` parameter specifies the text pointer to be checked.

This function will return 0 if the pointer is invalid and 1 if the pointer is valid.

B

Connectivity

Having a SQL Server but not allowing programs to connect to it is the same as not having a SQL Server at all. Sure, we may log into Management Studio and write queries directly, but the reality is that the vast majority of our users out there never actually see the database directly — they are just using input and reporting screens in some system we've written.

With this in mind, it probably makes sense to figure out how your application is actually going to talk to the database. There are tons of books out there that cover this topic directly (and, outside of a basic connection, it really is a huge topic unto itself), so we're going to stick a few basic methods of connecting and also some information on do's and don'ts of connectivity.

I can't stress enough how these examples are truly the basics. You can make many, many choices and optimizations for connectivity. I'll touch on a few key things about it here and there, but this is mostly just code. I highly recommend taking a look at a connectivity-specific book.

Some General Concepts

Before we get going too deep with "just code," there are a few key constructs that we need to understand. There are several different connectivity models that have come and gone over time — you'll hear about such names as OLE-DB, ODBC, and, of course, ADO among others. The connectivity model of the day these days is ADO.NET. Given the life span of these things, I wouldn't be surprised at all if there were yet a different one by the time the next version of SQL Server comes out.

Even with all the different models that have come and gone, some concepts seem to always exist in every object model — let's take a look at these real quick:

- ❑ **Connection** — The connection object is pretty much what it sounds like — the object that defines and establishes your actual communications link to your database. The kinds of parameter information that will exist for a connection object include such things as the username, password, database, and server you wish to connect to. Some of the methods that will exist include such things as connect and disconnect.

Appendix B

- ❑ **Command**—This object is the one that carries information about what it is you want to do. Some object models will not include this object, or at least not feature it, but the concept is always there (it is sometimes hidden as a method of the connection object).
- ❑ **Data set**—This is the result of a query—that is, if the query returns data. Some queries you execute—for example, a simple `INSERT` statement—will not return results, but, if results are returned, there will be some sort of data set (sometimes called a result set or recordset) that the query returns them into. Data set objects will generally allow for you to iterate through the records in them (often forward only in direction but usually settable to allow for more robust positioning). They will also generally allow for data to be updated, inserted, and deleted.

General Performance Considerations

In many places in this book, we talk about things you can do on the server side to help your system perform well. Keep in mind, however, that there are a lot of places your system can have performance issues introduced—not just in your server-side components. The manner and frequency of your connections can play a significant role in performance. Here are some key issues to think about.

Connection Management

The opening of a connection can be one of the most expensive parts of any query. Exactly how long it takes will vary depending on a wide variety of issues, but, even in the best of scenarios, a connection can take several times as long to open as the actual execution will take for most simple queries.

So—how to manage this? Fortunately, utilizing some basics will eliminate most of the issue:

- ❑ **Utilize “connection pooling” if possible**—Connection pooling is a concept that is built into most modern connectivity methods and is usually turned on by default. Under connection pooling, the underlying connectivity model you’re using (ODBC, OLE-DB, SQL Native Client) “pools” the connections on your client. This means that when you close a connection in your code, the connection is not actually immediately closed to the server—instead, the connection is put into a pool for possible reuse. When another block of code (even from a different process on your client) makes a connection with the same connection properties (server, username, password, protocol), the client will reuse the existing open connection rather than create a totally new one—saving all of that connection overhead. This is not without its risks, however, so I’ll devote a small section to this in a bit.
- ❑ **Share connections within your process**—If your application does not issue queries asynchronously, then it most likely only needs one connection to the server. In such a case, consider keeping that connection global to your application or at least to a given functional area (the larger the application, the more hassle sharing a connection is as multiple developers work on things, so consider those risks when deciding how wide of a scope the connection is used in). Open the connection the first time you need it and then reference it from many functions. Avoid opening and closing a new connection in each SQL related function you have.
- ❑ **Properly close connections**—At whatever level you’re creating connections, be sure to actively close the connection. If you’re opening one connection that is being used globally in the application, then close it as part of your shutdown code. Do not just assume that the connection will be

closed when your connection object goes out of scope. It indeed should be closed automatically, but I've found this to be something less than reliable (though the server should also eventually time the connection out), and extra connections waste memory both on the client and server side.

- ❑ **Be careful with unusual connection settings**—This is most problematic when utilizing pooled connections, but be careful about using SQL Server SET options or anything else that changes connection behavior if that connection may be later utilized by a different function. Connection pooling is a concept that is not database system specific (Oracle, MySQL, and other connections may also be connection pooled), so the connection pool manager may not recognize that your connection is now different from other connections with the same login time connection properties. So, imagine, for example, that we set the transaction isolation level to be something other than the default—this could have a catastrophic effect on a different process that utilized that connection thinking that the default isolation level was in effect.
- ❑ **Consider the use of Multiple Active Result Sets (MARS)**—This is new with SQL Server 2005 and allows for multiple asynchronous operations to be performed on the same connection. Each active “environment” is copied from a default environment for that connection but then operates relatively independently (but goes away if the connection is destroyed).

Roundtrips

This is a positively huge performance area, so listen up extra close here!

We just did a section on connection management. As we saw in that section, the overhead of creating the connection was very pricey indeed. Keep in mind, however, that there is a smaller, but similar, overhead each time you initiate a separate use of that connection. Each time you issue a request, that request is wrapped up in your network protocol, sent along the wire (negotiating whatever connectivity issues may exist at the time), and decoded at the other end. Each request may have a short process queue to contend with.

For a single request, the time involved in this is very short (usually milliseconds to tens of milliseconds) from a usability perspective. Imagine, however, that we are issuing hundreds, thousands, or even millions of separate server requests in order to complete one functional unit of code!

I can almost hear a bunch of you out there laughing out loud and saying to yourself how stupid such a thing would be and how you would never do that. Well, the “you would never do that” part sounds nice, but I’ve seen a lot —let me say it again, a lot —of otherwise very smart developers create code that does this very thing. Often they did not realize how many times the offending code would be called, and often they just didn’t think about it. Well, please be one of the ones who do think about it!

This manifests itself in a few different ways, so here are some of the common things I see.

Hierarchies

This problem occurs very frequently in user interface code, though it can also happen frequently on an object model that implements a hierarchy (for example, an object that contains collections of other objects).

Let’s say you have a table or other hierarchical display where there is some heading followed by some detail data. Sometimes the headings have several levels to them. A very common approach is to have a separate function that encapsulates each level of the hierarchy. Indeed, it’s not at all uncommon for this

Appendix B

to just be some form of recursive call to the same function for all or at least part of the hierarchy. Since your function is responsible for building just its one level of output, it receives a parameter and utilizes it to issue a query for its part of the data. With that information in hand, it constructs what it needs to and either calls another layer to do the same thing or returns leaving the calling function to repeat the process as needed.

What we have in this scenario are several small queries where, quite often, one or very few queries would have done the trick. Consider a design that allows you to fetch all of your data in one or just a few master queries and then pass the result set(s) into your function. Presto! You make call the function many, many times, but you have trimmed things down such that only a few separate requests—or “roundtrips”—were made.

Dependent Data

This is a scenario where you are issuing a query that returns only working data—that is, the result is only needed so you know what the next question to ask is (the first result is not really the end data). Imagine a process where you want to report on all the customers that meet a certain profile, but you want a separate report for each. You might issue a query that generates the original list and then a separate request for each customer in that list.

There are a couple of different ways of dealing with this:

- ❑ Issue a single query that groups the data, then monitor for the change in the group value to indicate the start of a new report. I’m not a fan of this one since it requires process-aware code that I find often breaks as the process changes a bit. Keep this solution in mind, but it’s not the solution for every situation.
- ❑ Utilize a stored procedure or include a full batch in your request such that it issues all the necessary queries and returns multiple data sets in one call (then just handle the data sets individually).

Bandwidth

This is all about managing the size of total data passed back and forth across the wire. Imagine trying to move megabytes of data across the wire every time a user goes to a new screen or opens up a tree list. Can we say SLOW? Just to add insult to injury, imagine what all that data is doing to everyone else on the network between that client and the server. Okay, just to expand things even a bit further, imagine that several clients on the network are all doing that at the same time! Yuck!

A common cause for bad decisions on this one is the roundtrip issue we just discussed. In an effort to minimize roundtrips, or simply because the developer doesn’t like SQL and is more comfortable in his or her own language, the code will fetch a very large block of data and then filter and manipulate it on the client side.

Okay, I’m going to stop right here and kindly ask you to literally smack yourself if you’re one of the aforementioned developers that does this just because it’s more comfortable to work with it in your client language. I see this one all too frequently, and, at the risk of being politically incorrect, doing things this way tells me that you are either too lazy to learn to do it right or you have had a brain lapse of monumental proportions (I’m going to assume the former). Harsh? Perhaps, but this is really bad stuff!

Let's put it this way: As in all things, "balance." Yes, you want to avoid roundtrips, but do not use that as an excuse to take SQL Server kind of work away from your SQL Server. SQL is all about filtering and sorting data—let it do it! Just do so efficiently.

Connectivity Examples

What we're going to do in this section is provide some hyper (and I do mean in the extreme) examples of how to get connected. For each language, we'll show a couple of examples—one for each of two different kinds of operations (fetching a simple data set and executing a query that does not return a data set).

Connecting in C#

C# is somewhat hard to ignore. When I wrote the last version of this book, C# was really just an up and coming thing. It is a fairly clean language and is relatively easy to learn much like VB, but it has the extra benefit of providing some C-based concepts and also being much closer in syntax (making it easier to transition between the two).

Returning a Data Set

```
using System;
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        //Create some base strings so you can look at these
        // separately from the commands they run in

        // Integrated Security - next line should be uncommented to use
        string strConnect = "Data Source=(local);Initial Catalog=master;Integrated
Security=SSPI";
        // SQL Security - next line should be uncommented to use
        //string strConnect = "Data Source=(local);Initial Catalog=master;User
        Id=sa;Password=MyPass";

        string strCommand = "SELECT Name, database_id as ID FROM sys.databases";

        SqlDataReader rsMyRS = null;

        SqlConnection cnMyConn = new SqlConnection(strConnect);

        try
        {
            // "Open" the connection (this is the first time it actually
            // contacts the database server)
            cnMyConn.Open();

            // Create the command object now
            SqlCommand sqlMyCommand = new SqlCommand(strCommand, cnMyConn);

            // Create the result set
```

Appendix B

```
        rsMyRS = sqlMyCommand.ExecuteReader();

        //Output what we got
        while (rsMyRS.Read())
        {
            // Write out the first (ordinal numbers here)
            // column. We can also refer to the column by name
            Console.WriteLine(rsMyRS["Name"]);
        }
        Console.WriteLine();
        Console.WriteLine("Press any key to continue...");
        Console.ReadKey();
    }
    finally
    {
        // Clean up
        if (rsMyRS != null)
        {
            rsMyRS.Close();
        }

        if (cnMyConn != null)
        {
            cnMyConn.Close();
        }
    }
}
```

Executing Commands with No Data Set

```
using System;
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        //Create some base strings so you can look at these
        // separately from the commands they run in

        // Integrated Security - next line should be uncommented to use
        string strConnect = "Data Source=(local);Initial Catalog=master;Integrated
Security=SSPI";
        // SQL Security - next line should be uncommented to use
        //string strConnect = "Data Source=(local);Initial Catalog=master;User
        Id=sa;Password=MyPass";

        string strCommand = "CREATE TABLE Foo(Column1      INT NOT NULL PRIMARY KEY)";
        string strCommand2 = "DROP TABLE Foo";

        SqlConnection cnMyConn = new SqlConnection(strConnect);

        try
        {
```

```
// "Open" the connection (this is the first time it actually
// contacts the database server)
cnMyConn.Open();

// Create the command object now
SqlCommand sqlMyCommand = new SqlCommand(strCommand, cnMyConn);

// Execute the command
sqlMyCommand.ExecuteNonQuery();

Console.WriteLine("Table Created");
Console.WriteLine("Press enter to continue (you can go check to make
sure that it's there first) ");
Console.ReadLine();

// Change the command
sqlMyCommand.CommandText = strCommand2;

sqlMyCommand.ExecuteNonQuery();

Console.WriteLine("It's gone");

Console.WriteLine();
Console.WriteLine("Press any key to continue... ");
Console.ReadKey();
}

finally
{
    // Clean up
    if (cnMyConn != null)
    {
        cnMyConn.Close();
    }
}
}
```

Connecting in VB.NET

VB.NET continues replacing the venerable Visual Basic. While I'm increasingly favoring C# for my own efforts, VB remains the code of choice for many.

Returning a Data Set

```
Imports System
Imports System.Data
Imports System.Data.SqlClient

Module Program

    Sub Main()

        'Create some base strings so you can look at these
```

Appendix B

```
'separately from the commands they run in

' Integrated Security - next 2 lines should be uncommented to use
Dim strConnect As String = _
    "Data Source=(local);Initial Catalog=master;Integrated Security=SSPI"
' SQL Security - next 2 lines should be uncommented to use
'Dim strConnect As String = _
    "Data Source=(local);Initial Catalog=master;User Id=sa;Password=MyPass"

Dim strCommand As String = _
    "SELECT Name, database_id as ID FROM sys.databases"

Dim rsMyRS As SqlClient.SqlDataReader

Dim cnMyConn As New SqlClient.SqlConnection(strConnect)

' "Open" the connection (this is the first time it actually
' contacts the database server)
cnMyConn.Open()

' Create the command object now
Dim sqlMyCommand As New SqlClient.SqlCommand(strCommand, cnMyConn)

' Create the result set
rsMyRS = sqlMyCommand.ExecuteReader()

'Output what we got
Do While rsMyRS.Read
    ' Write out the first (ordinal numbers here)
    ' column. We can also refer to the column by name
    Console.WriteLine(rsMyRS("Name"))
Loop

Console.WriteLine()
Console.WriteLine("Press any key to continue...")
Console.ReadKey()

' Clean up
rsMyRS.Close()
cnMyConn.Close()

End Sub

End Module
```

Executing Commands with No Data Set

```
Imports System
Imports System.Data
Imports System.Data.SqlClient

Module Program

Sub Main()

    'Create some base strings so you can look at these
```

```
'separately from the commands they run in

' Integrated Security - next 2 lines should be uncommented to use
Dim strConnect As String = _
    "Data Source=(local);Initial Catalog=master;Integrated Security=SSPI"
' SQL Security - next 2 lines should be uncommented to use
'Dim strConnect As String = _
    '"Data Source=(local);Initial Catalog=master;User
Id=sa;Password=MyPass"

Dim strCommand As String = "CREATE TABLE Foo(Column1      INT NOT NULL PRIMARY
KEY)"
Dim strCommand2 As String = "DROP TABLE Foo"

Dim cnMyConn As New SqlClient.SqlConnection(strConnect)

' "Open" the connection (this is the first time it actually
' contacts the database server)
cnMyConn.Open()

' Create the command object now
Dim sqlMyCommand As New SqlClient.SqlCommand(strCommand, cnMyConn)

' Execute the command
sqlMyCommand.ExecuteNonQuery()

Console.WriteLine("Table Created")
Console.WriteLine("Press enter to continue (you can go check to make sure
that it's there first)")
Console.ReadLine()

' Change the command
sqlMyCommand.CommandText = strCommand2

sqlMyCommand.ExecuteNonQuery()

Console.WriteLine("It's gone")

Console.WriteLine()
Console.WriteLine("Press any key to continue...")
Console.ReadKey()

' Clean up
cnMyConn.Close()

End Sub

End Module
```


C

Getting Service

SQL Server is a wildly complex product. Putting it that way is somewhat at odds with the notion that it is one of the easiest Relational Database Management Systems (RDBMSs) to use. But, while using the basic functionality comes much easier than it does with most database management systems, SQL Server has all sorts of “extras,” and some of these extras are almost full products into themselves. So, what this appendix is about is outlining each of the additional services SQL Server provides, but that, for whatever reason, is not covered in its own chapter in the book.

In this appendix, we’ll take a look at:

- ❑ **Analysis Services**—This started as a robust Online Analytical Processing (OLAP) engine, but has grown into much more. Data mining—also often sold by other companies as a standalone product—is also provided within Analysis Services.
- ❑ **Notification Services**—This is an event-monitoring and notification service. You can have “subscribers” receive information automatically when events they have subscribed to occur. The service monitors events you have defined and checks them against subscriptions. If there is a match, it can notify the subscriber using one of a few different delivery methods (such as e-mail or text message).
- ❑ **Service Broker**—You can think of this as something of a replacement for the Microsoft Message Queue (MSMQ). Since the Service Broker is built into the core SQL Server, you do not have the issues that used to exist mixing two different server products. The Service Broker facilitates asynchronous communication between different processes or even different servers.

What we’ll find is that SQL Server is much more than just SQL Server. Oh sure, we’ve seen that somewhat in our looks at Reporting Services, Integration Services, and the Full-Text Search engine, but we’ll see that what SQL Server has to offer just keeps going and going.

Analysis Services

In the book proper, we talked some about OLTP- versus OLAP-oriented databases. Most of the book focused on Online Transaction Processing (OLTP) databases, because those are, far and away, the most common kind of database out there. There is, however, another, and Online Analytical Processing (OLAP) deserves its due. Indeed, OLAP continues to increase in importance, and it is no coincidence that SQL Server added support for analytical applications back in version 7.0. Since then, it added data mining support, and, with this release, completely rewrote everything from the ground up with more customer input and a lot of extra experience under their belt.

So, the question may come about why I don't have a chapter on this. The answer is that I did have a chapter in it on prior editions of this book but decided to forgo a full-blown chapter on it this time. This topic has really grown to a degree where full coverage of it is just too much for one chapter in this book.

So, instead of a larger treatment that couldn't be complete what I hope to do here is much of what I did in the past—just in a bit more compact form. What is that exactly? Well, that is to give you a taste for what can be done. From there, you can pursue the many books that are exclusive to just Analysis Services (and I'm not kidding when I say it needs and deserves its own book).

A Brief Review of OLAP vs. OLTP

OLAP databases—which serve as the backbone for most Analysis Services applications—have a much different design focus than we have for OLTP. Whereas in OLTP databases, we use notions such as normalization to eliminate repeating data to save space and increase inserted and update performance, among other things. With OLAP we want everything in one place. The smaller, but more scattered nature of OLTP databases is very helpful for insert, update, and delete performance—the problem is that it largely stinks for pure read performance.

OLAP takes a different approach—it embraces the idea of repeating and redundant data for the sake of having everything related to a given row right in one place. No joins to deal with means faster reads in most circumstances. Other OLTP “no-no’s” such as derived data are embraced for much the same reason—that which is stored right next to other things that it is used with will be available much faster than if we had to compute it or otherwise fetch it from a different location.

Let's take a look at this by examining the design differences between the AdventureWorks database we have used so often in this book and the companion AdventureWorksDW (the DW stands for Data Warehouse) database. Whereas AdventureWorks is designed with the notion of online invoices, inventory control, and other immediate access and update notions in mind, the AdventureWorksDW database is focused on reporting—nothing else. The result is that AdventureWorksDW uses far fewer tables. All the rules are normalization are broken in a big way, but all our data is in a more simplified model that is both easier to understand for an end user who might be accessing the tables for reporting and is retrieved more quickly since it is all in one place.

To see this in action, let's look at the models for the `Production.Product`-related tables. First, take a look at the series of tables we've been using from the AdventureWorks database, as shown in Figure C-1.

There are 14 tables involved in this submodel. Don't let that large number cause you to think it's a bad model (not quite the way I would have done it, but it fits the theory just fine). Instead, just realized that

it's designed to meet a different goal than our OLAP database is. So, with that in mind, let's look at the OLAP version in AdventureWorksDW as shown in Figure C-2.

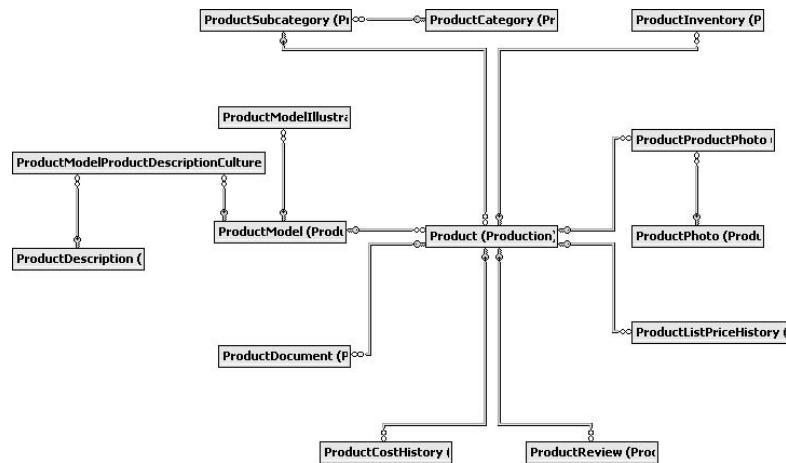


Figure C-1

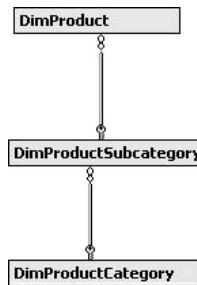


Figure C-2

Now, this has to bring about the question of “well, how did they condense it down that much?” So, let's look at the **DimProduct** table in particular by expanding the column list out (in Figure C-3).

If you compare that to the **Product** table in the AdventureWorks database, you'll quickly see that a ton of columns that had been in other tables are now in the main **Product** table. This is rather inefficient spacewise, and, were this in an OLTP setting, would risk having conflicting data. It does, however, simplify the model for people trying to write reports against it and is wildly efficient from the standpoint of reporting against the data that is in the table (no work to join things—it's all already together).

Does this mean that we never join in OLAP databases? No—indeed the most typical model for OLAP (usually referred to as a “Star” schema) assumes that we'll be joining to some data, but it attempts to minimize how many joins are involved. The idea is that the core data is defined in what is referred to as a “fact” table—this is the table that typically contains quantitative information. The fact table is supported by “Dimension” tables, which describe a breakdown within the fact table.

Appendix C

DimProduct
ProductKey
ProductAlternateKey
ProductSubcategoryKey
WeightUnitMeasureCode
SizeUnitMeasureCode
EnglishProductName
SpanishProductName
FrenchProductName
StandardCost
FinishedGoodsFlag
Color
SafetyStockLevel
ReorderPoint
ListPrice
Size
SizeRange
Weight
DaysToManufacture
ProductLine
DealerPrice
Class
Style
ModelName
LargePhoto
EnglishDescription
FrenchDescription
ChineseDescription
ArabicDescription
HebrewDescription
ThaiDescription
StartDate
EndDate
Status

Figure C-3

In the example of our DimProduct table, the “Dim” on the front of it is meant to imply that the table is a Dimension table. If you look through the list of tables in the AdventureWorksDW database, you will see that there are many tables that are prefixed “Dim” and also several that are prefixed “Fact.”

If we focus in on the FactInternetSales table and the tables directly related to it, as shown in Figure C-4, we can see the foundation of a star schema.

The basis for the “star” name is how there is a center (the FactInternetSales table) with spokes off to several other tables that are only connected to the fact table (the Dimension tables). You may also bump into what is called a “Snowflake” schema, which you can think of as a star schema that is directly attached to another star schema.

If you’re curious as to why there are three relationships between FactInternetSales and DimTime, it’s because there are three times (OrderDateKey, DueDateKey, ShipDateKey) to relate to the dimension table.

