



Beginning **SQL Server™ 2005** **Programming**

Robert Vieira



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

Beginning SQL Server™ 2005 Programming

Robert Vieira



WILEY

Wiley Publishing, Inc.

Beginning SQL Server™ 2005 Programming

Beginning SQL Server™ 2005 Programming

Robert Vieira



WILEY

Wiley Publishing, Inc.

Beginning SQL Server™ 2005 Programming

Published by

Wiley Publishing, Inc.

10475 Crosspoint Boulevard

Indianapolis, IN 46256

www.wiley.com

Copyright © 2006 by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN-13: 978-0-7645-8433-6

ISBN-10: 0-7645-8433-2

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

1MA/QT/QS/QW/IN

Library of Congress Cataloging-in-Publication Data: Available from publisher

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

Robert Elliott

Development Editor

Adaobi Obi Tulton

Technical Editor

John Mueller

Production Editor

Pamela Hanley

Copy Editor

Nancy Rapoport

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

Project Coordinator

Kristie Rees

Quality Control Technician

Laura Albert

Jessica Kramer

Graphics and Production Specialists

Carrie A. Foster

Lauren Goddard

Denny Hager

Stephanie D. Jumper

Barbara Moore

Alicia South

Proofreading and Indexing

TECHBOOKS Production Services

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

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 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 knowledge of business and computing. Beyond his Bachelor’s degree, he has been certified as a Certified Management Accountant as well as Microsoft Certified as a Solutions Developer (MCSD), Trainer (MCT), and Database Administrator (MCDBA).

Rob is currently a Software Architect for WebTrends Corporation in Portland, Oregon.

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

Acknowledgments

Five years have gone by, and my how life has changed since the last time I wrote a title on SQL Server. So many people have affected my life in so many ways, and, as always, there are a ton of people to thank.

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. Even as my youngest has asked me several times when I'm "going to be done with that book" (she is not pleased with how it takes up some of my play time), she has been tremendously patient with me all during the development of this book. My eldest just continues to amaze me in her maturity and her sensitivity to what doing a book like this requires (and, of course, what it means to her college education!). The thank yous 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 the 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 a 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 edit 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 you — wherever you are Kate, I hope you are doing splendidly.

Adaobi Obi Tulton — Who has had to put up with yet another trialing year in my life and what that has sometimes meant to 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.

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...).

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 know when to let it be — John did a terrific job on both counts.

Acknowledgments

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 Richard Waymire.

Contents

Acknowledgments	ix
Introduction	xxi
Chapter 1: RDBMS Basics: What Makes Up a SQL Server Database?	1
An Overview of Database Objects	2
The Database Object	2
The Transaction Log	6
The Most Basic Database Object: Table	6
Filegroups	8
Diagrams	8
Views	9
Stored Procedures	10
User-Defined Functions	10
Users and Roles	11
Rules	11
Defaults	11
User-Defined Data Types	11
Full-Text Catalogs	12
SQL Server Data Types	12
NULL Data	17
SQL Server Identifiers for Objects	17
What Gets Named?	17
Rules for Naming	18
Summary	18
Chapter 2: Tools of the Trade	19
Books Online	20
The SQL Server Configuration Manager	21
Service Management	22
Network Configuration	22
The Protocols	23
On to the Client	26

Contents

The SQL Server Management Studio	28
Getting Started	28
Query Window	33
SQL Server Integration Services (SSIS)	38
Bulk Copy Program (bcp)	39
SQL Server Profiler	40
sqlcmd	40
Summary	40
Chapter 3: The Foundation Statements of T-SQL	41
Getting Started with a Basic SELECT Statement	42
The SELECT Statement and FROM Clause	42
The WHERE Clause	45
ORDER BY	49
Aggregating Data Using the GROUP BY Clause	52
Placing Conditions on Groups with the HAVING Clause	61
Outputting XML Using the FOR XML Clause	63
Making Use of Hints Using the OPTION Clause	63
The DISTINCT and ALL Predicates	64
Adding Data with the INSERT Statement	66
The INSERT INTO . . . SELECT Statement	70
Changing What You've Got with the UPDATE Statement	72
The DELETE Statement	75
Summary	77
Exercises	77
Chapter 4: JOINS	79
JOINS	79
INNER JOINS	81
How an INNER JOIN Is Like a WHERE Clause	85
OUTER JOINS	89
The Simple OUTER JOIN	90
Dealing with More Complex OUTER JOINS	95
Seeing Both Sides with FULL JOINS	99
CROSS JOINS	100
Exploring Alternative Syntax for Joins	102
An Alternative INNER JOIN	102
An Alternative OUTER JOIN	103
An Alternative CROSS JOIN	104

The UNION	104
Summary	109
Exercises	110
Chapter 5: Creating and Altering Tables	111
Object Names in SQL Server	111
Schema Name (aka Ownership)	112
The Database Name	114
Naming by Server	114
Reviewing the Defaults	114
The CREATE Statement	115
CREATE DATABASE	115
CREATE TABLE	121
The ALTER Statement	133
ALTER DATABASE	133
ALTER TABLE	137
The DROP Statement	140
Using the GUI Tool	142
Creating a Database Using the Management Studio	142
Backing into the Code: The Basics of Creating Scripts with the Management Studio	148
Summary	149
Exercises	149
Chapter 6: Constraints	151
Types of Constraints	152
Domain Constraints	152
Entity Constraints	153
Referential Integrity Constraints	154
Constraint Naming	154
Key Constraints	155
PRIMARY KEY Constraints	155
FOREIGN KEY Constraints	158
UNIQUE Constraints	169
CHECK Constraints	170
DEFAULT Constraints	171
Defining a DEFAULT Constraint in Your CREATE TABLE Statement	172
Adding a DEFAULT Constraint to an Existing Table	173
Disabling Constraints	173
Ignoring Bad Data When You Create the Constraint	174
Temporarily Disabling an Existing Constraint	176

Contents

Rules and Defaults — Cousins of Constraints	178
Rules	178
Defaults	180
Determining Which Tables and Datatypes Use a Given Rule or Default	181
Triggers for Data Integrity	181
Choosing What to Use	181
Summary	183
Chapter 7: Adding More to Our Queries	185
What Is a Subquery?	186
Building a Nested Subquery	186
Correlated Subqueries	190
How Correlated Subqueries Work	190
Correlated Subqueries in the WHERE Clause	190
Dealing with NULL Data — the ISNULL Function	194
Derived Tables	195
The EXISTS Operator	197
Using EXISTS in Other Ways	199
Mixing Datatypes: CAST and CONVERT	201
Performance Considerations	204
JOINS vs. Subqueries vs. ?	204
Summary	205
Exercises	206
Chapter 8: Being Normal: Normalization and Other Basic Design Issues	207
Tables	208
Keeping Your Data “Normal”	208
Before the Beginning	209
The First Normal Form	211
The Second Normal Form	214
The Third Normal Form	216
Other Normal Forms	218
Relationships	219
One-to-One	219
One-to-One or Many	221
Many-to-Many	223
Diagramming	227
Tables	230
Adding and Deleting Tables	230
Relationships	237

De-Normalization	241
Beyond Normalization	241
Keep It Simple	242
Choosing Datatypes	242
Err on the Side of Storing Things	242
Drawing Up a Quick Example	243
Creating the Database	243
Adding the Diagram and Our Initial Tables	244
Adding