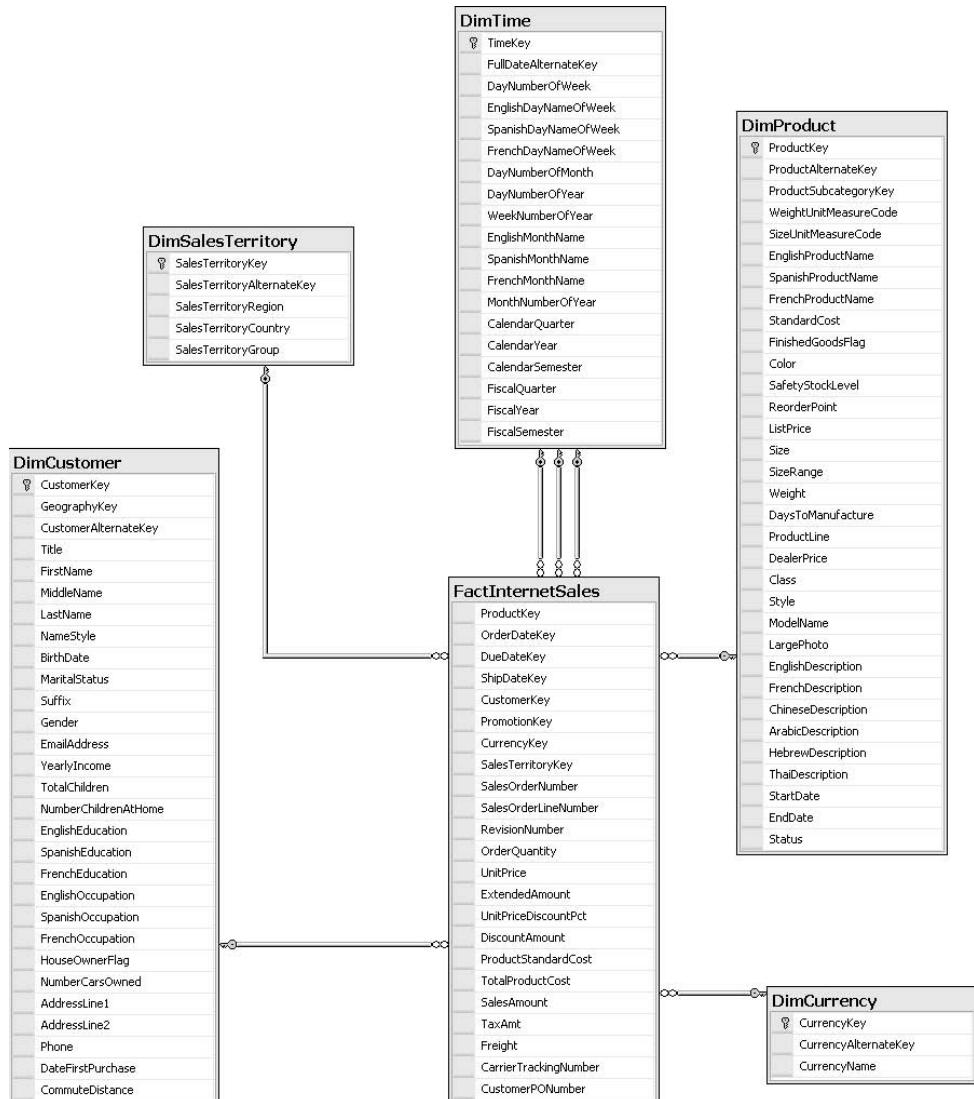


Figure C-4

Cubes

A cube is something of a multidimensional schema. It supports the notion of a “drill down” by utilizing a fact table, but analyzing it along one or more dimensions. The term “cube” uses a three-dimensional object (the cube) to imply that we are dealing with more than just the two dimensions that we have with a standard table (columns and rows). Don’t let that fool you though—a data cube is really n -dimensional and can be analyzed across more than just three, four, or five dimensions. The cube works nicely though to visualize how we are looking for the intersection of data. Figure C-5, for example, shows how we might

Appendix C

represent the intersection of the sale of Operating Systems in the Michigan store in 1998. Other nodes are available to us (such as Operating Systems in California in 1997), and we can line these up any way we choose to facilitate analysis. The key is that the relevant data is already “rolled up” in some fashion, so we can expect to get results back very quickly.

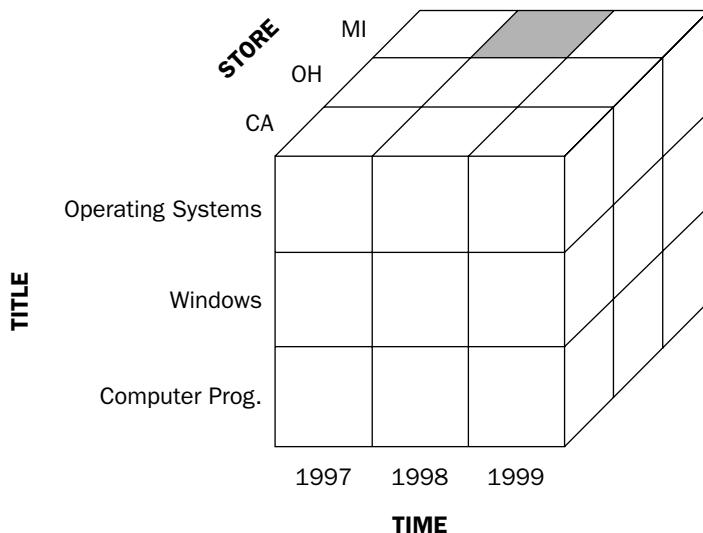


Figure C-5

If we were trying to query these numbers out of a standard table, we would need a complex `GROUP BY` statement coupled with some `SUM()`s. Aggregates are notoriously slow, and we would likely be waiting a long time for our results (assuming that there was a reasonable amount of sales data to be aggregated).

By building a cube, we define rules for Analysis Services (or other multidimensional database tool of your choice) to pre-aggregate numbers for us.

Storage Types

When a multidimensional database stores its rollup information, it uses a totally different kind of structure than we see in the typical B-Tree used by the main relational engine. B-Trees are efficient for two-dimensional data, but they have no special constructs to deal with multidimensional needs. Three different storage models are used to store multidimensional data:

- Multidimensional OLAP (MOLAP)
- Relational OLAP (ROLAP)
- A hybrid of the previous two options (HOLAP)

Each of these options provides certain benefits, depending on the size of your database and how the data will be used.

MOLAP

MOLAP is a high-performance, multidimensional data storage format. The OLAP data supporting the cubes, along with the cubes themselves, are stored on the OLAP server in multidimensional structures (OLAP structures). MOLAP gives the best query performance because it is specifically optimized for multidimensional data queries. Performance gains stem from the fact that the fact tables are compressed with this option and bitmap indexing is used.

Since MOLAP requires that all of the data be copied, MOLAP is appropriate for small to medium-sized data sets. Copying all of the data for such data sets would not require significant loading time nor would it utilize large amounts of disk space.

ROLAP

Relational OLAP storage keeps the OLAP data that feeds the cubes, along with the cube data (aggregations), in relational tables located in the relational database. In this case, a separate set of relational tables is used to store and reference aggregation data (cube data) in this OLAP system. These tables are not downloaded to the Decision Support Systems (DSS) server. The tables that hold the aggregations of the data are called *materialized views*. These tables store data aggregations as defined by the dimensions when the cube is created.

With this option, aggregation tables have fields for each dimension and measure. Each dimension column is indexed. A composite index is also created for all of the dimension fields. The balance between not creating a full copy of the data versus not having the results already calculated means ROLAP is ideal for large databases or legacy data that is infrequently queried.

HOLAP

A combination of MOLAP and ROLAP is also supported. This combination is referred to as HOLAP. With HOLAP, the OLAP data feeding the cubes is kept in its relational database tables similar to ROLAP. Aggregations of the data (cube data) are performed and stored in a multidimensional format. An advantage of this system is that HOLAP provides connectivity to large data sets in relational tables while taking advantage of the faster performance of the multidimensional aggregation storage. A disadvantage of this option is that the amount of processing between the ROLAP and MOLAP systems may affect its efficiency.

Data Warehouse Concepts

A *data warehouse* is a data store that holds the data collected during the company's conduction of business over a long period of time. The data warehouse uses the OLTP systems that collect the data from everyday activities and transactions as its source. The data warehouse concept also includes the processes that "scrub" (this is the process eliminating or corrected bad data as well as filtering down to just the data you actually want) and transform the data, making it ready for the data warehouse. It also includes the repository of summary tables and statistics, as well as the dimensional database. Finally, it also includes the tools needed by the business analysts to present and use the data. These tools include OLAP tools (such as Analysis Services), as well as data-mining and reporting tools. Figure C-6 depicts the conceptual structure and components of a data warehouse solution.

Appendix C

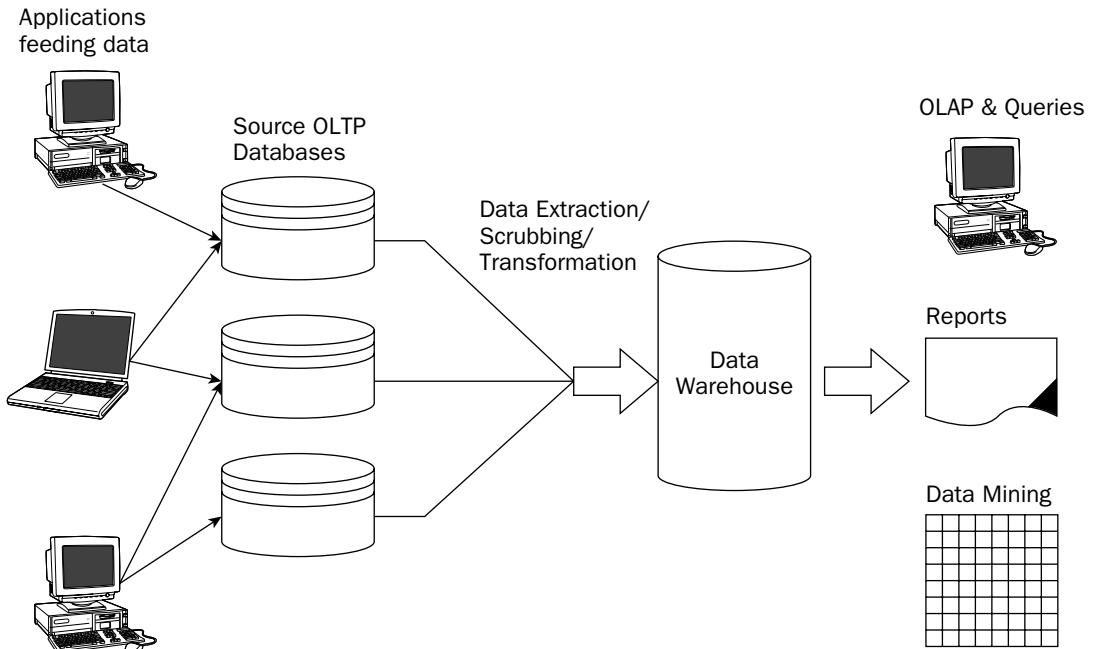


Figure C-6

Data Warehouse Characteristics

A data warehouse is usually built to support decision making and OLAP because it is designed with the following unique characteristics:

- ❑ **Consolidated and consistent data**—In a data warehouse, data is collected from different sources and consolidated and made consistent in many ways, including the use of naming conventions, measurements, physical attributes, and semantics. This is important because business analysts, accessing the data warehouse and using its data for their decision-making processes, have to use consistent standards. For example, date formats may all follow one standard, showing day, month, quarter, and year. Data should be stored in the data warehouse in a single, acceptable format. This allows for the referencing, consolidating, and cross-referencing of data from numerous heterogeneous sources, such as legacy data on mainframes, data in spreadsheets, or even data from the Internet, giving the analysts a better understanding of the business.
- ❑ **Subject-oriented data**—The data warehouse organizes key business information from OLTP sources so that it is available for business analysis. In the process, it weeds out irrelevant data that might exist in the source data store. The organization takes place based on the subject of the data, separating customer information from product information, which may have been intermingled in the source data store.
- ❑ **Historical data**—Unlike OLTP systems, the data warehouse represents historical data. In other words, when you query the data warehouse, you use data that had been collected using the OLTP system in the past. The historical data could cover a long period of time compared to the OLTP system, which contains current data that accurately describes the system for the most part.

- ❑ **Read-only data**—After data has been moved to the data warehouse, you may not be able to change it unless the data was incorrect in the first place. The data in the data warehouse cannot be updated because it represents historical data, which cannot be changed. Deletes, inserts, and updates (other than those involved in the data loading process) are not applicable in a data warehouse. The only operations that occur in a data warehouse once it has been set up are loading of additional data and querying.

Data Marts

You may find out, after building your data warehouse, that many people in your organization only access certain portions of the data in the data warehouse. For instance, the sales managers may only access data relevant to their departments. Alternatively, they may only access data for the last year. In this case, it would be inefficient to have these people query the whole data warehouse to get their reports. Instead, it would be wise to partition the data warehouse in smaller units, called *data marts*, which are based on their business needs.

In addition, some people in your organization may want to be able to access the data in the data warehouse in remote areas far from the company buildings. For instance a sales manager may want to access data about products and sales particular to his or her market area while on a sales venture. People such as this would benefit from a data mart, as they would be able to carry a section of the data warehouse on their laptop computers, allowing them to access the data they need at any time.

Of course, with data marts, the data should be kept in sync with the data warehouse at all times. This can be done in a variety of ways, such as using SQL Server Integration Services (SSIS), ActiveX scripting, or programs. Figure C-7 shows the structure of the data warehouse concept with data marts included.

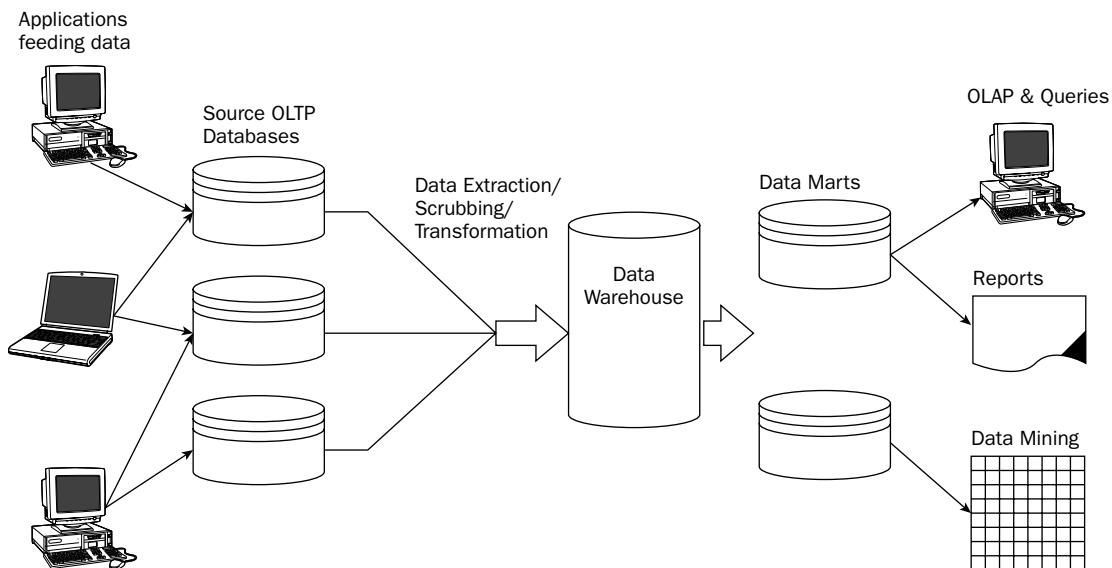


Figure C-7

Data Mining

Data mining is the concept of using advanced mathematical algorithms to analyze data for complex relationships, patterns, and tendencies. Often, this involves extracting information that might not otherwise be apparent. An example of this might be an analysis that was done many years ago that figured out that people that buy beer are much more likely to also buy cheese. When I first heard this, I realized that it was not something I would have come up with on my own, and, indeed, the people who first figured it out were rather surprised by it. But, in the end, the analysis was hard to ignore, and today you will often find small packs of cheese stored close to the beer aisle to make the most out of that behavior. You can see data mining in use in everything from predicting aircraft defects to complex marketing behaviors or complex sales forecasts.

The method of determining these relationships varies but will employ any one of a large number of built-in algorithms or even the use of your own custom algorithm. Examples of algorithm types include regression, classification, time series, association, and sequence. Within each of these varieties of algorithms, there are many individual implementations that each may provide a substantially different result from the other.

Again, the goal here is extracting data across what can be very complex relationships.

What Analysis Services Has to Offer

Analysis Services is a very powerful engine for supporting Decision Support Systems (DSS) in your enterprise. While it will typically leverage one or more data warehouses, it is not limited to just a data warehouse as a source of data. Indeed, it can build cubes and data-mining models from multiple data sources ranging from the data warehouse, to a OLTP system, to even something as complex as hitting a web service or other URL to retrieve data.

Building an Analysis Services project involves:

- Defining one or more data sources
- Defining data source views against one or more of your data sources
- Utilizing your data source views, defining which tables are facts and which are dimensions
- Establishing any hierarchies in your data (for example, that cities are within counties, which are within states/provinces, which are within countries)
- Defining the data-mining structures (if any) you wish to use
- Processing the cube (this is when the actual analysis happens)

Defining an analysis services cube can take as little as a few minutes (if the data is very simple), and as long as many weeks (if you have multiple data sources, complex relationships, and custom calculations and data mining implementations to create). Perhaps the more challenging task is to properly educate your users—who are typically not “database people”—on what, exactly, the data means (more than one bad decision has been made with good data that a user thought was something other than what it really was).

Building a Simple Cube

Perhaps the best way to get you a taste (and, truly, it will only be a taste) of what an Analysis Cube can do is to toss one together real quick.

1. Start by opening the SQL Server Business Intelligence Development Studio. Select the Business Intelligence Project node and create an Analysis Server project (I'm calling mine `AnalysisServicesExample`).

2. Create a new Data Source, but, unlike other examples in this book, pay attention to the Impersonation Information dialog—change it to utilize the service account.

3. Now create a new Data Source View from the Data Source you just created. In the Select Tables and Views dialog, select the `DimCustomer`, `DimGeography`, `DimProduct`, `DimTime`, and `FactInternetSales` tables.

Design Studio will detect relationships between the tables and plot a schema for you as shown in Figure C-8.

4. We're now ready to start putting some meat into our cube. Right-click on the Cubes node in the solution explorer, and choose New Cube....

5. Leave the defaults in place ("Build the cube using a data source," Auto build, and "Create attributes and hierarchies"), and choose Next.

6. Select the Data Source View you just created, and click Next.

7. Wait for the Detecting Fact and Dimension Tables dialog to complete, and again click Next.

8. In the Identify Fact and Dimension Tables dialog, choose the `DimTime` table in the dropdown box for "Time Dimension Table." Click Next.

9. Fill in the Select Time Periods dialog as follows:

- Year = CalendarYear
- Half Year = CalendarSemester
- Quarter = CalendarQuarter
- Month = EnglishMonthName
- Date = FullDateAlternateKey
- Day of Week = EnglishDayNameOfWeek

10. Click Next to move on to the Select Measures dialog.

11. Deselect the Promotion Key, Currency Key, Sales Territory Key, and Revision Number measures (these are just surrogate keys, and are not real measures. SQL Server has guessed them to be measures because they are numeric).

12. In the review dimensions dialog, uncheck the `DimProduct`→`Attributes`→`LargePhoto` attribute (this is a BLOB file that holds a image file—it's large, and not real useful for drill-down or descriptive purposes.)

13. Go ahead and click Finish.

Now, review the slight change in the diagram—you should see that the fact table header has been changed to yellow in color, and the dimension tables now have blue headers.

Appendix C

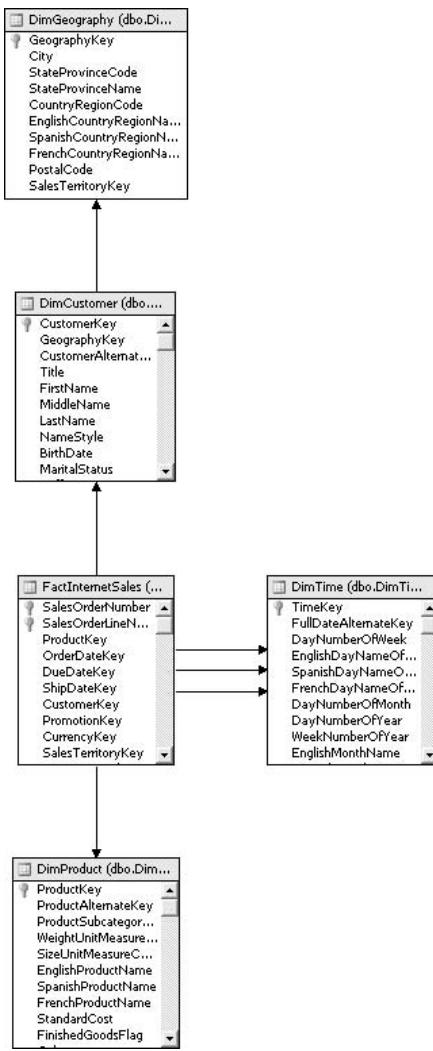


Figure C-8

Since the goal of this quick walkthrough is to give you an idea of what's possible and what's involved, I'm not going to get overly involved in the "polish" items. That said, it is worth noting that you can do things to make things more user-friendly for those who will eventually be querying your cube. For example, for each table or view, as well as the columns in the table or view, you can set a FriendlyName. You'll find that in the property area for each item you want to set it for (right-click the item and set the FriendlyName property).

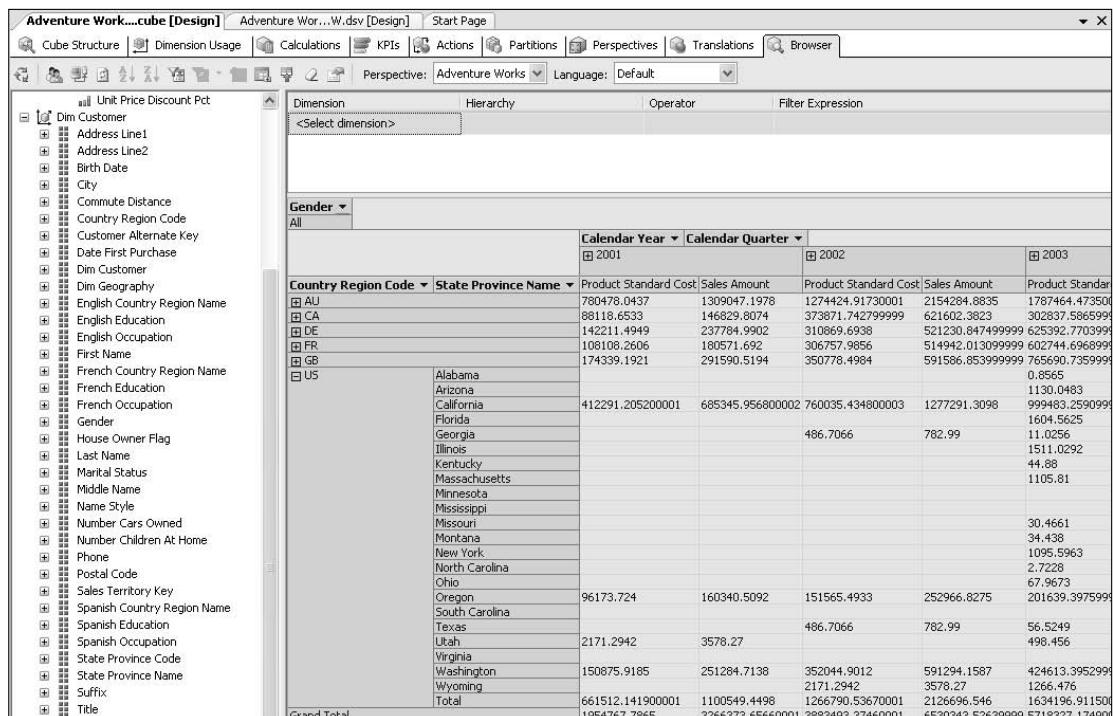
Now right-click your project in the solution explorer and choose Deploy. This moves the project to the server (make sure that your Analysis Services Service is running in your Services applet). Finally, right-click your cube and select Process. This is what actually populates the cube and makes it ready for

queries to run against it. When it is complete, drill down into the processing status information until you get to where it lists SQL queries—if you double-click that node, you can see the actual SQL that was executed to supply data for that part of the population process.

Unlike your core database engine data, the cube is updated in a batch process.
Unless the cube is purely against fixed, historical data, you will want to schedule a process to update your cube regularly (daily is typical, though more or less frequent is not unusual depending on specific need). Be sure to plan the schedule of your updates such that the many queries issued will have as small of an impact on other users as possible.

Finally, right-click the cube and select Browse. This brings up a special explorer you can use to drag and drop measures and dimensions for analysis (Well, not all that special—it's basically a pivot table like you would find in Excel. Indeed, I could connect directly from Excel if I wanted to, and you can bet that your end users will want to do just that!).

Figure C-9 shows an example where Country Region Code and State Province Name have been dragged into the row slot, the Calendar year and quarter have been dragged into the column spot, and Gender has been added as a filter. I can now expand or collapse these nodes as a chose, or even add additional nodes.



The screenshot shows the Analysis Services cube browser interface. The top navigation bar includes tabs for 'Adventure Work....cube [Design]', 'Adventure Wor...W.dsv [Design]', and 'Start Page'. Below the tabs are buttons for Cube Structure, Dimension Usage, Calculations, KPIs, Actions, Partitions, Perspectives, Translations, and Browser. The 'Perspective' dropdown is set to 'Adventure Works' and the 'Language' dropdown is set to 'Default'. On the left, a tree view of dimensions is shown, including Dim Customer, Dim Geography, and Dim Product. The main area displays a pivot table with the following structure:

Gender	Country Region Code	State Province Name	Calendar Year	Calendar Quarter	Product Standard Cost	Sales Amount	Product Standard Cost	Sales Amount	Product Standard Cost
All	All	All	2001	2002	2003				
	AU	Alabama							
	CA	Arizona							
	DE	California	412291.205200001	685345.956800002	760035.434600003	1277291.3098	999483.2590999	1604.5625	30.4661
	FR	Florida				486.7066	782.99	11.0256	34.438
	GB	Georgia						1511.0292	1095.5963
	US	Illinois						44.88	2.7228
		Kentucky						1105.81	67.9673
		Massachusetts							56.5249
		Minnesota							498.456
		Mississippi							
		Missouri							
		Montana							
		New York							
		North Carolina							
		Ohio							
		Oregon	96173.724	160340.5092	151565.4933	252966.8275	201639.397599		
		South Carolina							
		Texas				486.7066	782.99		
		Utah	2171.2942	3578.27					
		Virginia							
		Washington	150875.9185	251284.7138	352044.9012	591294.1587	424613.395299		
		Wyoming				2171.2942	3578.27		1266.476
		Total	661512.141900001	1100549.4498	1266790.536700001	2126696.546	1634196.911500		
		Grand Total	1954767.7865	3266373.656600001	3883493.374600001	6530343.526399999	5718327.174900		

Figure C-9

Appendix C

This particular example shows off only the most commonly used feature of Analysis Services—the multidimensional database. In addition to the relatively simple pivot table, Analysis Services also offers the ability to query the cube using a complex, multidimensional query language and data mining.

A Few Parting Thoughts

This has really just been something of “the once over” of Analysis Services—something of a little taste. There are tons of other ways to access and utilize Analysis Services (thus entire books dedicated to just that subject). Some of the things that go beyond the discussion here include:

- ❑ **Use of MDX**—The multidimensional query language for writing custom reports that access “tuples” (think of this as the multidimensional version of a join—perhaps even better thought of as an intersection of data). MDX has its functions and syntax that is specific to the multidimensional world, and takes you well beyond the simplicity of a pivot table.
- ❑ **Data mining**—This one really requires that you understand data warehousing and multidimensional modeling as well the more typical OLTP data that has served as the foundation of this book. Still, as you build that understanding, data mining is a critical tool in a larger-scale arsenal of researching your company’s data to make your business more effective.
- ❑ **Administrative considerations**—Just like your relational databases, your cubes need to be appropriately administered. How often should they be processed? What is the right storage method? Do they need to be backed up, or is a delay to reprocess the entire cube from the base data acceptable? What about training and security? In short, many of the same general concepts that you need to think of in administering a relational database also need to be considered for your multidimensional data.

If what we’ve discussed here seems of interest, I highly encourage you to pickup a text that addresses data warehousing and analysis services as the main topic for the book.

Notification Services

Notification Services is an entirely new functional area within SQL Server 2005. Unlike most of the other add-on services, it has little functional use without client-side code done in some .NET language. Think of Notification Services as the convergence of several technologies

The nature of Notification Services is so broad that it is largely beyond the scope of this book. As such, I will not be providing a specific installation example here, which would require an additional configuration definition followed by a special service and client install—all of these before your event was even configured. I will, however, hopefully give you a taste of what Notification Services can be used for so you know whether it is a tool that you need to look into further.

So, what is Notification Services all about? Well, at the risk of stating the obvious, it’s about notifying users or other applications when some “event” has occurred. Since the concept of custom events is supposed, the nature of the events can vary almost as wide as your imagination. Out of the box, however, SQL Server supports three event types:

- ❑ **Analysis Services events**—These are based on MDX queries. This is executed by Notification Services on a scheduled basis, and the results are compared against events that users or other applications have subscribed to.
- ❑ **T-SQL events**—Based on standard T-SQL queries. Like MDX queries, these are executed by Notification Services on a scheduled basis, and the results are compared against events that users or other applications have subscribed to.
- ❑ **File system watcher events**—This makes use of Windows' file services libraries to receive active notification when a file has been created, altered, or removed. Since Windows will proactively notify its subscribers (in this case, Notification Services becomes a subscriber), this kind of event is monitored in an active way and does not require a scheduled polling or query. Once a change is detected on disk, the file's contents can be scanned (assuming it wasn't a file deletion) using an XML schema to determine whether there is an event match or not.

Your custom events can also be written based on the idea of running on a scheduled basis or based on proactive monitoring. It could be anything from polling a Web service to querying a third-party database.

So, what kind of events would you want to track? I was not kidding when I said it was limited pretty much just by your imagination. Microsoft likes to give the example of a stock reaching a certain price (you might get this from scanning a transaction history database, or perhaps off a RSS feed) and a user being notified by e-mail or pager. Some other diverse examples might be:

- ❑ Automatic inventory control that orders a product for you when an item has gone below its emergency restocking level (an example of an application being a subscriber)
- ❑ Notifying people interested in concerns that a new concert has been scheduled that matches a genre of music they are interested in (an end-user example)
- ❑ Telling users of forum software that a new post has been made in a topic they are interested in (another user example)

Architecture Overview

Deploying a notification application requires several components working together. Notification provides several core functionality areas and something of a framework for utilizing the engines involved.

Figure C-10 shows the flow of subscription and event information into the system, and how the generator matches the events with subscriptions that are interested in those particular events to generate notification records. The distributor then sends the notification through whatever means the subscriber requested.

As someone developing for Notification Services, you utilize Notification Services APIs (found in the `Microsoft.SqlServer.NotificationServices` namespace) to identify specific events and to register user subscriptions to those events. As part of the subscription, you identify the means by which a subscriber wants to be notified of the event. Notification Services then applies the rules you have provided and generates a list of notifications that must be made based on those rules. Those notifications are then distributed by notification services via whatever means have been configured for that subscriber that match with how that subscriber wanted to be notified for that particular event (e-mail, a file drop, hitting a Web address, etc.).

Appendix C

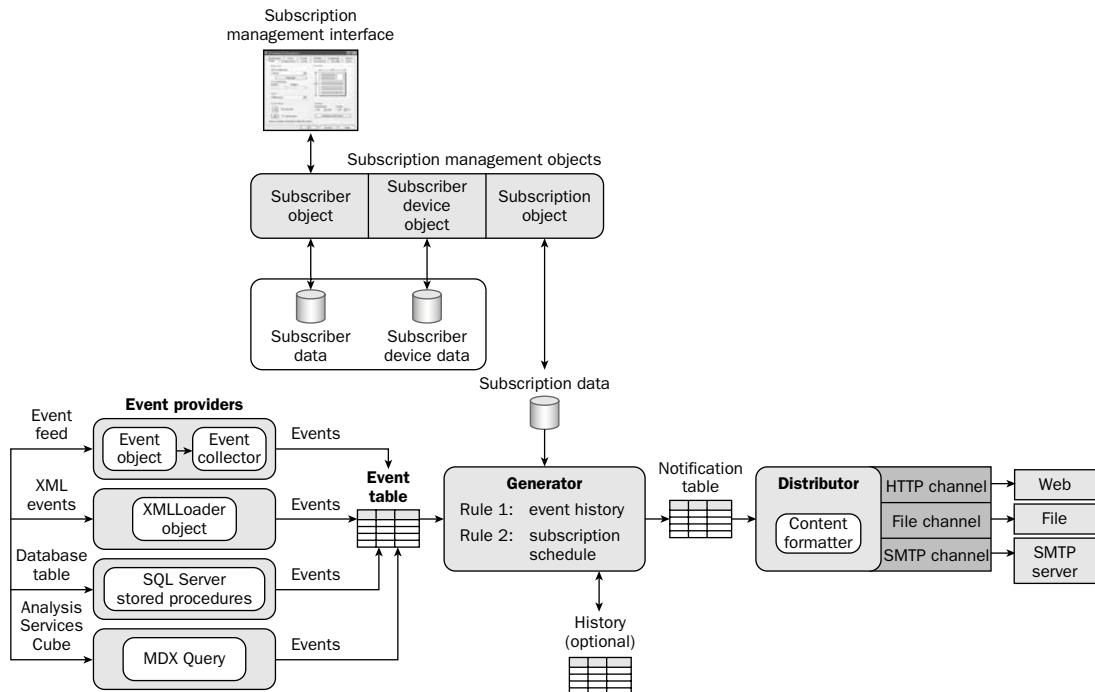


Figure C-10

Deployment

Deploying a Notification Services application involves installing not only your client application but also the Notification Services client. Since the Notification Service install is part of the SQL Server install, you will need the user to run that install prior to installing your own application. In addition, the system that will serve as the Notification Services instance must have that instance configured (this is usually done using Management Studio together with a configuration file).

A Few Parting Thoughts

Notification Services is a positively wonderful new tool/framework to have available for your applications that are already going to be utilizing SQL Server (or for applications that have the need and where there is a SQL Server already available for some other need). Indeed, in some cases, it may well justify the purchase of SQL Server for no other purpose than to support a notification application.

Don't let the complexity of the various pieces fool you—compared to writing your own diverse notification system, using the Notification Services engine provided with SQL Server makes applications with active notification requirements a relative breeze. Do, however, realize that you're going to have a steep learning curve to start out so you can properly deal with:

- ❑ The user interface or other means of registering subscriptions (utilizing the Subscription Management Interface)
- ❑ Learning how to utilize the built in event handlers
- ❑ Building, if necessary, your own event handlers utilizing the Notification Services assemblies
- ❑ Installation and deployment considerations

With planning, these are all very surmountable, and do not even have to require that much grief (again, planning is the keyword), so keep Notification Services in mind as a potential expansion of the feature set of your existing application or to meet the need of a totally new application.

Service Broker

I'm creating something of a misnomer here by calling this a different service. You see, the Service Broker is built into the core database service. That said, it really is its own thing, and thus I treat it as though it is a different service.

So then, what is it?

The Service Broker is something of an asynchronous communication service, but it offers functionality that goes beyond notification. Its job is to coordinate messages between different applications. Not all applications are always able to communicate directly. For example, the applications may run at completely different times (not all applications run as services—which is to say, they aren't “always on”). Even if your applications can communicate directly, that doesn't mean that you really want them to. Take the case of an application that is subject to burst usage—say, for example, a company that sells concert tickets. A given concert will have tickets go on sale at a specific time. If it is a popular performer, it is not at all uncommon for a show to sell out thousands of seats in the first few minutes of ticket sales—imagine the load on your system! Now, imagine being able to distribute those requests to a farm of servers that pick up the next ticket request in the queue as they are ready to process the next request.

The Service Broker allows different programs to communicate indirectly—with one application leaving a message for another and then continuing with its own work reasonably assured that the second application will indeed get the message when the time comes. The nature of the message delivery mechanism is such that the sending application can check on the status of the message at any time to know whether the message has been received yet or not. Likewise, the message recipient can either be proactively notified (for example, the service broker can start an application based on receiving a message intended for that application) or check a queue for messages at the application's leisure. You can utilize this to form very complex, distributed transactions that potentially span a long period of time (conceivably years, though I can not guess what situations that would make sense in).

In order to pull all this off, there are several key notions that need to be supported:

- ❑ **Messaging**—The message needs to have structure. While the Service Broker does support the notion of a “Default” message, most applications are going to want to have messages that have particular meaning for one or more steps of whatever process you are messaging about.
- ❑ **Contract**—This is the concept of specifying what applications are responsible (or even able) to send particular messages and which applications should be receiving those messages. It may include the notion of timed expiration of those messages.

Appendix C

- ❑ **Transactioning**—This has all the same concepts that we discuss in Chapter 12. The transaction not only should make certain that both sides of the agreement are confirmed, or neither side is, but should also be recoverable in the event of system down time (messages should not be lost!).
- ❑ **Scalability**—You need to be able to distribute message responsibility across different messaging hosts.

The Service Broker coordinates many different pieces into one cohesive application. You begin by defining a set of *messages* you will be using, and utilizing those messages to implement a *contract*. The contract defines what queue a given message goes into, and what response, if any, is expected.

Actual communication is started by an *initiator* starting what is called a *conversation*. The initiator is special only in the sense that it is the application that issued the command that “initiated” the conversation. The initiator defines both a target of the message and a contract that is to be applied to the particular conversation. A *conversation group* identifies a set of conversations that work together toward a particular goal.

A conversation may utilize multiple instances of the service. To do this, there is the concept of routing—where a given instance of the service routes a message to a queue that is managed by another service. Figure C-11 illustrates the various pieces that work together to form one service broker application.

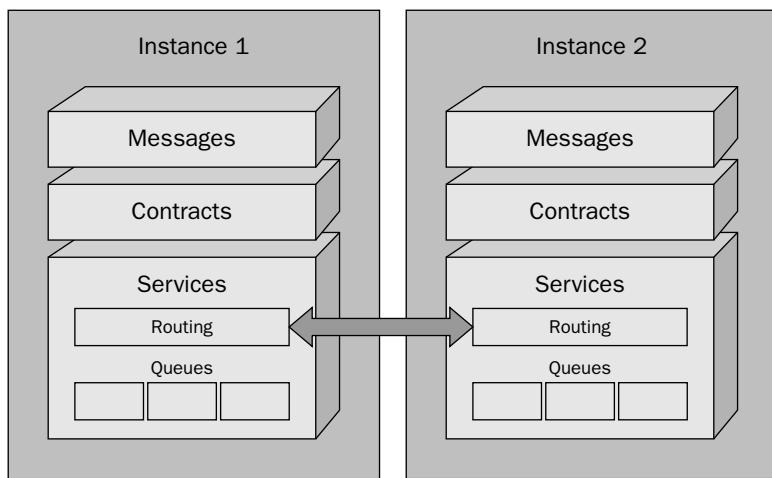


Figure C-11

Notice that messages can be delivered between different instances of SQL Server (most likely on completely different servers). This provides a foundation for the messaging part of your application to scale outward as needed. You can send messages that queue work and then have different applications pickup that workload as they are able. Likewise, you can queue work that may run as part of a batch process at a later time.

Index

Symbols and Numerics

- * (asterisk) in selection criteria, avoiding, 38
- @ (at symbol)
 - with column names using FOR XML PATH clause, 489–490, 492
 - variable scope with one versus two symbols, 293
- [] (brackets) for names, 16
- , (commas) as column placeholders for BCP import, 533
- @@CONNECTIONS function, 762
- @@CPU_BUSY function, 762
- @@CURSOR_ROWS function, 762–763
- @@DATEFIRST function, 763
- @@DBTS function, 763–764
- " (double quotes) for names, 16
- = (equals sign) in comparison operators, 47
- @@ERROR function
 - error 547 trapped by, 293–294
 - error 2714 not trapped by, 295
 - @Error variable versus, 293
 - ERROR_NUMBER() function compared to, 292
 - INSERT example using, 292–293
 - overview, 251, 764
 - saving the value from, 292–293
 - TRY/CATCH blocks versus, 295
 - using in sprocs, 293–295
- ! (exclamation mark)
 - in comparison operators, 47
 - separating groups in universal table column names, 475
- @@FETCH_STATUS function
 - overview, 251, 764
 - possible values, 424
 - simple cursor example, 424
- / (forward slash) with FOR XML PATH clause, 491–492
- > (greater than sign) in comparison operators, 47
- @@IDENTITY function
 - moving value to local variable, 252
 - overview, 251, 765
 - using in scripts, 252
- @@IDLE function, 765
- @@IO_BUSY function, 765
- @@LANGID function, 765
- @@LANGUAGE function, 765
- < (less than sign) in comparison operators, 47
- @@LOCK_TIMEOUT function, 765–766
- @@MAX_CONNECTIONS function, 766
- @@MAX_PRECISION function, 766
- @@mp:id metaproPERTY, 495
- @@mp:localname metaproPERTY, 495
- @@mp:namespacerule metaproPERTY, 495
- @@mp:parentid metaproPERTY, 495
- @@mp:parentlocalname metaproPERTY, 495
- @@mp:parentnamespacerule metaproPERTY, 495
- @@mp:parentprefix metaproPERTY, 495
- @@mp:prefix metaproPERTY, 495
- @@mp:prev metaproPERTY, 495
- @@mp:xmltext metaproPERTY, 495
- @@NETLEVEL function, 766
- @@OPTIONS function, 766–767
- @@PACK_ERRORS function, 768
- @@PACK_RECEIVED function, 767
- @@PACK_SENT function, 767
- % (percent sign)
 - as modulus operator, 271
 - as wildcard, 48
- @@PROCID function, 768
- @@REMSERVER function, 768
- @@ROWCOUNT function
 - EXEC scope example, 264
 - moving value to local variable, 253
 - overview, 251, 768
 - using in scripts, 252–253

@@SERVERNAME function

@@SERVERNAME function, 252, 768–769
@@SERVICENAME function, 769
‘(single quotes) within strings, 263
@@SPID function, 769
[] (square brackets) for names, 16
@@TEXTSIZE function, 769
@@TIMETICKS function, 769
@@TOTAL_ERRORS function, 769
@@TOTAL_READ function, 769
@@TOTAL_WRITE function, 769
@@TRANCOUNT function, 252, 770
_ (underscore)
in table names, avoiding, 83
as wildcard, 48
@@VERSION function, 770
1NF (first normal form), 157
2NF (second normal form), 157
3NF (third normal form), 155, 157
32-level limit for recursion, 306–307
547 error
CHECK constraints for monitoring, 344
from foreign key violations, 291
for inline errors in sprocs, 291, 293–294
trapped by @@ERROR, 293–294
1205 error. See deadlocks
2714 error, 295

A

abbreviations

for dateparts, 775
in table names, 83

ABS function, 777

accent sensitivity. See also case and case sensitivity

COLLATE parameter for setting, 80
enabling for full-text catalogs, 612
in SQL Server indexes, 194–195

Access (Microsoft), 85

ACID test for transactions, 359

ACOS function, 778

ADD parameter (ALTER FULLTEXT INDEX), 618

administration tasks. See also maintenance; specific tasks

archiving data, 736
backup and recovery, 722–733
index maintenance, 733–736
scheduling jobs, 700–722
usefulness for developers, 699

advanced query design

casting and converting data types, 148–150
correlated subqueries, 139–144
derived tables, 144–146
EXISTS operator, 146–148
external calls, 150–151

importance for professionals, 134
nested subqueries, 135–139
performance optimization, 151–153
procedural versus set-based thinking for, 133
subqueries, defined, 134

AdventureWorks sample database

as core sample database for this book, 4
diagram of, 7, 8
overview, 4
uspLogError sproc, 285–288
viewing columns in Management Studio, 46

AdventureWorksDW sample database, 5

AFTER clause (CREATE TRIGGER), 364

aggregate functions. See also specific functions

Accumulate method, 407, 409–410
analysis supporting, 407
creating from assemblies, 407–412
with GROUP BY clause, 53–56
Init method, 407, 409
Merge method, 407, 410
NULLs ignored by, 56
overview, 54–56
system functions, 771–773
Terminate method, 407, 410

aggregating data, subcategories for, 171–177

aliases

for columns returned by function call, 55
for computed columns, 87
Configuration Manager Aliases list, 24
in correlated subqueries, 142
sa role as dbo alias, 76
for self-referencing INNER JOINS, 40, 41
for table names, 37
using, 37

ALL keyword

ALTER INDEX statement, 216, 736
GRANT statement, 650

ALL operator, 48, 139

ALTER DATABASE statement

for increasing database size, 91
options, 92
overview, 89–92
setting cursor default to local, 429
syntax, 91
termination options for users, 92
TRUSTWORTHY parameter, 406
viewing existing database settings, 89–90

ALTER FULLTEXT CATALOG statement, 613–614

ALTER FULLTEXT INDEX statement

ADD parameter, 618
differences to other ALTER statements, 617
DROP parameter, 618
ENABLE/DISABLE parameters, 618
START FULL POPULATION parameter, 618

- START INCREMENTAL POPULATION parameter, 618
 START UPDATE POPULATION parameter, 619
 STOP parameter, 619
 syntax, 617
- ALTER FUNCTION statement, 317**
- ALTER INDEX statement**
- acting on ALL indexes, 216, 736
 - DISABLE option, 217, 735
 - for index maintenance, 227, 734–736
 - naming the index for, 216, 734
 - naming the table or view for, 216, 734
 - overview, 215, 734
 - REBUILD option, 216–217, 425–427, 734–735
 - REORGANIZE option, 217, 735–736
 - syntax, 215–216, 734
 - TableCursor for rebuilding all indexes in database, 423–427
- ALTER LOGIN statement, 642–643**
- ALTER PROC statement**
- accepting a parameter, 284–285
 - CREATE PROCEDURE compared to, 283
 - overview, 283
 - with RETURN statement, 290
 - for spEmployee sproc, 284–285
- ALTER statement, 89. See also specific forms**
- ALTER TABLE statement**
- adding CHECK constraints, 119
 - adding DEFAULT constraints, 121
 - adding foreign key, 110
 - adding primary key, 108
 - adding UNIQUE constraints, 118
 - batch for establishing precedence, 258
 - enabling/disable triggers, 378
 - example using, 94
 - NOCHECK option for constraints, 124–126
 - overview, 92–94
 - for self-referencing foreign keys, 111
 - syntax, 92–93
 - viewing existing table settings, 93
- ALTER TRIGGER statement, 363**
- ALTER VIEW statement**
- CREATE VIEW compared to, 237
 - permissions retained by, 237
 - for rebuilding views to included added columns, 94
 - WITH ENCRYPTION option, 240
 - WITH SCHEMABINDING option, 242–243
- ALTER XML SCHEMA COLLECTION statement, 462–463**
- Analysis Services**
- administrative considerations, 838
 - building a project, process of, 834
 - Configuration Manager for service management, 19
 - cubes, 829–830, 835–837
 - data mining, 834, 838
 - data warehouses, 831–833
 - described, 825
 - MDX with, 838
 - OLAP versus OLTP, 826–829
 - storage types, 830–831
 - uses for, 834
- Analysis Services Tasks (SSIS), 549**
- AND Boolean operator**
- with FTS, 629
 - overview, 48
- ANSI SQL 92 standard**
- CAST versus CONVERT and compliance with, 149
 - portability insured by, 36
 - “sys” functions not compliant with, 37
 - T-SQL entry-level compliance, 27, 36
- ANSI_NULLS option**
- not allowed for indexed views, 242
 - setting to OFF (recommended), 266
- ANY operator, 48, 139**
- application roles**
- adding permissions to, 662
 - creating, 662
 - described, 656, 661
 - dropping, 663–664
 - other roles versus, 656, 661
 - process of, 661–662
 - uses for, 661
 - using, 662–663
- applications, partition-aware, 181**
- APP_NAME function, 804**
- archiving data, 736**
- AS DEFAULT parameter (CREATE FULLTEXT CATALOG), 612**
- AS keyword**
- CREATE TRIGGER statement, 367
 - GRANT statement, 650
- ASC sort order options (CREATE INDEX), 209–210**
- ASCII function, 799**
- ASIN function, 778**
- assemblies. See .NET assemblies**
- asterisk (*) in selection criteria, avoiding, 38**
- asymmetric keys**
- certificates versus, 667
 - CREATE LOGIN...FROM option, 642
 - CREATE USER option, 648
 - defined, 642
 - limiting access to, 666
 - overview, 666
- at symbol (@)**
- with column names using FOR XML PATH clause, 489–490, 492
 - variable scope with one versus two symbols, 293
- ATAN function, 778**
- ATN2 function, 778**

atomicity

defined, 157
first normal form for guaranteeing, 157
of transactions, 329–330, 359

attributes

of documents, 171
in entity boxes, 161

audit trails

defined, 362
triggers for, 362

authentication

SQL Server Authentication, 26, 27, 639–646
Windows authentication, 26–27, 639, 646–647

AUTHORIZATION parameter

CREATE ASSEMBLY statement, 398
CREATE FULLTEXT CATALOG statement, 612

AUTO option (FOR XML clause), 470, 473–474

autonomy in replication

conflict management trade-off with, 570
defined, 566
merge replication, 574
overview, 566–567
snapshot replication, 571

autonomy of batches, 255

AVG function, 54, 771

B

backing up. *See also* recovery

backup and restore for creating database copies, 302
backup set for, 724–725
creating a device using T-SQL, 725
database using SMO, 750–751
DCM information for, 193
destination for, 725
differential backup, 724
early and often, 722
full backup, 724
including mechanism in applications, 722
Management Studio for, 722–725
more important than RAID for data safety, 683
scheduling, 725
transaction log, 724, 728
T-SQL for, 726–728

BACKUP DATABASE statement

example using, 727
parameters, 726–727
syntax, 726
user rights for, 654–655

BACKUP LOG statement, 654–655

Balanced Trees. *See* B-Trees

bandwidth, 818–819

batches. *See also* scripts

autonomous nature of, 255
defined, 247, 253

establishing precedence with, 256–258

purpose of, 256
runtime errors in, 253, 255
sent to server separately, 254–255
separating scripts into, 253–254
statements requiring their own batch, 256
syntax (parse-time) errors in, 253, 255

BCM (Bulk Changed Map), 193

BCP (Bulk Copy Program)

advantages of, 525–526
case-sensitivity of switches, 526
command for importing data to remote system, 533
commands for importing with format file, 536, 537
commas in source file as column placeholders, 533
data import example, 531–533
for exporting bulk data, 534–535
fast versus slow mode, 534
format files for, 531, 536–540
for importing bulk data, 531–534
interactive mode, 530
logged versus nonlogged mode, 534
overview, 33, 525–526, 543–544
parallel loads with, 541
source files usable for export, 535
source files usable for import, 531
switches, 527–530
syntax, 526
TABLOCK hint with, 534, 541
transaction log size issues, 534

BCP .fmt file, 536

BEGIN TRAN statement

described, 331
example using, 332, 333–334
missing, implicit transactions with, 340–341
overview, 331
syntax, 331
@@TRANCOUNT incremented by 1 with, 252

BEGIN...END blocks

with IF...ELSE statement, 268–270
with WHILE statement, 276

Beginning SQL Server 2005 Programming (Vieira), 17, 25, 44, 329, 457

Beginning XML (Hunter), 458

BETWEEN operator, 48

bigint data type

for identity columns, 84
overview, 11

binary data type, 13

binary order for indexes, 194

binding

defaults, 128
rules, 127

bit data type, 11

-
- BLOBS (Binary Large Objects)**
backward compatibility issues for, 168
data types for, 168
Full-Text Search against, 170
pages for, 192
performance issues for, 168, 170
row size limit extended by, 194
using file storage instead of, 168–170
- blocking, 177. See also locking**
- BOL (Books Online), 18–19**
- Boolean operators**
AND, 48, 629
with FTS, 629
NOT, 48, 629
OR, 48, 629
overview, 48
in searched CASE statements, 270–271, 272–275
table summarizing, 48
unknown results returned as FALSE, 268
- bound connections, for avoiding deadlocks, 358**
- Boyce-Codd normal form, 158**
- brackets ([]) for names, 16**
- BREAK statement**
controversy about, 276
with WHILE statement, 276
- B-Trees (Balanced Trees)**
defined, 195
leaf-level nodes, 196, 199, 200, 201
navigation by clustered indexes, 201–203
non-leaf level nodes, 196–197, 200
page splits with, 197–199
root node, 195–196, 198
as self-balancing, 197
- BU (bulk update) locks, 349**
- Bulk Changed Map (BCM), 193**
- Bulk Copy Program. See BCP**
- BULK INSERT command, 541–542, 657**
- Bulk Insert Task (SSIS), 549, 558–559**
- bulkadmin role, 657**
- Bulk-Logged recovery model, 728**
- Business Intelligence Studio**
Connection Manager, 506–507
Data Source Wizard, 505, 507
deploying report models, 512
Management Studio versus, 504
opening SSIS from, 546
overview, 32
Report Model Wizard, 509–511
Report Server Projects, 517–523
Report Wizard, 518–519
setting up Visual Studio for, 505
starting a Report Model Project, 504
starting a Report Server Project, 517
- C**
- calling**
aliases for columns returned by functions, 55
EXEC command, concatenating strings before, 262, 265
external calls, 150–151
GetFiles method, 405
recursion, 305–307
sprocs, assembly-based, 399–400
sprocs, EXEC keyword for, 288, 399
sprocs using OUTPUT keyword, 287
sprocs using WITH RECOMPILE option, 304
sprocs within sprocs, 301
top-level function for ExampleTVF assembly, 404
TRY/CATCH example with uspLogError, 286–288
- Callstack window (Debugger)**
overview, 322
using, 323–327
- cardinality, 162**
- cascading**
defined, 112
deletes, 112–116
ON clause of foreign key for, 112–113
SET NULL and SET DEFAULT, 112, 116
updates, 112
- case and case sensitivity. See also accent sensitivity**
BCP switches, 526
COLLATE parameter for setting, 80
index collation, 194–195
SQLCMD flags, 259
table naming standards, 83
- CASE statement**
overview, 804–805
searched, 270–271, 272–275, 805
similar statements in other languages, 266, 270
simple, 270, 271–272, 805
syntax with Boolean expression (searched CASE), 270–271
syntax with input expression (simple CASE), 270
for trigger event types, 413–414
using inline with SELECT, 271–275
- CAST function**
CONVERT versus, 148–149
converting dates, 150
converting number to string, 149
in EXEC procedures, 265
overview, 805
syntax, 149
- catalogs for FTS. See full-text catalogs**
- CEILING function, 778**
- certificates**
asymmetric keys versus, 667
certificate authority (CA) with SQL Server, 667
CREATE LOGIN...FROM option, 642

certificates (continued)

certificates (continued)

CREATE USER option, 648
described, 642
limiting access to, 666

char data type, 12

CHAR function, 799

CHARINDEX function, 799–800

CHECK constraints

- adding to existing table, 119
- adding with diagramming tools, 187
- applying assembly-based UDF as, 403
- BCP with, 531
- as column constraint, 88
- criteria examples for, 118
- deleting with diagramming tools, 187
- disabling on replication subscribers, 588
- as domain constraints, 103
- editing with diagramming tools, 187
- in logical model, 167
- monitoring for error 547, 344
- naming, 5
- overview, 117–118
- preventing negative numbers, 13
- for preventing non-repeatable reads, 344
- rules versus, 126
- specifying in CREATE TABLE statement, 87
- system-generated names for, avoiding, 5
- as table constraint, 88
- triggers versus, 362, 367–371
- violating, example of, 119

checkpoints

- at change of database options, 338
- Checkpoint on Recovery option, 337–338
- circumstances issuing, 337
- defined, 5, 336
- failure and recovery, 339–340
- issuing manually with CHECKPOINT command, 337
- at normal server shutdown, 338
- Truncate On Checkpoint option, 338
- when recovery interval option is exceeded, 339

client statistics, 692

client-side cursors

- advantages of, 435
- views supporting, 241

client-side versus server-side processing, 672–673

closing

- connections, 816–817
- cursors, 424, 430–431

CLR (Common Language Runtime)

- for external calls, 151
- file storage implementation using, 170
- required for .NET assemblies, 394
- T-SQL compliance with, 35

cluster key, 201

clustered indexes

- advantages of, 220
- changing columns for, 219–221
- choosing a column for, 220, 221
- defined, 6
- disadvantages of, 220–221
- for indexed views, 243
- indicated by sp_helpconstraint, 109
- leaf level as actual data, 201
- navigating the B-Tree, 201–203
- one per table allowed, 6
- overview, 201–203
- page split exceptions with, 193, 220–221
- performance benefits of, 203
- primary key as default for, 219, 220
- for ranged queries, 220–221
- selectivity within, 218–219
- as unique, 201

clustered tables

- cluster key for, 201
- defined, 201
- non-clustered indexes on, 205–208

COALESCE function, 805

COLLATE parameter

- CREATE DATABASE statement, 80
- CREATE TABLE statement, 86

COLLATIONPROPERTY function, 805–806

COL_LENGTH function, 782

COL_NAME function, 782–783

COLUMNPROPERTY function, 783

columns. See also identity columns

- available for ORDER BY clause, 50
- changing for clustered indexes, 219–221
- choosing for clustered indexes, 220, 221
- COLUMNS_UPDATED() function for checking trigger outcomes, 386–388

- computed, 87
- constraints, specifying in CREATE TABLE statement, 87

- defining as XML data type, 458–459
- as domain data, 5

- editable by cursors, specifying, 452
- listing explicitly with INSERT statement, 59
- maximum per row, 193

- moving using Management Studio, 99
- moving using T-SQL, problems with, 94

- order in indexes, 222
- returned by function call, name lacking for, 55
- UPDATE() function for checking trigger outcomes, 386

COLUMNS_UPDATED() function, 386–388

command object, 816

command-line console. See SQLCMD

commas (,) as column placeholders for BCP import, 533

- COMMIT TRAN statement**
 example using, 333, 335
 overview, 331
 syntax, 331
`@@TRANCOUNT` decremented by 1 with, 252
- Common Language Runtime.** See **CLR**
- comparison operators**
`ALL`, 48, 139
`ANY`, 48, 139
`BETWEEN`, 48
`IN`, 48
`LIKE`, 48, 607
`SOME`, 48, 139
 standard, 47, 138, 139, 266
 tests against NULLS, 138, 266
- compiling assemblies**
`ComplexNumber`, 417
`ExampleAggregate`, 410
`ExampleProc`, 397–398
`ExampleTrigger`, 414
`ExampleTVF`, 406
`ExampleUDF`, 402
 overview, 394
- ComplexNumber assembly, 417–418**
- computed columns, 87, 531**
- concatenation before calling EXEC, 262, 265**
- concurrency.** See also **isolation levels; locking**
 cursor options for, 447–450
 defined, 341
 not an issue between triggers and process firing, 389
 in OLAP environment, 342
 in OLTP environment, 342
 trade-off with consistency, 355
- Configuration Manager**
 Aliases list, 24
 client configuration, 23–24
 main areas of, 19
`NetLibs` configuration, 20–23
 Network Configuration area, 19–24
 protocols in, 21
 Service Management area, 19
- Configure Distribution Wizard, 589–592, 593**
- Connection Manager, 506–507**
- connections**
 adding for Debugger, 318–319
 bandwidth decisions, 818–819
 bound, for avoiding deadlocks, 358
`C#` example, 819–821
 closing properly, 816–817
 command object, defined, 816
 connection context of `EXEC`, 262
 connection object, defined, 815
 data set, defined, 816
 dependent data issues, 818
 hierarchy issues, 817–818
 managing, 816–817
 MARS for, 817
 for .NET assemblies, 396
 obtaining context for trigger, 412–413
 performance considerations, 816–819
 pooled, 816, 817
 for replication, 568, 570, 581
 for report models, 506–507
 settings, caveats for, 817
 sharing within processes, 816
 for SMO, 745
 for SSIS packages, 556–557, 559
 VB.NET example, 821–823
- @@CONNECTIONS function, 762**
- consistency**
 naming standards for, 84, 105
 replication concerns, 567–568
 for schema names, 74
`SERIALIZABLE` isolation level for, 355
 trade-off with concurrency, 355
 of transactions, 359
- constraints.** See also **specific kinds**
`BCP` with, 531
 defined, 6, 87, 101
 disabling, reasons for, 121
 disabling temporarily, 124–126
 disabling, to ignore existing bad data, 122–124
 domain, 103
 enforcing for XML, 458, 469
 entity, 103–104
 implied indexes created with, 215
 key constraints, 105–118
 in logical model, 167
 as metadata, 10
 mismatched for `BCP` import, format files for, 531, 536–540
 with mutually exclusive conditions, avoiding, 121
 naming, 4–5
`NOCHECK` option, 124–126
 other data integrity methods compared to, 129–131
 referential integrity, 104
 rules versus, 10
 specifying in `CREATE TABLE` statement, 87–88
`sp_helpconstraint` for viewing information on, 109
 system-generated names for, avoiding, 4–5
 types of, 102–104
`WITH NOCHECK` option, 122–124
- CONTAINS statement, 624–625, 628**
- CONTAINSTABLE function, 626–627, 794**
- CONTINUE statement with WHILE statement, 276**
- control-of-flow statements.** See also **specific statements**
`CASE`, 266, 270–275, 413–414, 804–805
`IF...ELSE`, 266–270
`TRY/CATCH` blocks, 277–280
`T-SQL` choices for, 266

control-of-flow statements (continued)

control-of-flow statements (continued)

WAITFOR, 277
WHILE, 275–277

conversion of cursors to other types
from FAST_FORWARD cursor, 445, 446
warning for, 446, 450–452

CONVERT function
CAST versus, 148–149
converting dates, 150
date formatting codes, 150
overview, 805
split point for two-digit to four-digit year conversion, 150
syntax, 149

converting data types

CAST function for, 148–150, 805

CONVERT function for, 148–150, 805

table summarizing, 14

correlated subqueries

aliases in, 142
importance of understanding, 139
ISNULL() function with, 143–144
nested subqueries versus, 140
overview, 140
in select list, 142–144
two-query approach versus, 140–141, 142
in WHERE clause, 140–142

COS function, 778–779

COT function, 779

COUNT function

DISTINCT clause with, 57
expressions with, 55–56
overview, 54–56, 771–772

COUNT_BIG function, 772

covered queries, 210

CPU intensive tasks

CPU time statistics, 692
increasing memory for, 683–684
I/O intensive tasks versus, 679–680
multiprocessors for, 683
query types, 680

@CPU_BUSY function, 762

CREATE ASSEMBLY statement

for aggregate functions, 410
assemblies uploaded by, 397, 398
AUTHORIZATION parameter, 398
for custom data types, 417
FROM clause, 398
for scalar UDFs, 402
for sprocs, 398
syntax, 397
for table-valued functions, 406
for triggers, 414
WITH PERMISSION_SET options, 398

CREATE authority, 75

CREATE DATABASE statement
basic syntax, 77
batch for establishing precedence, 256–258
COLLATE parameter, 80
example using, 80–82
FILEGROWTH parameter, 79
FILENAME parameter, 78
FOR ATTACH parameter, 80
full syntax, 77
LOG ON parameter, 79
MAXSIZE parameter, 79
model database as default for, 77, 78
NAME parameter, 78
ON clause, 78
overview, 77–82
PRIMARY keyword with, 78
SIZE parameter, 78
testing database existence before using, 148
TRUSTWORTHY parameter, 80
user rights for, 654–655
WITH DB CHAINING parameter, 80

CREATE DEFAULT statement, 128, 654–655

CREATE FULLTEXT CATALOG statement

AS DEFAULT parameter, 612
AUTHORIZATION parameter, 612
example using, 613
IN PATH parameter, 612
ON FILEGROUP parameter, 612
syntax, 611
WITH ACCENT_SENSITIVITY parameter, 612

CREATE FULLTEXT INDEX statement

column list with, 615
example using, 614–615
KEY INDEX parameter, 616
LANGUAGE parameter, 615
ON parameter, 616
syntax, 614
TYPE COLUMN parameter, 615–616
WITH CHANGE_TRACKING parameter, 616–617
WITH NO POPULATION parameter, 616–617

CREATE FUNCTION statement

for assembly-based scalar UDFs, 400, 402
for DayOnly() UDF, 310
for fnContactList() UDF, 311
for fnContactSearch() UDF, 311
for fnGetReports() UDF, 313–314
syntax, 308, 400

CREATE INDEX statement

ALLOW_PAGE_LOCKS option, 213
ALLOW_ROW_LOCKS option, 213
ASC/DESC sort order options, 209–210
DROP_EXISTING option, 211–214
FILLCFACTOR option, 211

IGNORE_DUP_KEY option, 211
INCLUDE columns option, 210
 for indexed views, 243
 legacy syntax, 209
 MAXDOP option, 213
 ON clause, 209, 214
 ONLINE option, 213
 PAD_INDEX option, 210
 SORT_IN_TEMPDB option, 212–213
 STATISTICS_NORECOMPUTE option, 212
 syntax, 208–209
 WITH keyword, 210

CREATE LOGIN statement
 FROM clause, 641–642
 sp_addlogin deprecated by, 640
 syntax, 640
 WITH clause, 640–641

CREATE PROCEDURE statement
 ALTER PROC compared to, 283
 for assembly-based sprocs, 398–399
 basic example, 282–283
 with RETURN statement, 288–289
 for spEmployee sproc, 282–283
 for spFactorial sproc, 305–306
 for spInsertValidatedStoreContact sproc, 293–294
 for spTestReturns sproc, 289–290
 for spTriangular sproc, 306–307
 syntax, 282
 user rights for, 654–655

CREATE RULE statement, 126–127, 654–655

CREATE statement. See also *specific forms*
 basic syntax, 77
 limited to local server, 77
 testing object existence before using, 147–148

CREATE TABLE statement
 basic syntax, 82
 for cascading actions, 112–113
 COLLATE parameter, 86
 column constraints option, 87
 computed columns option, 87
 data types specification, 84
 for DEFAULT constraints, 84, 120–121
 defining column as XML data type, 459
 example using, 88–89
 with foreign key, 109
 full syntax, 82
 IDENTITY parameter, 84–85
 model database as default for, 82
 NONCLUSTERED option, 219
 NOT FOR REPLICATION parameter, 85
 NULL/NOT NULL parameter, 86
 ON clause, 88, 112–113
 overview, 82–89

with primary key, 87, 107
 ROWGUIDCOL parameter, 85–86
 script for creating if not already existing, 267, 268, 269–270
 for self-referencing foreign keys, 111–112
 table constraints option, 87–88
 testing table existence before using, 147–148
 TEXTIMAGE_ON clause, 88
 for UNIQUE constraints, 117–118
 user rights for, 654–655

CREATE TRIGGER statement
 AFTER clause, 364
 AS keyword, 367
 DELETE parameter, 366
 FOR clause, 364
 INSERT parameter, 366
 INSTEAD OF clause, 364
 NOT FOR REPLICATION parameter, 367
 ON clause, 363
 syntax, 363
 UPDATE parameter, 366
 WITH APPEND parameter, 366
 WITH ENCRYPTION option, 363

CREATE TYPE statement, 418

CREATE USER statement, 647–648

CREATE VIEW statement
 ALTER VIEW compared to, 237
 basic syntax, 232
 extended syntax, 232
 simple example, 232
 user rights for, 654–655
 WITH SCHEMABINDING option, 241

CREATE XML SCHEMA COLLECTION statement, 461–462

CreateImportText.vbs file, 553

CROSS JOINS
 alternative (legacy) syntax for, 68
 as Cartesian product, 44
 overview, 44–45
 uses for, 45

C# connectivity example, 819–821

cubes, 829–830, 835–837

CURRENT_TIMESTAMP function, 806

CURRENT_USER function, 806

cursor data type
 declaring for sproc parameter, 284
 overview, 12

@@CURSOR_ROWS function, 762–763

ursors. See also DECLARE statement for cursors; FETCH statement; specific kinds
 altering data within, 453–456
 avoiding when possible, 133, 456, 676–677
 client-side, advantages of, 435, 673
 client-side, views supporting, 241

cursors (continued)

cursors (continued)

closing, 424, 430–431
concurrency options, 447–450
`CURSOR_STATUS` function, 773–774
deallocating, 424, 430, 431
declaring, 423
default scope, 428, 429
default type, 427
described, 421
detecting conversion of types, 450–452
dynamic, 442–445
extended syntax, 427
`FAST_FORWARD`, 445–446
`@FETCH_STATUS` variable for, 424, 425
forward-only, 432
keyset-driven, 438–441
lifespan, 422–427
`LOCAL` versus `GLOBAL` scope, 428–432
object libraries for creating, 422
opening, 423–424
performance considerations, 434
for rebuilding all indexes in database, 423–427
result set features of, 422
scrollability of, 432–434
`SELECT` statements valid for, 452
sensitivity, 435, 441, 442, 445
simple example, 424–425
specifying editable columns for, 452
sproc for creating, 428–432
static, 435–437
stating options explicitly, 429
types of, 434–435
utilizing and navigating, 424
warning for conversion to other types, 446, 450–452

`CURSOR_STATUS` function, 773–774

`CursorTest` cursor

dynamic cursor example, 443–444
global scope example, 429, 430
keyset-driven cursor example, 438–440
local scope example, 431
scrollable cursor example, 433
`SCROLL_LOCKS` example, 448–449
static cursor example, 436–437
`TYPE_WARNING` example, 451

custom data types. See user-defined data types

D

data access. See also user rights

direct, for users, 302
impersonation contexts for, 80
indexes for, 200
metadata access paths, 37
ownership for, 77
table scans for, 199

data convergence, 567

Data Flow Task (SSIS), 549

data integrity. See also constraints

checking the delta of an update, 369–371
choosing mechanisms for, 129–131
custom error messages, 371
as database responsibility, 101–102
`DRI` (declarative referential integrity), 362, 367–369
referential integrity, 104, 362
for requirements sourced from other tables, 367–369
triggers for implementing, 129, 367–371

Data Manipulation Language (DML). See also T-SQL (Transact-SQL)

basic statements, 35

triggers, 362

data marts, 833

data mining, 834, 838

Data Mining Query Task (SSIS), 549

data pages, 192

data sets

defined, 816

returning using C#, 819–820

returning using VB.NET, 821–822

Data Source View Wizard, 508

Data Source Views, 507–508

Data Source Wizard, 505, 507

Data Transformation Services (DTS), 31, 545, 546.

See also SSIS (SQL Server Integration Services)

data types

for columns in `UNIONS`, compatibility required, 70

conversions (table summarizing), 14

`CREATE TABLE` statement specification of, 84

custom, creating from assemblies, 417–419

declaring for variables, 249

in entity boxes, 161

equivalents in other programming languages, 13

for identity columns, 84

in logical model, 167

overview, 13–14

for sproc parameters, 284

table summarizing, 11–13

user-defined, 10, 417–419, 587

using rule or default, determining, 129

data warehouses, 831–833

Database Consistency Checker (DBCC)

`DBCC DBREINDEX` command, 227–228, 733

`DBCC SHOWCONTIG` command, 224–227, 228

“sys” functions versus, 224

for troubleshooting performance, 692

Database Engine Tuning Advisor, 222, 672

Database Master Key, 666

database object

attaching to current server when creating, 80

creating using Management Studio, 96–97

creating using SMO, 745–746

DECLARE statement for cursors

current as default, 76
 database-qualified names, 76
 dropping, 95, 750
 editing using Management Studio, 97
 enabling full-text indexing for, 610–611
 files, 190
 getting structure information for, 81–82
 granting user access to, 647–648
 increasing size explicitly, 91
 locking, 346
 other objects as children of, 2
 overview, 2–5
 sample databases, 3, 4–5
 specifying current database, 37
 in storage hierarchy, 189
 system databases, 3–4
 testing existence before creating, 148

database owner (dbo), 75, 76

database roles. *See also specific roles*
 fixed, 656, 658–659
 limited scope of, 658
 user-defined, 656, 659–661

DATABASEPROPERTY function, 783–784

DATABASEPROPERTYEX function, 610, 785

DATALENGTH function, 806

DATEADD function, 775

DATEDIFF function, 775

@@DATEFIRST function, 763

DATENAME function, 775

DATEPART function, 776

dates
 converting with CAST, 150
 converting with CONVERT, 150
 dateparts and abbreviations, 775
 formatting for Report Server Projects, 520–521
 system functions, 774–777

datetime data type
 overview, 12
 UDF for, 309–310

DAY function, 776

DayOnly() UDF, 310, 316–317

db_accessadmin role, 658

db_backupoperator role, 658

DBCC (Database Consistency Checker)
 DBCC DBREINDEX command, 227–228, 733
 DBCC SHOWCONTIG command, 224–227, 228
 “sys” functions versus, 224
 for troubleshooting performance, 692

DBCC SHOWCONTIG command
 for checking DBCC DBREINDEX command effects, 228
 example using, 225
 OLAP versus OLTP systems and results desired, 226–227
 output from, 225–226
 syntax, 224–225

dbcreator role, 657

db_datareader role, 658

db_datawriter role, 658

db_ddladmin role, 75, 658

db_denydatareader role, 658

db_denydatawriter role, 658

DB_ID function, 785

DB_NAME function, 785

dbo (database owner), 75, 76

db_owner role
 CREATE authority as default for, 75
 overview, 658
 ownership of objects created by, 76

db_securityadmin role, 658

@@DBTS function, 763–764

DCM (Differential Changed Map), 193

DDL triggers, 362. *See also triggers*

deadlocks
 defined, 347–348, 355
 determination by server, 356
 error 1205 for, 355
 prevented by update locks, 347–348
 rules for avoiding, 356–358
 with subcategories, 177
 victim, 355, 356

deallocating cursors, 424, 430, 431

Debugger
 Callstack window, 322
 challenges for remote debugging, 317
 icons at top, 321
 Locals window, 321–322
 Output window, 322
 setting up SQL Server for, 318
 starting, 318–321
 for triggers, 376–377, 390–392
 using, 322–327
 Watch window, 322
 yellow arrow on left, 321

decimal data type, 11

declarative referential integrity (DRI), 362, 367–369

DECLARE statement for cursors
 basic syntax, 423
 DYNAMIC option, 442–445
 extended syntax, 427
 FAST_FORWARD option, 445–446
 FOR clause, 452
 FOR UPDATE option, 452
 FORWARD_ONLY option, 432
 GLOBAL option, 428–431
 KEYSET option, 438–441
 LOCAL option, 428, 431–432
 OPTIMISTIC option, 450
 READ_ONLY option, 447
 SCROLL option, 432–434
 SCROLL_LOCKS option, 447–450

DECLARE statement for cursors (continued)

DECLARE statement for cursors (continued)

 STATIC option, 435–437
 TableCursor example, 423, 424–425, 426
 TYPE_WARNING option, 446, 450–452

DECLARE statement in scripts, 249

declaring

 cursors, 423
 data types for variables, 249
 member variables in ExampleAggregate assembly,
 408, 409
 parameters for sprocs, 284–285
 SMO, 745
 variables in scripts, 62, 249

DEFAULT constraints

 adding to existing table, 121
 cascading actions for setting, 116
 defaults compared to, 128
 as domain constraints, 103
 ignored by DELETE statement, 120
 overview, 119–121
 specifying in CREATE TABLE statement, 84, 120–121
 specifying in INSERT statement, 60
 UPDATE statement with, 120
 values for, 119

defaults. See also model database

 binding and unbinding, 128
 creating, 128
 DEFAULT constraints compared to, 128
 as domain constraints, 103
 dropping, 128
 as legacy items, 10, 126
 other data integrity methods compared to, 129–131
 overview, 128
 types of, 10
 viewing dependencies for, 129

deferred name resolution, 129

DEGREES function, 779

DELAY parameter (WAITFOR), 277

DELETE statement

 cascading deletes, 112–116
 DEFAULT constraints ignored by, 120
 overview, 64–66
 syntax, 64
 two FROM clauses for, 64
 user rights for, 649
 with views, 236–237
 WHERE clause with, 65

DELETE triggers

 checking the delta of an update, 369–371
 CREATE TRIGGER parameter for, 366
 creating from assemblies, 412–417
 firing order for, 378
 INSTEAD OF triggers, 384–385
 for setting condition flags, 373–375

DELETED tables for triggers, 366, 370

deleting. See also DELETE statement; DROP statement

 application roles, 663–664
 cascading deletes, 112–116
 custom error messages, 301
 database using SMO, 750
 existing index when creating new one of same name,
 211–212

 model database, avoiding, 4
 tables from database, 184
 tables from diagram, 184
 user-defined database roles, 661
 users from user-defined database roles, 660

de-normalization. See also normalization

 situations benefiting from, 179–180
 strategic, 674
 triggers for feeding data into reporting tables, 372–373

DENY statement, 652–653

dependencies

 connectivity considerations, 818
 with deferred name resolution, 129
 normal forms for handling, 157–158
 retained by ALTER PROC statement, 283
 viewing with sp_depends, 129

dependency chains, 114

deployment

 of Notification Services, 840
 of report models, 512
 of Report Server Project reports, 522–523
 sharing logical model with customers before, 166

derived tables, 144–146

DESC keyword

 CREATE INDEX statement, 209–210
 ORDER BY clause, 51

design. See also advanced query design

 de-normalization in, 179–180
 diagramming, 159–165, 181–187
 for file-based information, 168–170
 as foundation for all else, 281
 logical models for, 165–167
 logical versus physical, 165
 normalization issues, 155–158
 partitioning for scalability, 180–181
 performance tuning as part of, 670–671
 relationships overview, 158–159
 for reusability, 177–179
 subcategories in, 171–177
 triggers and architecture changes, 377

deterministic functions, 242, 316–317

diagramming with SQL Server tools

 adding CHECK constraints, 187
 adding existing tables to diagram, 182, 183
 adding foreign keys, 185–186
 adding new table to diagram and database, 183–184

- adding primary keys, 184, 185
AdventureWorks example, 7, 8
deleting CHECK constraints, 187
deleting foreign keys, 187
deleting primary keys, 185
dropping tables, 184
editing CHECK constraints, 187
editing foreign keys, 186–187
limitations of, 7
opening the tools, 181
saving changes to database, 184
table windows, 183
- diagrams.** *See also ERDs (entity-relationship diagrams)*
IDEFIX paradigm for, 159
IE paradigm for, 159
overview, 7
SQL Server tools for, 7, 181–187
third-party tools for, 159–160
- dictionary order for indexes, 194–195**
- DIFFERENCE function, 800**
- differential backup, 724. *See also backing up***
- Differential Changed Map (DCM), 193**
- DirectoryList() function, 404**
- DirectorySearch() function, 405**
- dirty pages, 336**
- dirty reads, 343, 347**
- DISABLE option**
ALTER FULLTEXT INDEX statement, 618
ALTER INDEX statement, 217, 735
- disabling**
actions on replication subscribers, 588
constraints, for existing data, 122–124
constraints, temporarily, 124–126
full-text indexing for database, 610, 611
indexes, 217, 735
logins with ALTER LOGIN, 643
triggers, 377–378
- diskadmin role, 657**
- DISTINCT clause**
for derived tables, 146
overview, 57–59
for UNIONs, 70
- Distributed Management Objects (DMO), 740**
- Distribution Agent, 572**
- distributor (replication)**
defined, 568
topology models, 582–583
- DLLs, compiling for assemblies, 394**
- DML (Data Manipulation Language). *See also T-SQL (Transact-SQL)***
basic statements, 35
triggers, 362
- DMO (Distributed Management Objects), 740**
- Do Case statement (Xbase). *See CASE statement***
- documents**
attributes of, 171–172
logical model for subcategory implementation, 173–175
physical model for subcategory implementation, 175–176
subcategories of, 172
- domain constraints, 103. *See also specific kinds***
- domain data. *See columns***
- domain listing, 45**
- domain tables**
defined, 104
non-identifying relationships with, 160
referential integrity for, 104
for subcategories, 174
- double quotes (“) for names, 16**
- downtime risks, 685**
- DRI (declarative referential integrity), 362, 367–369**
- driver support, 686**
- DROP parameter (ALTER FULLTEXT INDEX), 618**
- DROP statement**
for custom data types, 419
for database objects, 95
for defaults, 128
error for schema-bound tables, 112
for full-text catalogs, 614
for full-text indexes, 619
for indexes, 218, 222
for logins, 643
for rules, 128
for sprocs, 283
syntax, 95
for tables, 95
for triggers, 390
for views, 238
for XML schema collections, 463
- DROP_EXISTING option (CREATE INDEX), 211–214**
- dropping. *See DELETE statement; deleting; DROP statement***
- DTS (Data Transformation Services), 31, 545, 546. *See also SSIS (SQL Server Integration Services)***
- dual core processors, 683**
- durability of transactions, 359**
- dynamic cursors**
avoiding when possible, 442
CursorTest example, 443–445
FAST_FORWARD cursor conversion to, 446
keyset-driven cursors versus, 442, 445
non-unique index for, avoiding, 445
rebuilt with each FETCH, 442
sensitivity, 435, 442, 445
- dynamic SQL. *See EXEC command***

ELSE statement

E

ELSE statement, 267–268

enabling

CLR, 394
full-text indexing for database, 610–611
logins with ALTER LOGIN, 643
NetLibs, 21–22
triggers, 377–378

encryption

for password storage, 638, 646
for triggers, 363
for user profile storage, 638
for views, 239–241

entity boxes in ERDs, 160–161

entity constraints, 103–104. See also specific kinds

entity data. See rows

equals sign (=) in comparison operators, 47

erasing. See deleting

ERDs (entity-relationship diagrams). See also diagramming with SQL Server tools; diagrams

defined, 7
entity boxes, 160–161
examples, 164–165
for exclusive subcategories, 173
IDEFIX paradigm for, 159
IE paradigm for, 159
need for, 159
for non-exclusive subcategories, 172–173
relationship lines, 161–162
SQL Server tool for, 159
for subcategories logical model, 174–175
for subcategories physical model, 176
terminators, 162–163
third-party tools for, 159–160

@ERROR function

error 547 trapped by, 293–294
error 2714 not trapped by, 295
@Error variable versus, 293
ERROR_NUMBER() function compared to, 292
INSERT example using, 292–293
overview, 251, 764
saving the value from, 292–293
TRY/CATCH blocks versus, 295
using in sprocs, 293–295

error handling

@ERROR for, 292–295
manually raising errors, 296–299
for sprocs, 290–301
TRY/CATCH blocks for, 277–280, 291, 295

error handling for sprocs

common types of errors, 290
custom error messages, 299–301
@ERROR for, 292–295
inline errors, 291

manually raising errors, 296–299

TRY/CATCH blocks for, 291, 295

error levels

ERROR_NUMBER() function for, 279
ERROR_SEVERITY() function for, 279
returned by RAISERROR command, 296–297
table summarizing, 278, 297

error messages. See also error handling

for batches, 255
custom, for sprocs, 299–301
custom, triggers for, 371
for DROP statement with schema-bound tables, 112
error 547, 291, 293–294
error 1205 (deadlock), 355
error 2714, 295
functions for error condition retrieval, 279
for GO command in pass-through queries, 255
passed through by data access object models, 291
system, viewing all, 279–280

@Error variable, 293

ERROR_LINE() function, 279

ERROR_MESSAGE() function, 279

ERROR_NUMBER() function

@@ERROR compared to, 292
overview, 279
testing for object existence with, 287

ERROR_PROCEDURE() function, 279

ERROR_SEVERITY() function, 279

ERROR_STATE() function, 279

escalation of locking, 346

ETL (Extract, Transform, and Load), 546. See also SSIS (SQL Server Integration Services)

Evaluate statement (COBOL). See CASE statement

ExampleAggregate assembly

Accumulate method, 407, 409–410
changing class name to Product, 408, 409
creating aggregate function from, 411
creating Visual Studio project for, 408
declaring member variables, 408, 409
fContainsNull variable, 409
Init method, 407, 409
initializing member variables, 409
Merge method, 407, 410
support methods for calls, 407
template for, 408
Terminate method, 407, 410
testing the function, 411–412
uploading to SQL Server, 410

ExampleProc assembly

adding a sproc to the assembly project, 395
command object for, 396
creating spCLRExample sproc from, 398–399
creating Visual Studio project for, 395
database connection for, 396
ExampleSP method for, 396

- executing the command with `.Pipe` object, 397
 populating the output variable, 397
 setting references, 395
`StoredProcedures` class for, 396
 uploading to SQL Server, 397–398
 Visual Studio directive for, 396
- ExampleSP method, 396**
- ExampleTrigger assembly**
 CASE statement for event types, 413–414
 creating class for, 413
 creating `trgExampleTrigger` trigger from, 415
 creating Visual Studio project for, 412
 including all event types, 413
 obtaining context for, 412–413
 template for, 412–413
 uploading to SQL Server, 414
- ExampleTVF assembly**
 creating `fExampleTVF` function from, 406
 creating Visual Studio project for, 404
`EXTERNAL_ACCESS` permission set for, 406
`FillRow()` function, 405
 function to enumerate directory list, 405
`GetFiles` method call, 405
 populating the array, 405
 setting references, 404
 top-level function call, 404
 uploading to SQL Server, 406
- ExampleUDF assembly**
 code for validating e-mail fields in tables, 401–402
 creating `fCLRExample` UDF from, 402
 template for, 400–401
 uploading to SQL Server, 402
- exclamation mark (!)**
 in comparison operators, 47
 separating groups in universal table column names, 475
- exclusive locks**
 compatibility with other lock modes, 349
 intent exclusive, 348
 overview, 347
- EXEC command**
 AdventureWorks example, 260–262
 capturing return values for sprocs, 289
 concatenation performed before calling, 262, 265
 connection context of, 262
 Management Console for running, 260
 not allowed in UDFs, 262, 265
 restrictions for, 262
 scope of, 262–264
 security context for, 262, 264
 in sprocs, 264
 sprocs in, 265, 303, 304
 syntax, 260
 temp table to communicate between scopes, 264
- transaction context of, 262
 WITH RECOMPILE option for sprocs, 303, 304
- EXEC keyword for sproc calls, 288, 399**
- Execute Package Utility, 560–562**
- Execute Process Task (SSIS), 554**
- Execute SQL Task (SSIS), 555, 556**
- Execute Tasks (SSIS), 550**
- EXECUTE user rights, 649**
- executing packages (SSIS)**
 Execute Package Utility for, 560–562
 Management Studio for, 562–563
 within a program, 560
 with SQL Server Agent, 560
- .exist method (XML), 463, 468–469**
- EXISTS operator**
`IF...ELSE` examples using, 267, 268, 269–270
 indexes compared to, 200
 join-based syntax versus, 147
 nested subquery example, 146–147
 overview, 48
 performance benefits of, 199
 testing object existence before creating, 147–148
`TRUE/FALSE` return from, 146
- EXP function, 779**
- expiration of passwords, 635–636**
- EXPLICIT option (FOR XML clause)**
`cdata` directive, 487
 column naming by, 475–479
 described, 470, 474
 directives, associating information with attributes, 477–478
 directives, attribute names and, 476
 directives, overview, 478–479
 element directive, 479–480
 exclamation mark separating groups in column names, 475
 format for column names, 476
 granularity of control provided by, 474
`hide` directive, 480–482
`id` directive, 482–483
`idref` directive, 483–484
`idrefs` directive, 483, 484–486
 minimum query using, 476–477
 Parent metadata column, 475
 PATH option compared to, 486
 sufficient metadata required by, 474–475
 Tag metadata column, 475, 476
 universal table created by, 474–475
`xml` directive, 480
`xmltext` directive, 486
- exporting, BCP for, 534–535**
- expressions**
 in CASE statements, 270–271
 for computed columns, 87

expressions (continued)

expressions (continued)

in COUNT() function, 55–56
in IF...ELSE statements, 266
in SET clause for UPDATE statement, 64
extended sprocs (XPs), 304–305
extensibility, subcategories for, 176–177. See also scalability
Extensible Markup Language. See XML
extents, 190–191, 346
external calls, 150–151
EXTERNAL_ACCESS permission set
conditions required for, 406
datasource connections possible with, 400
described, 398
for ExampleProc assembly, 400
for ExampleTvf assembly, 406
Extract, Transform, and Load (ETL), 546. See also SSIS (SQL Server Integration Services)

F

factorials, computing, 306
FAST_FORWARD cursors, 445–446
fCLRexample assembly-based scalar UDF. See also ExampleUDF assembly
applying as CHECK constraint, 403
CREATE FUNCTION statement for, 400, 402
creating from ExampleUDF assembly, 402
testing, 402–403
FETCH statement
ABSOLUTE argument, 453
for altering data within a cursor, 454–455
dynamic cursor example, 443–444
dynamic cursors rebuilt each time issued, 442
FIRST argument, 433, 453
for forward-only cursors, 432
global scope example, 429
keyset-driven cursor example, 439–440
LAST argument, 433, 453
local scope example, 431
NEXT argument, 432, 453
overview, 452
PRIOR argument, 432, 453
RELATIVE argument, 453
for scrollable cursors, 432–433
SCROLL_LOCKS example, 449
simple cursor example, 424, 425
spCursorScope example, 429, 430, 431
spCursorScroll example, 433
static cursor example, 436–437
TYPE_WARNING example, 451
@@FETCH_STATUS function
overview, 251, 764
possible values, 424
simple cursor example, 424

fExampleTvf table-valued function, 406–407
fifth normal form, 158
file storage for BLOBS, 168–170
File System Tasks (SSIS), 550
FILEGROUP_ID function, 786
FILEGROUP_NAME function, 786
FILEGROUPPROPERTY function, 786
filegroups
ON keyword for, 88
overview, 7
primary, 7, 78
secondary, 7
storing full-text catalog internal structures with, 612
TEXTIMAGE_ON clause for, 88
for VLDBs, 78
FILEGROWTH parameter (CREATE DATABASE), 79
FILE_ID function, 785–786
FILE_NAME function, 786
FILENAME parameter (CREATE DATABASE), 78
FILEPROPERTY function, 787
FILLFACTOR option
CREATE INDEX statement, 211
with DBCC DBREINDEX command, 227, 228
FillRow() function, 405
filtering. See also WHERE clause
client-versus server-side processing, 673
data for replication, 570, 596
SQL Server Profiler Column Filters for, 695–696
table as source for, 49
firing order for triggers
controlling for logic reasons, 379–380
controlling for performance reasons, 380
overview, 378–379
setting with sp_settriggerorder, 378
first normal form (1NF), 157
547 error
CHECK constraints for monitoring, 344
from foreign key violations, 291
for inline errors in sprocs, 291, 293–294
trapped by @@ERROR, 293–294
fixed database roles, 656, 658–659
flags and switches
BCP 527–530
condition, triggers for setting, 373–375
OPENXML function mapping, 494
RAISERROR command, 298
SQLCMD, 259
float data type, 11
FLOOR function, 275, 779
fnContactList() UDF, 311
fnContactSearch() UDF, 311
fnGetReports() UDF, 313–314
FOR ATTACH parameter (CREATE DATABASE), 80

FOR clause

`CREATE TRIGGER` statement, 364
`DECLARE` statement for cursors, 452

For Each Container task (SSIS), 549**For Loop Container task (SSIS), 548****FOR XML clause. See also specific options**

`AUTO` option, 470, 473–474
`BINARY BASE64` option, 470
`ELEMENTS` option, 470
`EXPLICIT` option, 470, 474–487
 overview, 57
`PATH` option, 470, 488–492
`RAW` option, 470, 471–473
`ROOT` option, 471
 syntax, 469
`TYPE` option, 471
`XMLDATA` option, 470

foreign keys

adding to existing table, 110
 adding with diagramming tools, 185–186
 avoiding for reusability, 178
`BCP` enforcement of, 531
 cascading actions with, 112–116
 deleting with diagramming tools, 187
 dropping before moving columns, 94
 editing with diagramming tools, 186–187
 in entity boxes, 161
 error 547 from violating, 291, 293–294
 in identifying relationships, 160
 indexes on, 218–219
 in logical model, 167
 maximum per table, 110
 in non-identifying relationships, 160
`NULLS` in, 108, 117
 other data integrity methods compared to, 129–131
 overview, 108–116
 primer row for, 110, 111, 112
 referencing versus referenced tables, 108
 as referential integrity constraints, 104
 required versus optional values in, 117
 in self-referencing tables, 110–112
 specifying in `CREATE TABLE` statement, 87, 109
 trigger performance versus, 179

format files

for data file with fewer columns than the table, 538–539
 for data file with more columns than the table, 539
 default, 536
 example using, 540
 for mismatched field order, 539–540
 non-XML, 536–537
 performance, maximizing, 541
 situations benefiting from, 536
 for timestamp or computed columns with `BCP`, 531
 XML, 537

FORMATMESSAGE function, 806**FORMSOF () function (FTS), 631**

forward slash (/) with FOR XML PATH clause,
 491–492

forward-only cursors

client-side implementation, 673
 as default, 432
 described, 432
 efficiency of, 434

fourth normal form, 158**fragmentation of indexes**

`ALTER INDEX` statement for managing, 227, 734–736
`DBCC DBREINDEX` command for managing, 227–228,
 733

defined, 223
 identifying, 224–227
 problems from, 223–224

FREETEXT statement, 626, 628**FREETEXTTABLE function, 628, 794****FROM clause**

`CREATE ASSEMBLY` statement, 398
`CREATE LOGIN` statement, 641–642
`DELETE` statement, 64
`SELECT` statement, 36

FTP Tasks (SSIS), 550**FTS (Full-Text Search). See also full-text catalogs; full-text indexes**

advantages of, 607–608
 architecture, 608–610
 against BLOBs, data type for, 170
 against BLOBs, file storage for, 170
 Booleans in searches, 629
 changing startup to automatic, 609
`Configuration Manager` for service management, 19
`CONTAINS` statement, 624–625
`CONTAINSTABLE` statement, 626–627
 document types supported by, 607
 enabling for database, 610–611
`FORMSOF ()` function, 631
`FREETEXT` statement, 626
`FREETEXTTABLE` statement, 628
`INFLECTIONAL` keyword, 631
`ISABOUT ()` function, 630–631
`NEAR` keyword, 629–630
 noise word list, 631–632
 phrases in searches, 628
 proximity searches, 629–630
 query syntax, 624–631
`WEIGHT` keyword, 630–631

full backup, 724. See also backing up**FULL JOINS**

with `DELETE` statement, 65–66
 inclusive nature of, 42
 overview, 43–44

Full recovery model, 728

full-text catalogs

full-text catalogs

altering, 613–614
creating with new syntax, 611–613
creating with old syntax, 619–620
defined, 11
dropping, 614
removed when indexing is disabled, 611
in storage hierarchy, 194

full-text indexes

altering, 617–619
checking if enabled, 610
creating with new syntax, 614–617
creating with old syntax, 621–622
creating without populating, 617
disabling, 610, 611
dropping, 619
enabling, 610–611
populating, 609, 618–619, 622–624
removed when indexing is disabled, 611
SQL Server indexes versus, 609–610

Full-Text Search. See FTS

FULLTEXTCATALOGPROPERTY function, 787
FULLTEXTSERVICEPROPERTY function, 787–788

fully qualified names, 73

functions. See also aggregate functions; system functions; UDFs (user-defined functions); specific functions
deterministic, 242, 316–317
for error condition retrieval, 279
referenced by indexed views, 242
table-valued, creating from assemblies, 403–407

G

GAM (Global Allocation Map), 192

GETANSINULL function, 806–807

GETDATE function

DayOnly() UDF using, 310
described, 309–310
as non-deterministic, 316
overview, 776

GetFiles method, 405

GETUTCDATE function, 776

Global Allocation Map (GAM), 192

global variables (legacy system functions), 762–770.
See also specific functions

Globally Unique Identifiers. See GUIDs

GO statement

not a T-SQL command, 254, 255
not sent to server with batches, 255
in pass-through queries, avoiding, 255
separate line required for, 253, 254
for separating scripts into batches, 253–254

GRANT statement, 650–652

graphical showplan, 689–690

greater than sign (>) in comparison operators, 47

GROUP BY clause

aggregate functions with, 53–56
HAVING clause with, 56–57
overview, 52–56

GROUPING function, 772

guest account, 664

GUIDs (Globally Unique Identifiers)

defined, 86
NEWID() function for, 86
primary keys versus, 106
ROWGUIDCOL parameter for, 85–86
specifying for logins, 641

H

hardware considerations

budgeting, 678
CPU intensive tasks, 683–684
dedicated SQL Server, 679
diminishing returns, 685–686
driver support, 686
ideal system, 686
I/O intensive tasks, 680–683
I/O versus CPU intensive tasks, 679–680
lost data, 685
OLTP and OLAP on separate servers, 684
on-site versus off-site, 684
questions to ask when purchasing, 678–679
RAID, 680–683
risks of being down, 685

HAS_DATABASE function, 796

HAVING clause, 56–57

heap

defined, 201
non-clustered indexes on, 203–205
row ID (RID) for, 201, 203

hints. See optimizer hints

HOLAP (Hybrid OLAP), 831

HOLDLOCK/SERIALIZABLE optimizer hint, 349, 355

horizontal filtering or partitioning, 570, 596

HOST_ID function, 807

HOST_NAME function, 807

HTTP endpoints

creating and managing, 499–500
defined, 498
methods, 499
not supported in SQL Server Express, 498
not supported in Windows XP, 498
realm of, 497
security, 498
SOAP for, 498

Hunter, David (*Beginning XML*), 458

Hybrid OLAP (HOLAP), 831

INDEXPROPERTY function

I

ideal system, 686
IDEFIX diagrams, 159
IDENT_CURRENT function, 807
identifying relationships
 defined, 160, 162
 entity box representation of, 161
 relationship lines for, 162
IDENT_INCR function, 807
identity columns
 Access AutoNumber columns compared to, 85
 choosing in Management Studio, 97
 data types for, 84
 defined, 84
`@IDENTITY` function for, 251, 252
 primary keys versus, 85
`ROWGUIDCOL` compared to, 85–86
 specifying in `CREATE TABLE` statement, 84–85
 uses for, 85
 values automatically generated for, 60, 236
IDENTITY function, 808
@@IDENTITY function
 moving value to local variable, 252
 overview, 251, 765
 using in scripts, 252
IDENTITY parameter (`CREATE TABLE`), 84–85
IDENTITY property, disabling on replication subscribers, 588
IDENT_SEED function, 807
@@IDLE function, 765
IE (Information Engineering) diagrams, 159
IEnumerable interface, 404
IF...ELSE statement
 BEGIN...END blocks with, 268–270
 for creating table if not already existing, 267, 268, 269–270
 ELSE clause, 267–268
 expressions in, 266
 syntax, 266
 testing NULLS in, 266
IGNORE_DUP_KEY option (`CREATE INDEX`), 211
image data type, 13
image system functions, 812–813
immediate-update subscribers, 578, 580–581
impersonation contexts, 80
implicit transactions, 340–341
implied indexes created with constraints, 215
importing, BCP for, 531–534
IN operator, 48
IN PATH parameter (`CREATE FULLTEXT CATALOG`), 612
INCLUDE columns option (`CREATE INDEX`), 210
INDEX_COL function, 788

indexed views
 creating the index for, 243
 defined, 8
 example, 242–243
 index used by SQL Server, 244
 performance impacts of, 8–9
 restrictions for, 242
indexes (SQL Server). *See also full-text indexes*
 adjusting page densities, 211
 altering, 215–217
 BCP with, 531, 534, 541
 B-Tree structure for, 195–199
 clustered, 6, 109, 193, 200–203
 collation options, 194–195
 column order in, 222
 creating as implied object with constraints, 215
 creating for indexed views, 243
 creating with `CREATE INDEX` statement, 208–214
 as critical to planning and maintenance, 189
 Database Engine Tuning Advisor for, 222, 672
 defined, 194
 disabling, 217, 735
 dropping, 218, 222
 dropping and recreating, 211–212
 for dynamic cursors, 445
`EXISTS` operator compared to, 200
 on foreign keys, 218–219
 fragmentation of, 223–228, 733–736
 full-text indexes versus, 609–610
 for keyset-driven cursors, 438, 440, 441
 maintaining, 223–228
 maintenance, 733–736
 non-clustered, defined, 6
 non-clustered on a clustered table, 205–208
 non-clustered on a heap, 203–205
 not usable with `LIKE` operator, 607
 overview, 5–6, 228–229
 pages for, 192
 performance considerations, 189, 200, 219, 671–672
 questions to ask about, 229
 rebuilding, 211–212, 216–217, 734–735
 recreating after moving columns, 94
 reorganizing, 217, 735–736
 selectivity within, 218–219
 sort order for, 194–195, 209–210
 storing separately from the data, 214
`sysindexes` table for, 203
`TableCursor` for rebuilding all indexes in database, 423–427
 types of, 200
 use by SQL Server, 200
 XML, 214–215, 497
INDEXKEY_PROPERTY function, 788
INDEXPROPERTY function, 788–789

INFLECTIONAL keyword (FTS)

INFLECTIONAL keyword (FTS), 631

Information Engineering (IE) diagrams, 159

INFORMATION_SCHEMA access path, 37

inline errors, 291

inline use of UDFs, 309

inline views (derived tables), 144–146

INNER JOINS

alternative (legacy) syntax for, 67

as default type, 41–42

exclusive nature of, 40

overview, 39–42

self-referencing, 40, 41

syntax, 40

table aliases with, 40

INSERT INTO...SELECT statement, 61–62

INSERT statement

column list with, 59, 60

DEFAULT keyword, 60

dependency chains with, 114

INTO keyword, 59

syntax, 59–60

with views, 236–237

INSERT triggers

checking the delta of an update, 369–371

CREATE TRIGGER parameter for, 366

creating from assemblies, 412–417

for feeding data into de-normalized tables, 372–373

firing order for, 378

INSTEAD OF triggers, 381–383

for requirements sourced from other tables, 367–369

for setting condition flags, 373–375

INSERT user rights, 649

INSERTED tables for triggers, 366, 370

installing Microsoft Sample set, 394

INSTEAD OF triggers

for changing data with views containing joins, 236, 237

defined, 380

DELETE triggers, 384–385

described, 364

FOR | AFTER triggers versus, 364–365, 380

INSERT triggers, 381–383

tables for examples, 380–381

types of, 380

UPDATE triggers, 384

int data type

for identity columns, 84

overview, 11

Integration Services. See SSIS (SQL Server Integration Services)

integrity. See data integrity

Intelligence Studio. See Business Intelligence Studio

intent locks, 348, 349

INTO keyword (INSERT), 59

inversion keys, 105

I/O intensive tasks

CPU intensive tasks versus, 679–680

RAID for, 680–683

@IO_BUSY function, 765

ISABOUT() function (FTS), 630–631

ISDATE function, 808

ISDETERMINISTIC property, 316

IS_MEMBER function, 796–797

ISNULL function

with correlated subqueries, 143–144

overview, 808

ISNUMERIC function, 808

isolation levels

performance tuning, 675

for preventing dirty reads, 343, 353

for preventing non-repeatable reads, 344, 354–355

for preventing phantoms, 345

READ COMMITTED, 343, 353

READ UNCOMMITTED, 354

REPEATABLE READ, 344, 354–355

SERIALIZABLE, 344, 345, 355

syntax for switching, 353

types available, 353

using lowest possible, 357

isolation of transactions, 359

isql.exe (older versions). See SQLCMD

IS_SRVROLEMEMBER function, 797

J

jobs. See scheduling jobs

JOINS. See also specific kinds

across multiple servers, 76

alternative (legacy) syntax for, 66–68

ANSI syntax recommended for, 66

changing data with views containing, 236, 237

CROSS JOINS, 44–45, 68

with DELETE statement, 65–66

for derived tables, 144–146

exclusive, 40

FULL JOINS, 43–44

inclusive, 42

INNER JOINS, 39–42, 67

OUTER JOINS, 42–43, 67–68

overview, 38–39

subqueries versus, 134–135, 152–153

on tables returned by UDFs, 311, 312, 315

UNIONS versus, 69

WHERE clause with, 48–49

K

KEY INDEX parameter (CREATE FULLTEXT INDEX), 616

-
- keys. See also foreign keys; primary keys**
- importance of, 106
 - inversion, 105
 - locking, 346
 - other data integrity methods compared to, 129–131
- keyset-driven cursors**
- CursorTest example, 438–441
 - dynamic cursors versus, 442, 445
 - FAST_FORWARD cursor conversion to, 446
 - keys for, 438
 - overview, 438
 - sensitivity, 441
 - unique index required for, 438, 440, 441
- keywords in names, avoiding, 16**
- L**
- @@LANGID function, 765**
- @@LANGUAGE function, 765**
- LANGUAGE parameter (CREATE FULLTEXT INDEX), 615**
- latency**
- defined, 567
 - in merge replication, 574
 - replication concerns, 567
- .ldf extension, 78, 190. See also transaction log**
- leaf-level nodes of B-Trees**
- for clustered indexes, 201
 - defined, 196, 200
 - page splits at, 199
- LEFT function, 800**
- LEFT OUTER JOINS, 43**
- legacy items**
- BCP as, 526
 - CREATE INDEX syntax, 209
 - defaults as, 10, 126
 - JOIN syntax, 66–68
 - login creation, 643–646
 - password maintenance, 646
 - rules as, 10, 126
 - sa role as, 657
 - sp_grantdbaccess sproc, 648
 - system functions, 762–770
- LEN function, 800**
- less than sign (<) in comparison operators, 47**
- lifespan of a cursor**
- closing, 424
 - deallocating, 424
 - declaring, 423
 - full example, 424–425
 - main parts, 422–423
 - opening, 423–424
 - utilizing and navigating, 424
- LIKE operator, 48, 607**
- linked servers, 76**
- Locals window (Debugger), 321–322**
- locking**
- ALLOW settings for indexes, 213
 - bulk update locks, 349
 - compatibility of lock modes, 349
 - cursors, OPTIMISTIC option for, 450
 - cursors, READ_ONLY option for, 447
 - cursors, SCROLL_LOCKS option for, 447–450
 - determining locks using Management Studio, 352–353
 - escalation of, 346
 - exclusive locks, 347
 - granularity of, 346
 - intent locks, 348
 - lock, defined, 190, 342
 - lock manager, 342
 - lockable resources, 346
 - modes, 347–349
 - ONLINE option of CREATE INDEX for preventing, 213
 - optimizer hints, 349–352
 - performance impacts of, 346
 - problems prevented by, 342–345
 - schema locks, 348
 - shared locks, 347
 - subcategory issues for, 177
 - update locks, 347–348
- @@LOCK_TIMEOUT function, 765–766**
- LOG function, 779**
- LOG ON parameter (CREATE DATABASE), 79**
- Log Reader Agent, 578, 598**
- logging. See also transaction log**
- by BCP, 534
 - custom error messages with sp_addmessage, 300
 - errors with RAISERROR, 299
 - filename extension for log files, 78
 - LOG ON parameter for, 79
- logical errors in sprocs, 290**
- logical models**
- constraints in, 167
 - data types in, 167
 - parts of, 166–167
 - physical models versus, 165
 - purpose of, 165–166
 - for reusability, 178
 - rules in, 165–166, 167
 - sharing with customers, 166
 - structure in, 167
 - for subcategories, 173–175
- logical reads statistics, 691**
- logins**
- adding users to a database, 647–648
 - altering, 642–643
 - creating logins WMI, 645
 - creating with CREATE LOGIN, 640–642

logins (continued)

logins (continued)

creating with Management Studio, 643–644
creating with SMO, 644
creating with `sp_addlogin`, 640, 644, 645–646
creating with `sp_grantlogin`, 646
creating with SQL-DMO, 645
double password scenarios, avoiding, 638
dropping, 643
guest account, 664
for Management Studio, 26
number of tries allowed, 637
one person, one login, one password principle, 634–635
securityadmin role for managing, 657
SQL Server Authentication, 26, 27, 639–646
unlocking with `ALTER LOGIN`, 643
Windows authentication, 26–27, 639, 646–647

LOG10 function, 779–780

lookup (domain) tables
defined, 104
non-identifying relationships with, 160
referential integrity for, 104
for subcategories, 174

lost data, costs of, 685

lost updates, 345

LOWER function, 800

lowercase. See case and case sensitivity

LTRIM function, 800

M

maintenance. See also administration tasks

for index fragmentation, 223–228, 733–736
indexes as critical to, 189
for passwords, 642–643, 646
performance tuning, 674–675
rebuilding indexes, 211–212, 216–217, 734–735
updating statistics, 212

Maintenance Tasks (SSIS), 551

Management Studio

authentication type for, 26–27
backing up using, 722–725
Business Intelligence Studio versus, 504
changing maximum column length, 461
columns truncated by, 472
Configure Distribution Wizard, 589–592, 593
connection dialog, 25–27
creating databases using, 96–97
creating jobs and tasks using, 704–712
creating logins using, 643–644
creating operators using, 701–702
creating tables using, 97–99
Database Engine Tuning Advisor, 222
deleting jobs using, 721
editing databases using, 97

editing jobs using, 721
editing tables using, 99
encrypted views not displayed by, 241
getting started, 25–27
identity column selection using, 97
lock determination using, 352–353
New Publication Wizard, 593–598, 599
New Subscription Wizard, 599–603
Object Explorer, 31–32, 722
overview, 24–32
query governor, 692–693
Query window overview, 27–31
replication publication configuration using, 592–598
replication server configuration using, 588–592
replication subscriber setup using, 598–603
server type for login, 26
SQL Server for login, 26
for SSIS package execution, 562–563
uses for, 25

many-to-many relationships, 159

MARS (Multiple Active Result Sets), 817

master database, 3. See also specific tables

materialized (indexed) views
creating the index for, 243
defined, 8
example, 242–243
index used by SQL Server, 244
performance impacts of, 8–9
restrictions for, 242

mathematical functions, 777–781

MAX function, 54, 772

@MAX_CONNECTIONS function, 766

@MAX_PRECISION function, 766

MAXSIZE parameter (CREATE DATABASE), 79

.mdf extension, 78, 190

MDX (multidimensional query language), 838

memory (RAM), 683–684

Merge Agent, 575

merge replication

autonomy, 574
defined, 574
latency, 574
Merge Agent for, 575
mixing with other types, 581
overview, 574–575
planning requirements, 576
process of, 575–576
uses for, 576

Message Queue Task (SSIS), 550

metadata

access paths to, 37
constraints as, 10
defaults as, 10
defined, 5
system functions, 781–794

- Microsoft Access**, 85
Microsoft Sample set, installing, 394
MIN function, 54, 772
mobile devices, replication with, 588
model database
 altering, considerations for, 3–4
 as default for CREATE DATABASE statement, 77, 78
 as default for CREATE TABLE statement, 82
 deleting, avoiding, 4
 minimum database size determined by, 4, 78
 as template for new databases, 3–4
.modify method (XML), 463, 465–466
modulus operator
 described, 271
 searched CASE example, 273
 simple CASE examples, 271–272
MOLAP (Multidimensional OLAP), 831
money data type, 11
MONTH function, 776
moving columns
 using Management Studio, 99
 using T-SQL, problems with, 94
Gmp:id metaproPERTY, 495
Gmp:localname metaproPERTY, 495
Gmp:namespacerule metaproPERTY, 495
Gmp:parentid metaproPERTY, 495
Gmp:parentlocalname metaproPERTY, 495
Gmp:parentnamespacerule metaproPERTY, 495
Gmp:parentprefix metaproPERTY, 495
Gmp:prefix metaproPERTY, 495
Gmp:prev metaproPERTY, 495
Gmp:xmltext metaproPERTY, 495
MS Search service, 170, 608
msdb database, 4
Multidimensional OLAP (MOLAP), 831
multidimensional query language (MDX), 838
Multiple Active Result Sets (MARS), 817
multiprocessors, 683
- N**
- NAME parameter (CREATE DATABASE)**, 78
Named Pipes, 22. *See also* NetLibs (network libraries)
namespaces
 for sp_xml_preparedocument sproc, 493
 SQL Namespaces (SQL NS), 740–741
 WITH NAMESPACES() declaration, 464
 for XML .exist method, 468
 for XML .modify method, 466
 for XML .nodes method, 467
 for XML .query method, 464
 for XML .value method, 465
 XML_SCHEMA_NAMESPACE() function, 460–461
- naming. *See also* aliases**
 of columns by FOR XML EXPLICIT, 475–479
 consistency in, 84
 constraints, 4–5
 database-qualified names, 76
 deferred name resolution, 129
 fully qualified names, 73
 in indexed views, 242
 keywords in names, avoiding, 16
 linked servers, 76
 object names, 73–76
 objects having names, 15
 optional parts of names, 73
 parameters for sprocs, 284
 rules for, 15–16, 83
 schemas, 74–76
 spaces embedded in names, avoiding, 16, 83
 tables, standards for, 83–84
 underscore use in, avoiding, 83
- nchar data type**, 12
NCHAR function, 800–801
.ndf extension, 78
NEAR keyword (FTS), 629–630
nested sprocs, 301
nested subqueries
 ALL operator with, 139
 ANY operator with, 139
 correlated subqueries versus, 140
 defined, 135
 EXISTS example, 146–147
 nested SELECT to find orphaned records, 138
 returning multiple values, 137–138
 returning single value, 136–137
 SOME operator with, 139
 syntax, 135
nested triggers, 376
.NET. *See also* .NET assemblies
 extended sprocs (XPs) with, 304–305
 file storage implementation using, 170
 online help interface for BOL, 18
 programming possibilities increased by, 281
 RMO, 605–606
 T-SQL compliance with, 35
.NET assemblies. *See also* specific assemblies
 compiling, 394–397
 creating aggregate functions from, 407–412
 creating custom data types from, 417–419
 creating scalar UDFs from, 400–403
 creating sprocs from, 395–400
 creating table-valued functions from, 403–407
 creating triggers from, 412–417
 defined, 394
 enabling CLR for, 394
 EXTERNAL_ACCESS permission set for, 398, 400, 406

.NET assemblies (continued)

.NET assemblies (continued)

installing Microsoft Sample set for examples, 394
overview, 419–420
SAFE permission set for, 398, 417
UNSAFE permission set for, 394, 398
uploading to SQL Server, 397–398
windowing not supported by, 394

@*NETLEVEL* function, 766

NetLibs (network libraries). See also specific NetLibs

communication process, 20
enabling, 21–22
included with SQL Server, 20
performance impacts of, 22
security issues for, 22
support required on both client and server, 20

New Publication Wizard, 593–598, 599

New Subscription Wizard, 599–603

NEWID function, 86, 808

.nodes method (XML), 463, 466–468

nodes of B-Trees

non-leaf level, 196–197
root, 195–196

noise word list for FTS, 631–632

NOLOCK/READUNCOMMITTED optimizer hint, 350, 354

non-clustered indexes

on a clustered table, 205–208
defined, 6
on a heap, 203–205
performance impacts of, 204, 205, 208
selectivity within, 218–219

NONCLUSTERED option (CREATE TABLE), 219

non-identifying relationships

defined, 160, 162
entity box representation of, 161
relationship lines for, 162

non-leaf level nodes of B-Trees, 196–197, 200

non-repeatable reads, 343–344

non-typed XML, 459

normalization. See also de-normalization

defined, 155
misunderstandings about, 156
normal form, defined, 155
normal forms, overview, 157–158
normalized database, defined, 38, 155
prerequisites for, 157
situations benefiting from de-normalization, 179–180

NorthwindSecure.sql script, 634

NOT Boolean operator

with FTS (AND NOT), 629
overview, 48

NOT EXISTS operator, 199

NOT FOR REPLICATION parameter

CREATE TABLE statement, 85
CREATE TRIGGER statement, 367
disabling actions on subscribers, 588

NOT NULL parameter

in CREATE TABLE statement, 86
for foreign key columns, 117
in nested SELECT to find orphaned records, 138
with XML columns, 459

Notification Services

architecture, 839–840
deployment, 840
described, 825, 838
event types, 838–839
learning curve for, 840–841
uses for, 839

ntext data type, 13

n-tier approach, 166, 169

NULLIF function, 808

NULLs

allowing with UNIQUE constraints, 117
ANSI_NULLS option, 242, 266
cascading actions for setting, 116
changing during transfer process, 33
comparison tests against, 138
CREATE TABLE statement parameters for, 86
in foreign keys, 108, 117
ignored by aggregate functions, 56
not allowed for primary keys, 106
overview, 14–15
specifying explicitly, 86
terminator indications for, 163
testing in IF...ELSE statements, 266
testing in OUTER JOIN, 66
testing with ISNULL() function, 143–144
with XML columns, 459

numeric data type, 11

nvarchar data type, 12

nvarchar(max) data type, 12, 13, 168

O

Object Explorer

creating a backup from, 722
overview, 31–32

OBJECT_ID function, 789

OBJECT_NAME function, 789

OBJECTPROPERTY function, 789–792

OBJECTPROPERTYEX function, 792

objects. See also specific objects

administration-related, 2
for connectivity, 815–816
database (overview), 2–5
diagrams (overview), 7
filegroups (overview), 7
full-text catalogs (overview), 11
identifiers for, 15–16
important, list of, 2
indexes (overview), 5–6

ORDER BY clause

- naming, 73–76
 roles (overview), 10
 rules (overview), 10
 schemas (overview), 6
 sprocs as, 9
 tables (overview), 5–6
 testing existence before creating, 147–148
 transaction log (overview), 5
 triggers (overview), 6
 UDFs (overview), 9
 user-defined data types (overview), 10
 users (overview), 10
 using in same order, 356–357
 viewing dependencies for, 129
 views (overview), 7–9
- off-site versus on-site server, 684**
- OLAP (Online Analytical Processing)**
- archiving data, 736
 - concurrency in, 342
 - data warehouse tools, 831
 - FILLFACTOR option for indexes with, 211
 - indexes and performance issues for, 672
 - as lesser focus of this book, 155
 - OLTP versus, with Analysis Services, 826–829
 - OLTP versus, with DBCC SHOWCONTIG, 226–227
 - running on separate server from OLTP, 684
 - SSIS for transforming data from OLTP, 546
- OLEDB providers for MS Search service, 170**
- OLTP (Online Transaction Processing)**
- concurrency in, 342
 - data warehouse use of, 831
 - de-normalizing databases, 674
 - as emphasis of this book, 155
 - FILLFACTOR option for indexes with, 211
 - indexes and performance issues for, 672
 - locks in, 354
 - OLAP versus, with Analysis Services, 826–829
 - OLAP versus, with DBCC SHOWCONTIG, 226–227
 - running on separate server from OLAP, 684
 - SSIS for transforming data for OLAP, 546
 - third normal form for, 155
 - triggers for feeding data into reporting tables, 372–373
- ON clause**
- CREATE DATABASE statement, 78
 - CREATE FULLTEXT INDEX statement, 616
 - CREATE INDEX statement, 209, 214
 - CREATE TABLE statement, 88
 - CREATE TRIGGER statement, 363
 - of foreign key, for cascading, 112–113
- ON FILEGROUP parameter (CREATE FULLTEXT CATALOG), 612**
- OnError event (SSIS), 551**
- one-to-many relationships, 158, 162**
- one-to-one relationships, 158**
- OnExecStatusChanged event (SSIS), 551**
- Online Analytical Processing. See OLAP**
- Online Transaction Processing. See OLTP**
- OnPostExecute event (SSIS), 552**
- OnProgress event (SSIS), 552**
- on-site versus off-site server, 684**
- OPENDATASOURCE function, 794–795**
- open-ended transactions, not allowing, 358**
- opening or running**
- commands with no data set in C#, 820–821
 - commands with no data set in VB.NET, 822–823
 - cursors, 423–424
 - Execute Package Utility, 560
 - packages (SSIS), 560–563
 - scripts with SQLCMD, 259–260
 - sprocs in EXEC procedures, 265
 - SQL Server Profiler, 693
- OPENQUERY function, 795**
- OPENROWSET (BULK) function, 542–543**
- OPENROWSET function, 795**
- OPENXML function**
- cleaning up memory with sp_xml_removedocument, 496
 - described, 493
 - mapping flags, 494
 - metaproperties, 495
 - overview, 795–796
 - script example, 496–497
 - setting up document with sp_xml_preparedocument, 493
 - syntax, 494
 - WITH clause for schema declaration, 494–495
 - XPath to a node for calling, 494
- operators for jobs**
- creating using Management Studio, 701–702
 - creating using T-SQL, 702–704
- operators (T-SQL). See also specific kinds**
- Boolean, 48, 268, 270–271, 272–275, 629
 - standard comparison, 47, 138, 139, 266
 - for WHERE clause, 47–48
- OPTIMISTIC option for cursors, 450**
- optimizer hints**
- defined, 349
 - examples using, 351–352
 - OPTION clause for, 57
 - syntax for, 351
 - table summarizing, 349–351
- OPTION clause, 57**
- @@OPTIONS function, 766–767**
- OR Boolean operator**
- with FTS, 629
 - overview, 48
- ORDER BY clause**
- columns available for, 50
 - DESC keyword in, 51
 - overview, 49–52
 - placed after WHERE clause, 50

orphaned records, finding with nested SELECT

orphaned records, finding with nested SELECT, 138

osql.exe (older versions). See SQLCMD

OUTER JOINS

alternative (legacy) syntax for, 67–68

with DELETE statement, 66

inclusive nature of, 42

LEFT, 43

left and right tables, defined, 42

overview, 42–43

RIGHT, 43

testing for NULLS, 66

OUTPUT keyword for sprocs, 287

Output window (Debugger), 322

ownership. See also naming

access for created objects, 77

ANSI-compliant, 74

chains of, 80

changes with SQL Server 2005, 74–75

of database (dbo), 75–76

of objects created by db_owner role, 76

sa role as dbo, 76

schema as owner, 74

P

packages (SSIS)

building, 552–559

connections for, 556–557, 559

creating a project for, 546–548

defined, 546

executing with Execute Package Utility, 560–562

executing with Management Studio, 562–563

executing with SQL Server Agent, 560

executing within a program, 560

Package Explorer tab, 552

vbScript file for generating data, 552–553

@@PACKET_ERRORS function, 768

@@PACK_RECEIVED function, 767

@@PACK_SENT function, 767

PAD_INDEX option (CREATE INDEX), 210

Page Free Space (PFS), 192

page splits

with B-Trees, 197–199

clustered index exceptions to, 193, 220–221

defined, 193

illustrated, 197–198

at leaf level, 199

performance impacts of, 198–199

at root node level, 198

pages

BCM, 193

BLOB, 192

data, 192

DCM, 193

in an extent, 190, 191

GAM, 192

index, 192

locking, 346

PFS, 192

rows contained by, 191

SGAM, 192

splits, 193, 197–199, 220–221

in storage hierarchy, 191–193

types of, 191

PAGLOCK optimizer hint, 350

parallelism, MAXDOP index option for, 213

PARSENAME function, 809

partition-aware applications, 181

partitioned tables, 180–181

partitioned views, 180, 181, 244

passwords

changing, authentication types and, 26

changing with ALTER LOGIN, 643

changing with sp_password, 646

CREATE LOGIN...WITH options, 641

double, avoiding, 638

encrypting, 638, 646

expiration of, 635–636

length of, 637

makeup of, 637

number of tries to log in, 637

one person, one login, one password principle,

634–635

sharing, prohibiting, 635

sniffers to enforce, limitations of, 636

with SQLCMD, 259

storage of, 637–638

PATH option (FOR XML clause)

at symbol (@) with column names, 489–490, 492

described, 470

EXPLICIT option compared to, 488

forward slash (/) with, 491–492

named columns using, 489

unnamed columns using, 488–489

XPath standard with, 488

PATINDEX function, 801

percent sign (%)

as modulus operator, 271

as wildcard, 48

perfmon (Performance Monitor), 33, 696–697

performance. See also hardware considerations; performance tuning; troubleshooting performance

asterisk (*) in selection criteria reducing, 38

BCP maximizing, 541

BLOBs' impact on, 168, 170

clustered index benefits for, 203

connectivity considerations, 816–819

of cursors, 434

EXISTS and NOT EXISTS benefits for, 199

firing order for triggers impacting, 380

primary keys

of foreign keys versus triggers, 179
 index benefits for, 189, 200
 index fragmentation's impact on, 223–224
 index impacts when modifying data, 219
 indexed views' impact on, 8–9
 join-based syntax versus `EXISTS`, 147
 locking impacts on, 346
 making reasonable tests for, 153
 meanings of, 670
 NetLibs' impact on, 22
 non-clustered indexes' impact on, 204, 205, 208
 page splits' impact on, 198–199
 partitioning considerations for, 180, 181
 perceived, 670, 671
 portability versus, 36
 reusability's impact on, 179
 Shared Memory advantages for, 22
 sproc benefits for, 284, 302–303
 sproc optimization incorrect for, 303–304
 subcategories for improving, 176
 subcategory bottlenecks, 177
 subqueries versus joins and, 135, 152–153
 trigger considerations, 388–390
 user-defined data types' impact on, 10
 of views, 179, 233

Performance Monitor (permon), 33, 696–697

performance tuning. See also **hardware considerations; performance; troubleshooting performance**
 client- versus server-side processing, 673–674
 diminishing returns for, 685–686
 index choices, 671–672
 maintenance, 674–675
 OLTP and OLAP on separate servers, 684
 as ongoing task, 671
 on-site versus off-site server, 684
 overview, 669–670
 in requirements-gathering stage, 670–671
 small gains in repetitive processes, 677–678
 sproc organization, 675–677
 strategic de-normalization, 674
 temporary tables, 677

permissions
 assigned with roles, 75
 for HTTP endpoints, 498
 for .NET assemblies, 398
 object, granting to users, 648–654
 retained by `ALTER PROC` statement, 283
 retained by `ALTER VIEW` statement, 237
 for sprocs, 283
 statement-level, 654–655

PERMISSIONS function, 809

PFS (Page Free Space), 192

phantoms, 344–345

physical models
 defined, 165
 logical models versus, 165
 for reusability, 178
 for subcategories, 175–176
 subcategories in, 171

physical reads statistics, 690–691

PI function, 780

planning for replication
 autonomy concerns, 566–567
 connections, 568
 data concerns, 567, 587–588
 importance of, 587
 latency concerns, 567
 merge replication, 576
 for mobile devices, 588
`NOT FOR REPLICATION` clause, 588
 schema consistency concerns, 567–568
 snapshot replication, 574
 transactional replication, 580

Pointer task (SSIS), 548

pooled connections, 816, 817

populating full-text indexes
 after altering, 618–619
 creating without, 617
 examples, 623–624
 full population, 618, 622
 incremental population, 618, 622
 overview, 622–624
 process of, 609
`sp_fulltext_table` sproc options for, 622, 623
 update (change tracking) population, 619, 622–623

port settings for TCP/IP, 664–665

portability, 36

POWER function, 780

precedence, establishing with batches, 256–258

primary filegroup, 7, 78

primary keys
 adding to existing table, 107–108
 adding with diagramming tools, 185
 BCP enforcement of, 531
 as clustered index default, 219, 220
 defined, 106
 deleting with diagramming tools, 185
 in entity boxes, 161
 as entity constraints, 104
 GUIDs versus, 106
 in identifying relationships, 160
 identity columns versus, 85
 indexing every table with, 672
 in logical model, 167
 need for, 106
 in non-identifying relationships, 160
 NULLs not allowed for, 106
 other data integrity methods compared to, 129–131

primary keys (continued)

primary keys (continued)

overview, 106–108
specifying in CREATE TABLE statement, 87, 107
UPDATE statement for, avoiding, 64

PRIMARY keyword (CREATE DATABASE), 78

PRIVILEGES keyword (GRANT), 650

processadmin role, 657

processors. See CPU intensive tasks

@@PROCID function, 768

Profiler. See SQL Server Profiler

protocols

in Configuration Manager, 21

SOAP 498

TCP/IP 21–22, 24, 664–665

publications. See subscriptions (replication)

publisher (replication)

configuring a publication, 592–597

defined, 568

topology models, 582–583, 584–586

Q

Query Analyzer (older versions). See Query window
query design. See advanced query design

query execution plans

forcing sort order for, 50
graphical showplan for, 689–690
Include Actual Execution Plan option, 30–31
Show Estimated Execution Plan option, 30–31
SHOWPLAN options for, 687–689

query governor, 692–693

query hints. See optimizer hints

.query method (XML), 463–464

Query Optimizer, 30

Query window

color coding in, 27
DB combo box, 31
described, 27
execute button, 28
getting started, 27–29
Include Actual Execution Plan option, 30–31
Results in Grid option, 30
Results in Text option, 29
Results to File option, 30
selecting default database for queries, 31
Show Estimated Execution Plan option, 30–31

QUOTED_IDENTIFIER option, 242

QUOTENAME function, 801

R

RADIANS function, 780

RAID

backups more important for data safety, 683
in ideal system, 686

in larger installations, 682

levels (table), 681–682

meanings of acronym, 680–681

minimum level for database installations, 682

transaction log integrity with, 682

RAISERROR command

arguments with, 297–299

flags, 298

message ID/message string for, 296

placeholder type indicators, 298

setting parameter width, precision, and long/short status, 298–299

severity parameter, 296–297

state parameter, 297

syntax, 296

WITH LOG option, 299

WITH NOWAIT option, 299

WITH SETERROR option, 299

RAM (memory), 683–684

RAND function, 780

Random Array of Independent Disks. See RAID

Random Array of Individual Disks. See RAID

ranged queries, clustered indexes for, 220–221

RAW option (FOR XML clause)

attribute names, 471

columns truncated by Management Studio, 472

described, 470, 471

hierarchical nature of data not displayed by, 473

simple example, 471–472

XML DOM less deep with, 473

RDBMSs (Relational Database Management Systems), 1

RDL (Report Definition Language), 517, 521–522

READ COMMITTED isolation level

dirty reads prevented by, 343

overview, 353

READ UNCOMMITTED isolation level, 354

read-ahead reads statistics, 691

READCOMMITTED optimizer hint, 350

READCOMMITTEDLOCK optimizer hint, 350

READ_ONLY option for cursors, 447

READPAST optimizer hint, 350

READUNCOMMITTED/NOLOCK optimizer hint, 350, 354

REBUILD option

ALTER FULLTEXT CATALOG statement, 613

ALTER INDEX statement, 216–217, 425–427, 734–735

rebuilding indexes

all in database, TableCursor for, 423–427

ALTER INDEX statement for, 216–217, 425–427, 734–735

by dropping and recreating, 211–212

recovery. See also backing up

defined, 729

to different location, 730

- models for, 728–729
to original location, 729
status, 730
T-SQL for, 730–733
- recovery models, 728–729**
- recursion**
defined, 305
looping versus, 307
in sprocs, 305–307
32-level limit for, 306–307
in triggers, 376
- Redundant Array of Inexpensive Disks. See RAID**
- REFERENCES user rights, 649**
- referential integrity. See also data integrity; foreign keys**
constraints, defined, 104
for domain tables, 104
DRI (declarative referential integrity), 362, 367–369
triggers for enforcing, 362
- referential integrity actions, 113–114**
- Relational Database Management Systems (RDBMSs), 1**
- Relational OLAP (ROLAP), 831**
- relationship lines in ERDs, 161–162**
- relationships. See also ERDs (entity-relationship diagrams)**
identifying, 160, 161, 162
many-to-many, 159
non-identifying, 160, 161, 162
one-to-many, 158, 162
one-to-one, 158
subcategories, 171–177
types of, 158–159
- removing. See deleting**
- @@REMSERVER function, 768**
- REORGANIZE option**
ALTER FULLTEXT CATALOG statement, 613–614
ALTER INDEX statement, 217, 735–736
- REPEATABLE READ isolation level**
non-repeatable reads prevented by, 344
overview, 354–355
- REPEATABLEREAD optimizer hint, 350, 355**
- REPLACE function, 801**
- REPLICATE function, 801**
- replication. See also specific kinds**
autonomy, 566–567, 570, 571, 572
configuring a publication, 592–598
configuring the server for, 588–592
connections for, 568, 570, 581
for creating database copies for direct access, 302
data concerns, 567, 587–588
data convergence, 567
defined, 85, 565
filtering data, 570, 596
latency, 567
load and data distribution issues solved by, 565
- merge, 574–576
mixing types of, 581
for mobile devices, 588
models, overview, 570–571
not firing triggers for tasks related to, 367
NOT FOR REPLICATION parameter, 85, 367
planning for, 566–568, 574, 576, 580, 587–588
pull subscriptions, 569–570
push subscriptions, 569–570
RMO for, 605–606
roles, 568–569
schema consistency, 567–568
setting up in Management Studio, 588–603
snapshot, 571–574
subscribers, defined, 569
subscribers, immediate-update, 578, 580–581
subscribers, setting up, 598–603
subscribers, types of, 570
topology models, 581–586
transactional, 577–580
transactional consistency, 567
using the replicated database, 603–605
- Replication Management Objects (RMO), 605–606**
- Report Definition Language (RDL), 517, 521–522**
- Report Model Wizard, 509–511**
- report models**
advantages of, 517
building, 509–512
changing table properties, 511–512
Connection Manager for, 506–507
Data Source Views for, 507–508
Data Source Wizard for, 505, 507
deploying, 512
making available, 516–517
opening a project for, 504
programmability of, 517
RDL for reports, 517
report creation using, 512–517
Report Model Wizard for, 509–511
rights for report design and execution, 517
saving reports, 516
setting up Visual Studio for, 505
- Report Server Projects**
deploying the report, 522–523
formatting dates, 520–521
opening a project for, 517
RDL for reports, 521–522
Report Wizard for, 518–519
- Report Wizard, 518–519**
- Reporting Services**
building report models, 504–512
Configuration Manager for service management, 19
Data Source Views, 507–508
deploying report models, 512
making the model available, 516–517

Reporting Services (continued)

Reporting Services (continued)

overview, 33, 504
programmability of report models, 517
report creation, 512–517
Report Server Projects, 517–523
requirements for useful reports, 503
rights for report design and execution, 517
resizing databases, logical filename for, 78

RESTORE command

example using, 732–733
options, 731
syntax, 730–731

restoring. See recovery

RETURN statement

ALTER PROC example, 290
CREATE PROC example, 288–289
sproc unconditionally exited by, 288
syntax, 288

return values for sprocs

altering, 290
capturing with EXEC command, 289
as integers, 288
RETURN statement for, 288–290
uses for, 288

reusability

database candidates for, 177–178
designing databases for, 177–179
logical versus physical modeling for, 178
performance impacts of, 179

REVERSE function, 801

REVOKE statement, 653–654

RID (row ID)

for a heap, 201, 203
locking, 346

RIGHT function, 802

RIGHT OUTER JOINS, 43

rights

for EXEC commands in sprocs, 264
granting for sprocs, 264
user, 647–655

RMO (Replication Management Objects), 605–606

ROLAP (Relational OLAP), 831

roles. See also application roles; specific roles

with CREATE authority as default, 75
defined, 10, 656
fixed database, 656, 658–659
overview, 656
ownership of objects created by, 76
permissions granted to, 75
server, 656, 657–658
user-defined database, 659–661

ROLLBACK TRAN statement

described, 331
example using, 332–333, 334–335
with nested triggers, 376

overview, 331
syntax, 331
@@TRANCOUNT decremented to 0 by, 252
within triggers, avoiding, 389–390
root node of B-Trees
page splits at, 198
pointing at non-leaf level nodes, 196
pointing directly at data, 195–196
ROUND function, 780
row ID (RID)
for a heap, 201, 203
locking, 346
@@ROWCOUNT function
EXEC scope example, 264
moving value to local variable, 253
overview, 251, 768
using in scripts, 252–253
ROWCOUNT_BIG function, 809
ROWGUIDCOL, identity columns compared to, 85–86
ROWLOCK optimizer hint, 350
rows
contained by pages, 191
counting in a query, 54–56
default value for, 84
as entity data, 5
extending the size limit, 193–194
locking, 346
maximum columns per, 193
maximum size of, 193–194
primer, for foreign keys, 110, 111, 112
@@ROWCOUNT returning number affected by last
statement, 251
in storage hierarchy, 193–194
rowset functions, 794–796
rowversion data type, 12
RTRIM function, 802
rules
binding and unbinding, 127
CHECK constraints versus, 126
constraints versus, 10
creating, 126–127
as domain constraints, 103
dropping, 128
ignored by BCP, 531
as legacy items, 10, 126
in logical model, 165–166, 167
other data integrity methods compared to, 129–131
overview, 10, 126–128
viewing dependencies for, 129
viewing with sp_helptext, 126–127
running. See opening or running
runtime errors. See also error handling
in batches, 253, 255
higher-level, with SQL Server, 291, 296–297
in sprocs, types of, 290

S**sa role.** *See also sysadmin role*

- as alias for dbo, 76
- CREATE authority as default for, 75
- creating sysadmin role member using, 639
- as legacy, 657
- limiting use of, 639, 665
- SQL Server Authentication for, 639

SAFE permission set, 398, 417**SAVE TRAN statement**

- described, 332
- example using, 332–333, 334
- overview, 331–332
- syntax, 332

scalability

- partitioning for, 180–181, 244
- as performance issue, 670
- subcategory benefits for, 176–177

scalar UDFs

- assembly-based, creating, 400–403
- datetime field example, 309–310
- inline use of, 309
- T-SQL-based, creating, 308–310

scan count statistics, 691**scheduling jobs**

- for backing up, 725
- branching rules for, 700
- creating jobs and tasks using Management Studio, 704–712
- creating jobs and tasks using T-SQL, 712–721
- creating operators using Management Studio, 701–702
- creating operators using T-SQL, 702–704
- deleting jobs and tasks using Management Studio, 721
- deleting jobs using T-SQL, 722
- editing jobs using Management Studio, 721
- editing jobs using T-SQL, 722
- jobs, defined, 700
- notification of success or failure, 700, 702, 704, 706
- overview, 700
- tasks, defined, 700
- tasks stored in msdb database, 4

schema binding, 241, 242–243, 317**schema collections.** *See XML schema collections***schema modification locks (Sch-M), 348, 349****schema stability locks (Sch-S), 348, 349****SCHEMA_ID function, 792****SCHEMA_NAME function, 792****schemas.** *See also XML schema collections*

- avoiding, 6
- changes during replication, 567–568
- changing default, caveats for, 76
- consistency in use of, 74
- default (dbo), 75–76
- as foundation for all else, 281

naming, 74–76

for object references, 6

overview, 6

as owners, 74

scope

- of cursors, 428–432
- of database roles, 658
- of @@ERROR versus @Error, 293
- of EXEC command, 262–264

SCOPE_IDENTITY function, 809**Script Tasks (SSIS), 549****scripts.** *See also batches*

- CASE statements in, 270–275
- common parameter-less system functions for, 251–253
- control-of-flow statements, 265–280
- for creating table if not already existing, 267, 268, 269–270
- DECLARE statement in, 249
- defined, 61
- establishing precedence with batches, 256–258
- EXEC command for dynamic, 260–265
- @@IDENTITY function in, 252
- IF...ELSE statements in, 266–270
- for INSERT INTO...SELECT statement, 61–62
- for OPENXML function, 496–497
- for recursive sproc, 305–307
- @@ROWCOUNT function in, 252–253
- running with SQLCMD, 259–260
- separating into batches, 253–254
- setting variable values using SELECT, 250–251
- setting variable values using SET, 250, 251
- simple example, 248
- SMO, 752–753
- statements as parts of, 247
- stored as text files, 248
- tools for writing, 248
- TRY/CATCH blocks in, 277–280
- unified goal characteristic of, 247
- USE statement in, 248–249
- variables in, 62

WAITFOR statements in, 277

WHILE statements in, 275–277

scrollability of cursors, 432–434**SCROLL_LOCKS option for cursors, 447–450****searched CASE statement**

- examples, 272–275

syntax, 270–271, 805

second normal form (2NF), 157**secondary filegroups, 7****secondary files, 7****security.** *See also passwords; roles*

- application roles, 661–664

asymmetric keys, 642, 666–667

avoiding global user accounts, 634–635

carte blanche access, limiting, 635

security (continued)

certificates, 642, 666–667
database for examples, 634
disabling triggers and issues for, 378
enabling NetLibs, cautions for, 22
encrypting triggers for, 363
encrypting views for, 239–241
EXEC command security context, 262, 264
exposing server to the Internet, avoiding, 22
with file storage for BLOBs, 169
groups (pre-SQL Server 7.0), 655–656
guest account issues, 664
for HTTP endpoints, 498
limiting CREATE authority access for, 75
loading NorthwindSecure.sql script for examples, 634
number of tries to log in, 637
one person, one login, one password principle, 634–635
sa role issues, 639, 665
sprocs as tools for, 665
sprocs for, 301–302
system functions, 796–798
TCP/IP port settings, 664–665
total, not possible, 633
UDFs as tools for, 665
user rights, 647–655
variety of approaches to, 633–634
views as tools for, 665
xp_cmdshell issues, 665

securityadmin role, 657

SEEK lookup, 200

Select Case statement (Visual Basic). See CASE statement

SELECT statement. See also specific clauses and keywords

asterisk (*) in selection criteria, avoiding, 38
basic example, 36
correlated subqueries in select list, 142–144
eliminating duplicate data, 57–59
inline CASE statements with, 271–275
nested subqueries returning multiple values, 137–138
nested subqueries returning single values, 136–137
nested subquery syntax, 135
nested, to find orphaned records, 138
optimizer hints in, 351–352
overview, 36–37
for setting variables scripts, 250–251
specifying current database for, 37
specifying tables for, 37
syntax, 36, 52–53
for tables returned by UDFs, 311–315
user rights for, 649

selectivity within indexes, 218–219

self-referencing INNER JOINS, 40, 41

self-referencing tables, 110–112

Send Mail task (SSIS), 550

sensitivity of cursors

dynamic cursors, 435, 442, 445
keyset-driven cursors, 441
overview, 435
static cursors, 435

Sequence Container task (SSIS), 549

SERIALIZABLE isolation level

non-repeatable reads prevented by, 344
overview, 355
phantoms prevented by, 345

SERIALIZABLE/HOLDLOCK optimizer hint, 349, 355

server roles, 656–657. See also specific roles

serveradmin role, 657

@SERVERNAMESPACE function, 252, 768–769

SERVERPROPERTY function, 809–811

server-side versus client-side processing, 672–673

Service Broker, 825, 841–842

Service Master Key, 666

@@SERVICENAME function, 769

SESSIONPROPERTY function, 811–812

SESSION_USER function, 811

SET clause

ALTER DATABASE statement, 92
with expressions, for UPDATE statement, 64

SET DEFAULT, cascading, 116

SET NULL, cascading, 116

SET QUOTED_IDENTIFIER option, 16

SET statement

for IMPLICIT_TRANSACTIONS option, 341
for isolation levels, 353
separating the query from, 250
for setting variables scripts, 250, 251

setupadmin role, 657

severity of errors. See error levels

SGAM (Shared Global Allocation Map), 192

shared locks

compatibility with other lock modes, 349
dirty reads prevented by, 347
with intent exclusive, 348
intent shared, 348

Shared Memory, 22, 24. See also NetLibs (network libraries)

ShortDept.fmt file, 540

ShortDept.txt file, 540

ShortDeptx(fmt file, 540

SHOWPLAN

ALL option, 687–688, 689
TEXT option, 687, 688–689

SIGN function, 780–781

simple CASE statement

examples, 271–272
syntax, 270, 805

Simple Object Access Protocol (SOAP), 498

Simple recovery model, 728, 729

SIN function, 781

single quotes (‘) within strings, 263

SIZE parameter (CREATE DATABASE), 78

smalldate data type, 12

smallint data type, 11

smallmoney data type, 11

SMO (SQL Management Objects)

- backing up a database, 750–751
- complete form code, 753–758
- connection and server references, 745
- creating a database, 745–746
- creating a Windows Application project for, 744–745
- creating logins with, 644
- creating tables, 746–750
- declarations, 745
- defined, 739
- dropping a database, 750
- history of, 740–742
- object model, 742–743
- overview, 758–759
- scripting, 752–753

Snapshot Agent, 572, 598

snapshot folder, 590

snapshot replication

- defined, 571
- Distribution Agent for, 572
- ease of, 571
- mixing with other types, 581
- planning requirements, 574
- process of, 572–573
- Snapshot Agent for, 572, 598
- transactional replication versus, 577
- uses for, 573–574

SOAP (Simple Object Access Protocol), 498

Solution Explorer (SSIS), 552

SOME operator, 48, 139

sort order. See also ORDER BY clause

- COLLATE parameter for setting, 80
- CREATE INDEX options for, 209–210
- for indexes, 194–195, 209–210

SOUNDEX function, 802

SPACE function, 802

spaces embedded in names, avoiding, 16, 83

sp_addapprole sproc, 662

sp_add_job sproc, 712, 713–714, 722

sp_add_jobschedule sproc

- described, 712, 719
- @freq_interval parameter, 719–720
- @freq_recurrence_factor parameter, 721
- @freq_relative_interval parameter, 721
- @freq_subday_interval parameter, 720
- @freq_subday_type parameter, 720
- @freq_type parameter, 719–720
- related sprocs, 722
- syntax, 719
- using, 721

sp_add_jobserver sproc, 714–715, 722

sp_add_jobstep sproc

- @cmdexec_success_code parameter, 717
- @command parameter, 716–717
- @database_name parameter, 717
- described, 712
- @flags parameter, 718
- @job_id parameter, 716
- @job_name parameter, 716
- @on_fail_action parameter, 717
- @on_fail_step_id parameter, 717
- @on_success_action parameter, 717
- @on_success_step_id parameter, 717
- @os_run_priority parameter, 718
- related sprocs, 722
- @retry_interval parameter, 717
- @server parameter, 717
- @step_id parameter, 716
- @step_name parameter, 716
- @subsystem parameter, 716
- syntax, 715

sp_addlogin sproc

- deprecated by CREATE LOGIN, 640
- described, 644, 645
- parameters, 645–646
- syntax, 645

sp_addmessage sproc

- for custom error messages, 299–301
- @lang parameter, 300
- message added to master database sysmessages table, 301
- migrating messages to new server, 301
- @replace parameter, 300
- syntax, 299
- using, 300
- @with_log parameter, 300

sp_add_notification sproc, 704

sp_add_operator sproc, 702–704

sp_addrole sproc, 659

sp_addrolemember sproc, 660

sp_addserver sproc, 252

sp_addumpdevice sproc, 725

sp_attach_db sproc, 80

sp_bindefault sproc, 128

sp_bindrule sproc, 127

spCLRExample assembly-based sproc. See also ExampleProc assembly

- creating from ExampleProc assembly, 398–399
- test call to, 399–400

sp_configure sproc

- enabling CLR, 394
- setting query governor, 692
- setting split point with, 150
- turning on nested triggers, 376

spCursorScope sproc

spCursorScope sproc

global scope example, 428–431

local scope example, 431–432

spCursorScroll sproc, 433–434

sp_dboption sproc

checking database recovery, 732–733

setting cursor default to local, 429

sp_delete_job sproc, 722

sp_delete_jobschedule sproc, 722

sp_delete_jobserver sproc, 722

sp_delete_jobstep sproc, 722

sp_delete_operator sproc, 704

sp_depends sproc, 129

sp_detach_db sproc, 80

sp_dropapprole sproc, 663–664

sp_dropmessage sproc, 301

sp_droprole sproc, 661

sp_droprolemember sproc, 660–661

spEmployee sproc

creating, 282–283

declaring parameters, 284–285

spFactorial sproc, 305–306

sp_fulltext_catalog sproc

@action argument, 620

example using, 620

@ftcat argument, 619

@path argument, 620

syntax, 619

sp_fulltext_database sproc, 610–611

sp_fulltext_table sproc

@action argument, 621–622, 623

@ftcat argument, 621

@keyname argument, 621

options for populating indexes, 622, 623

syntax, 621

@tabname argument, 621

sp_grantdbaccess sproc, 648

sp_grantlogin sproc, 646

sp_help sproc

for verifying table creation, 89

for viewing existing table settings, 93

sp_helpconstraint sproc

clustered index indication from, 109

for constraint information, 109

with self-referencing tables, 110

verifying UNIQUE constraints, 118

sp_helpdb sproc, 81–82, 89–90

sp_help_operator sproc, 704

sp_HELPTEXT sproc

for displaying code for views, 238–239

encrypted views not displayed by, 240

for viewing rules, 126–127

@@SPID function, 769

spInsertValidatedStoreContact sproc, 293–294

split point, 150

sp_password sproc, 646

sprocs (stored procedures). See also Debugger; specific procedures

advantages of, 9

basic example, 282–283

changing with ALTER statement, 283

for creating callable processes, 301

for creating cursors, 428–432

creating from assemblies, 395–400

creating output parameters for, 285–288

custom error messages for, 299–301

declaring parameters, 284–285

DML triggers versus, 362

dropping, 283

error handling, 290–301

extended (XPs), 304–305

granting rights for, 264

incorrect optimization for, 303–304

keeping transactions short, 675

nesting (calling sprocs within), 301

OUTPUT keyword in declaration, 287

overview, 9

parameterization, 284–288

performance benefits of, 284, 302–303

performance tuning, 675–677

RAISERROR command with, 296–299

recursive, 305–307

RETURN statement with, 288–290

return values, 288–290

running in EXEC procedures, 265

for security, 301–302

security context of EXEC commands in, 264

as security tools, 665

stored in sysdatabases table, 3

for subcategory implementation, 174

syntax for creating, 282

syntax for parameter declarations, 284

UDFs versus, 9

WITH RECOMPILE option, 303, 304

sp_setapprole sproc, 662–663

sp_settriggerorder sproc, 378

spTestReturns sproc, 289–290

spTriangular sproc

creating, 306–307

debugging, 322–327

opening in Debugger, 318–320

sp_unbinddefault sproc, 128

sp_unbindrule sproc, 127

sp_update_job sproc, 722

sp_update_jobschedule sproc, 722

sp_update_jobstep sproc, 722

sp_update_operator sproc, 704

sp_xml_preparedocument sproc, 493

sp_xml_removedocument sproc, 496

SQL Management Objects. See **SMO**

SQL NS (SQL Namespaces), 740–741

SQL Server Agent
Configuration Manager for service management, 19
configuring for replication, 589–590, 598
for SSIS package execution, 560
tasks stored in msdb database, 4

SQL Server Authentication
advantages of, 27, 639
altering logins, 642–643
creating logins with CREATE LOGIN, 640–642
creating logins with Management Studio, 643–644
creating logins with SMO, 644
creating logins with sp_addlogin, 640, 644, 645–646
creating logins with sp_grantlogin, 646
creating logins with SQL-DMO, 645
creating logins with WMI, 645
described, 27, 639
disadvantages of, 639
dropping logins, 643
requirements for using, 26
for sa account, 639

SQL Server Books Online (BOL), 18–19

SQL Server Browser, 19

SQL Server Business Intelligence Studio. See **Business Intelligence Studio**

SQL Server Configuration Manager. See **Configuration Manager**

SQL Server Integration Services. See **SSIS**

SQL Server Management Studio. See **Management Studio**

SQL Server Profiler
capturing traces, 693–695
Column Filters, 695–696
importance of, 696
opening, 693
overview, 33–34
Performance Monitor versus, 33
selecting information to collect, 694–695
templates, 694
for troubleshooting performance, 693–696

SQLCMD
case-sensitivity of flags, 259
described, 259
including password with, 259
older tools replaced by, 34, 247, 259
overview, 34
running a query with, 259
running a script with, 259–260
syntax for running, 259

sql_variant data type, 13

SQL_VARIANT_PROPERTY function, 792–793

SQRT function, 781

square brackets ([]) for names, 16

SQUARE function, 781

SSIS (SQL Server Integration Services)
building a simple package, 552–559
Configuration Manager for service management, 19
connections for packages, 556–557, 559
Control Flow tab, 551
creating a project, 546–548
Data Flow tab, 551
as DTS successor, 545
Event Handlers tab, 551–552
events (table), 551–552
executing packages, 560–563
Main window, 546–548, 551–552
opening from Business Intelligence Studio, 546
overview, 32–33
Package Explorer tab, 552
packages, defined, 546
Precedence Constraint Editor, 555–556
Properties window, 552
reorganizing tasks, 548
Solution Explorer, 552
tasks, defined, 546, 548
tasks (table summarizing), 548–551
transformations, 32–33

START FULL POPULATION parameter (ALTER FULLTEXT INDEX), 618

START INCREMENTAL POPULATION parameter (ALTER FULLTEXT INDEX), 618

START UPDATE POPULATION parameter (ALTER FULLTEXT INDEX), 619

starting. See **opening or running**

statement-level permissions, 654–655

static cursors
client-side implementation, 435, 673
CursorTest example, 436–437
defined, 435
FAST_FORWARD cursor conversion to, 446
sensitivity, 435
using temporary tables instead of, 435, 677

statistics
client, 692
STATISTICS IO, 690–691
STATISTICS TIME, 692
for troubleshooting performance, 690–692
WHILE statement for updating, 276–277

STATS_DATE function, 812

STDEV function, 772–773

STDEVP function, 773

STOP parameter (ALTER FULLTEXT INDEX), 619

storage hierarchy
database as highest level of, 189
extents, 190–191
full-text catalogs, 194
pages, 191–193

storage hierarchy (continued)

storage hierarchy (continued)

physical database file, 190
rows, 193–194
transaction log, 190

storage types for Analysis Services, 830–831

stored procedures. See sprocs

StoredProcedures class, 396

STR function, 802

strings

concatenation performed before calling EXEC, 262
quotation marks within, 263

system functions, 798–803

structure in logical model, 167

STUFF function, 802–803

subcategories

bottlenecks from, 177
defined, 171
domain tables for, 174
ERDs for, 172–173, 174–175, 176
example for documents, 171–172
exclusive versus non-exclusive, 172–173
extensibility provided by, 176–177
logical model for, 173–175
performance improved by, 176
physical model for, 175–176
in physical models, 171
sprocs for implementing, 174
views for implementing, 174

subqueries. See also specific kinds

correlated, 139–144
defined, 134
joins versus, 134–135, 152–153
nested, 135–139, 146–147
uses for, 134

subscribers (replication)

anonymous, 570
defined, 569
global, 570
immediate-update, 578, 580–581
local, 570
NOT FOR REPLICATION clause with, 588
preparing for transactional replication, 577–578
setting up using Management Studio, 598–603
topology models, 583–586

subscriptions (replication)

pull, 569, 570
push, 569, 570

SUBSTRING function, 803

SubType relationships. See subcategories

SUM function

with GROUP BY clause, 53–54
overview, 773

Supertype relationships. See subcategories

SUSER_ID function, 797

SUSER_NAME function, 797

SUSER_SID function, 798

SUSER_SNAME function, 798

Switch statement (C, C++, C#, Delphi). See CASE statement

switches. See flags and switches

“sys” functions, 37

sysadmin role. See also sa role

as alias for dbo, 76

CREATE authority as default for, 75

overview, 657

Windows Administrators group mapped into, 657

syscomments table, 239

sysdatabases table, 3

sys.indexes function, 203

sysindexes table, 203

sys.messages function, 279–280

system databases, 3–4. See also specific databases

system functions. See also specific functions

aggregate, 771–773

categories of T-SQL functions, 761

cursor, 773–774

date and time, 774–777

EXEC command scope with, 264

legacy (global variables), 762–770

mathematical, 777–781

metadata, 781–794

moving values into variables, 249

parameter-less (table), 251–252

rowset, 794–796

security, 796–798

string, 798–803

system, 803–812

text and image, 812–813

using instead of system tables, 239

system tables, 3, 239. See also specific tables

SYSTEM_USER function, 812

T

table data type, 13

table scans, 199

TableCursor example

adding ALTER INDEX functionality, 425–427

closing, 424

code listing, 424–425

deallocating, 424

declaring, 423

FETCH statement for, 424

@FETCH_STATUS variable for, 424

opening, 423–424

tables. See also ALTER TABLE statement; CREATE

TABLE statement; specific kinds

adding existing tables to diagram, 182, 183

adding new, with diagramming tools, 183–184

aliases for, 37, 40

- assembly-based UDF for validating e-mail fields in, 400–403
 FROM clause for specifying, 36
 client- versus server-side processing, 673
 creating using Management Studio, 97–99
 creating using SMO, 746–750
 derived, 144–146
 domain data versus entity data in, 5
 dropping with diagramming tools, 184
 dropping with DROP statement, 95
 editing using Management Studio, 99
 identifying at runtime using EXEC, 261–262
 locking, 346
 making views look like, 241
 moving columns using Management Studio, 99
 moving columns using T-SQL, problems with, 94
 naming, 83–84
 overview, 5–6
 partitioned, 180–181
 self-referencing, 110–112
 system, 3
 testing existence before creating, 147–148
 UDFs returning, 310–316
 using rule or default, determining, 129
 viewing dependencies for, 129
- table-valued functions**
 creating from assemblies, 403–407
`IEnumerable` interface for, 404
 uses for, 403
- TABLOCK optimizer hint, 350, 534, 541**
- TABLOCKX optimizer hint, 350**
- TAN function, 781**
- tasks (job). See also scheduling jobs**
 creating using Management Studio, 704–712
 creating using T-SQL, 712–721
 defined, 700
 deleting using Management Studio, 721
 stored in `msdb` database, 4
- TCP/IP. See also NetLibs (network libraries)**
 default port for IP NetLib, 21
 exposing server to the Internet, avoiding, 22
 overview, 22
 port settings, 664–665
 Shared Memory versus, for local servers, 24
- tempdb database**
 creating objects directly in, avoiding, 4
 forcing index to sort in, 212–213
 overview, 4
- temporary tables**
 for EXEC procedures, 264
 for performance tuning, 677
 using instead of static cursors, 435, 677
- terminators**
 for BCP import, 531
 in ERDs, 162–165
- text data type**
 for BLOBs requiring FTS, 170
 overview, 12
- text editors for script writing, 248**
- text system functions, 812–813**
- TEXTIMAGE_ON clause (CREATE TABLE), 88**
- TEXTPTR function, 812**
- @@TEXTSIZE function, 769**
- TEXTVALID function, 813**
- THEN clause**
 in searched CASE statements, 270–271, 272–275
 in simple CASE statements, 270, 271–272
- third normal form (3NF), 155, 157**
- 32-level limit for recursion, 306–307**
- time data type, 12**
- TIMI parameter (WAITFOR), 277**
- timestamp columns**
 BCP with, 531
 replication with, 587
- timestamp data type, 12**
- @@TIMETICKS function, 769**
- tinyint data type, 11**
- TO clause (GRANT), 650**
- topology models for replication**
 central publisher/distributor, 582
 central publisher/remote distributor, 582–583
 central subscriber, 583–584
 mixed models, 584–586
 multiple subscribers/multiple publishers, 586
 publisher/subscriber, 585
 publishing subscriber, 584–585
 self-publishing, 586
 simple models, 581–583
- @@TOTAL_ERRORS function, 769**
- @@TOTAL_READ function, 769**
- @@TOTAL_WRITE function, 769**
- @@TRANCOUNT function, 252, 770**
- transaction context of EXEC command, 262**
- transaction log**
 active portion, defined, 337
 backing up, 724, 728
 BCP use of, 534
 changes propagated at checkpoints, 5, 336
 Checkpoint on Recovery option, 337–338
 checkpoints, 337–339
 circumstances issuing checkpoints, 337
 dirty pages, 336
 failure and recovery, 339–340
 issuing checkpoints manually, 337
`.ldf` extension for, 78, 190
 overview, 5, 336–337
 RAID and integrity of, 682
 restoring, 730–732
 as serial file, 5
 in storage hierarchy, 190

transactional consistency, 567**transactional replication**

- defined, 577
- with immediate-updating subscribers, 578
- Log Reader Agent for, 578, 598
- mixing with other types, 581
- overview, 577
- planning requirements, 580
- preparing subscribers for, 577–578
- process of, 578–579
- snapshot replication versus, 577
- testing, 604–605
- unlogged bulk operations not replicated by, 577
- uses for, 579

transactions. See also isolation levels; locking; transaction log

- ACID test for, 359
- atomicity of, 329–330
- beginning, 331
- committing, 331
- creating savepoints for, 331–332
- example using TRAN commands, 332–335
- implicit, 340–341
- keeping as short as possible, 357, 675
- marking definite begin and end points for, 330
- open-ended, not allowing, 358
- rolling back, 331
- TRAN commands for, 330–332
- @@TRANCOUNT returning active number of, 252

Transact-SQL. See T-SQL**Transfer Tasks (SSIS), 550****transformations, 32–33****trgExampleTrigger trigger**

- creating from ExampleTrigger assembly, 415
- creating table for testing, 414
- testing for delete action, 416–417
- testing for insert action, 415
- testing for update action, 416

triangular

- defined, 306
 - spTriangular sproc for computing, 306–307
- triggers. See also INSTEAD OF triggers**
- architecture changes' impact on, 377
 - BCP with, 531, 534
 - CHECK constraints versus, 362, 367–371
 - for checking the delta of an update, 369–371
 - COLUMNS_UPDATED() function with, 386–388
 - concurrency issues not present for process firing, 389
 - creating from assemblies, 412–417
 - for custom error messages, 371
 - for data integrity implementation, 129, 367–371
 - DDL, 362
 - debugging, 376–377, 390–392
 - defined, 6, 362

- DELETED tables for, 366, 370
- disabling on replication subscribers, 588
- DML versus DDL, 362
- dropping, 390
- encrypting, 363
- events for, 363
- extremism, avoiding, 362
- for feeding data into de-normalized tables, 372–373
- firing order for, 378–380
- FOR | AFTER versus INSTEAD OF, 364–365, 380
- foreign key performance versus, 179
- INSERTED tables for, 366, 370
- keeping as short as possible, 389
- logged versus unlogged activities and, 363
- nested, 376
- not firing for replication-related tasks, 367
- optimizing indexes for, 389
- other data integrity methods compared to, 129–131
- performance considerations, 388–390
- as reactive rather than proactive, 388
- recursive, 376
- for requirements sourced from other tables, 367–369
- reusability considerations for, 178
- rolling back within, avoiding, 389–390
- for setting condition flags, 373–375
- for 6.5 compatibility mode, 366
- syntax for creating, 363
- turning off without removing, 377–378
- UPDATE() function with, 386
- for updating summary information, 372
- uses for, 362

troubleshooting performance. See also performance; performance tuning

- client statistics for, 692
- DBCC for, 692
- Performance Monitor for, 696–697
- query governor for, 692–693
- SHOWPLAN options for, 687–690
- SQL Server Profiler for, 693–696
- STATISTICS IO for, 690–691
- STATISTICS TIME for, 692
- tools for, 686–687

Truncate On Checkpoint option, 338**TRUSTWORTHY parameter**

- ALTER DATABASE statement, 406
- CREATE DATABASE statement, 80

TRY/CATCH blocks

- for avoiding script termination on errors, 293
- calling uspLogError sproc, 286–288
- CREATE script example, 278–279
- error condition retrieval functions, 279
- @@ERROR function versus, 295
- error levels, 278
- syntax, 278

trapping 2714 error, 295
 using error condition retrieval, 286–288
 viewing all system error messages, 279–280

T-SQL (Transact-SQL). *See also specific statements*
 ANSI entry-level compliance of, 27, 36
 backing up using, 726–728
 CLR compliance, 35
 creating backup device using, 725
 creating jobs and tasks using, 712–721
 creating operators using, 702–704
 defined, 27
 deleting jobs using, 722
 editing jobs using, 722
 portability issues, 36
 restoring data using, 730–733

1205 error. *See deadlocks*

2714 error, 295

TYPE COLUMN parameter (CREATE FULLTEXT INDEX), 615–616

TYPE_WARNING option for cursors, 446, 450–452

U

UDFs (user-defined functions). *See also Debugger*
 applying as CHECK constraints, 403
 creating from assemblies, 400–403
 datetime field example, 309–310
 defined, 308
 deterministic, 316–317
 EXEC command not allowed in, 262, 265
 file storage implementation using, 170
 in indexed views, 242
 inline use of, 309
 overview, 9
 returning a scalar value, 308–310
 returning a table, 310–316
 as security tools, 665
 sprocs versus, 9
 syntax for creating, 308
 for validating e-mail fields in tables, 400–403

unbinding
 defaults, 128
 rules, 127

underscore (_)
 in table names, avoiding, 83
 as wildcard, 48

UNICODE function, 803

UNIONS
 compatibility required for columns, 70
 default return option for, 70
 DISTINCT clause with, 70
 headings returned for, 70
 JOINS versus, 69
 overview, 69–72
 rules for, 70

UNIQUE constraints
 adding to existing table, 118
 allowing NULLS, 117
 BCP enforcement of, 531
 as entity constraints, 104
IGNORE_DUP_KEY option with, 211
 in logical model, 167
 overview, 116–117
 specifying in CREATE TABLE statement, 117–118

uniqueidentifier columns, replication with, 587

uniqueidentifier data type, 12

UNSAFE permission set, 394, 398

UPDATE () function, 386

update locks
 compatibility with other lock modes, 349
 deadlocks prevented by, 347–348
 overview, 347

UPDATE statement
 cascading updates, 112
 changing multiple columns, 64
 DEFAULT constraints with, 120
 overview, 62–64
 for primary keys, avoiding, 64
 SET clause with expressions for, 64
 syntax, 62
 update locks, 347–348
 user rights for, 649
 with views, 236–237

UPDATE STATISTICS statement, 212

UPDATE triggers
 checking the delta of an update, 369–371
 COLUMNS_UPDATED() function with, 386–388
 CREATE TRIGGER parameter for, 366
 creating from assemblies, 412–417
 firing order for, 378
 INSTEAD OF triggers, 384
 for requirements sourced from other tables, 367–369
 UPDATE () function with, 386
 for updating summary information, 372

UPDLOCK optimizer hint, 351

UPPER function, 803

uppercase. See case and case sensitivity

USE statement
 in scripts, 248–249
 specifying current database, 37

USER function, 798

user rights
 categories of, 647
 DELETE, 649
 denying for targeted object, 652–653
 EXECUTE, 649
 granting access to a specific database, 647–648
 granting object permissions, 648–654
 granting statement-level permissions, 654–655
 INSERT, 649

user rights (continued)

user rights (continued)

REFERENCES, 649

revoking, 653–654

SELECT, 649

UPDATE, 649

user-defined data types

accessing, 418–419

caveats for using, 10

creating assembly for, 417

creating from assemblies, 417–419

dropping, 419

overview, 10

replication with, 587

rules restricting, 10

user-defined database roles

adding users to, 660

creating, 659

described, 656

dropping, 661

fixed database roles versus, 659

removing users from, 660–661

user-defined functions. See UDFs

USER_ID function, 798

USER_NAME function, 812

users. See also roles

adding to a database, 647–648

adding to user-defined database roles, 660

defined, 10

direct access for, 302

global accounts, avoiding for security reasons,
634–635

guest account, 664

one person, one login, one password principle,
634–635

removing from user-defined database roles, 660–661

rights, 647–655

storage of profiles, 637–638

termination by ALTER DATABASE statement, 92

uspLogError sproc

code for, 285–286

output parameter in, 286

TRY/CATCH example calling, 286–288

V

validating e-mail fields in tables, 400–403

.value method (XML), 463, 464–465

VALUES keyword (INSERT), 59–60

VAR function, 773

varbinary data type, 13

varbinary(max) data type, 13, 168, 194

varchar data type, 12

varchar(max) data type

for BLOBs, 168

for file names and paths, 169

overview, 12

replacing text data type, 12

row size limit extended by, 194

variables

declaring in scripts, 62, 249

global (legacy system functions), 762–770

moving system function values into, 249

setting values in scripts using SELECT, 250–251

setting values in scripts using SET, 250, 251

VARP function, 773

VB.NET connectivity example, 821–823

@EVERSION function, 770

vertical filtering or partitioning, 570

Very Large Databases (VLDBs), filegroups for, 78

VIA (Virtual Interface Adapter), 20, 23. See also

NetLibs (network libraries)

Vieira, Robert (*Beginning SQL Server 2005 Programming*), 17, 25, 44, 329, 457

VIEW_METADATA option for views, 241

views

added columns missing from, 94

changing data through, 236–237

complex examples, 233–236

Data Source Views, 507–508

defined, 231

displaying code for, 238–239

dropping, 238

editing with T-SQL, 237

encrypting, 239–241

indexed, 8–9, 242–244

inline, as derived tables, 144

INSTEAD OF DELETE triggers for, 384–385

INSTEAD OF INSERT triggers for, 381–383

INSTEAD OF UPDATE triggers for, 384

joined data with, 236, 237

making look like tables, 241

overview, 7–9, 244–245

partitioned, 180–181, 244

pass-through, 232

performance considerations, 179, 233

required fields with, 237

restricting insertions and updates in, 237

reusability considerations for, 178

schema binding for, 241, 242–243

as security tools, 665

simple example, 232

as stored queries, 231

for subcategory implementation, 174

supporting client-side cursors, 241

tips for using, 244–245

uses for, 7, 231, 244

VIEW_METADATA option for, 241

WHERE clause with, 234

WITH CHECK OPTION for, 237

Virtual Interface Adapter (VIA), 20, 23. See also

NetLibs (network libraries)

Visio diagramming tool, 159

Visual Studio

adding a sproc to assembly project, 395

creating a project for assemblies, 395, 404, 408, 412

setting up for Business Intelligence Studio, 505

VLDBs (Very Large Databases), filegroups for, 78

W

WAITFOR statement, 276–277

Watch window (Debugger), 322

Web Service Task (SSIS), 550

WEIGHT keyword (FTS), 630–631

WHEN clause

in searched CASE statements, 270–271, 272–275

in simple CASE statements, 270, 271–272

WHERE clause

correlated subqueries in, 140–142

in DELETE statement, 65

EXISTS operator in, 199

with JOINS, 48–49

nested subquery syntax, 135

operators (table), 47–48

overview, 45–49

placed before ORDER BY clause, 50

for tables returned by UDFs, 311, 312, 313–314, 315

using table as filtering source, 49

with views, 234

WHILE statement

BEGIN...END blocks with, 276

BREAK statement with, 276

CONTINUE statement with, 276

described, 275

example for updating statistics, 276–277

syntax, 276

for tables returned by UDFs, 314

wildcards for LIKE operator, 48

windowing not supported by .NET assemblies, 394

Windows authentication

advantages and disadvantages of, 26–27

described, 26, 639

overview, 646–647

Windows domain, domain listing versus, 45

Windows Management Instrumentation (WMI), 550, 645, 741

WITH ACCENT_SENSITIVITY parameter (CREATE FULLTEXT CATALOG), 612

WITH APPEND parameter (CREATE TRIGGER), 366

WITH CHANGE_TRACKING parameter (CREATE FULLTEXT INDEX), 616–617

WITH CHECK OPTION for views, 237

WITH clause

CREATE INDEX statement, 210

CREATE LOGIN statement, 640–641

OPENXML function schema declaration, 494–495

RAISERROR command, 299

WITH DB CHAINING parameter (CREATE DATABASE), 80

WITH ENCRYPTION option

ALTER TRIGGER statement, 363

ALTER VIEW statement, 240

CREATE TRIGGER statement, 363

WITH GRANT OPTION (GRANT), 650

WITH NO POPULATION parameter (CREATE FULLTEXT INDEX), 616–617

WITH NOCHECK option for constraints, 122–124

WITH PERMISSION_SET options (CREATE ASSEMBLY), 398

WITH RECOMPILE option for sprocs, 303, 304

WITH SCHEMABINDING option

ALTER FUNCTION statement, 317

ALTER VIEW statement, 242–243

CREATE VIEW statement, 241

for DayOnly() UDF, 317

required for indexed views, 242

WITH XMLNAMESPACES() declaration

overview, 464

for XML .exist method, 468

for XML .modify method, 466

for XML .nodes method, 467

for XML .query method, 464

for XML .value method, 465

WMI Tasks (SSIS), 550

WMI (Windows Management Instrumentation), 550, 645, 741

X

XLOCK optimizer hint, 351

XML data type

defining a column as, 458–459

enforcing constraints, 458, 469

.exist method, 463, 468–469

methods available, 463

.modify method, 463, 465–466

.nodes method, 463, 466–468

overview, 13, 458

.query method, 463–464

schema collections, 458, 460–463

.value method, 463, 464–465

XML (Extensible Markup Language). See also FOR XML clause; XML data type

format files, 537

further information, 458

HTTP endpoints for, 497–500

XML (continued)

XML (continued)

importance of understanding, 457
indexes, 214–215, 497
naming every column when using, 471
non-typed, 459
OPENXML function, 493–497
schema collections, 458, 460–463

XML indexes, 214–215, 497

XML schema collections

- altering, 462
- creating, 461–462
- defined, 460
- described, 458
- dropping, 463
- enforcing constraints beyond, 469

`XML_SCHEMA_NAMESPACE()` function for, 460–461

XML Tasks (SSIS), 551

`XML_SCHEMA_NAMESPACE()` function, 460–461
`xp_cmdshell`, limiting access to, 665
XPs (extended sprocs), 304–305

Y

YEAR function, 776–777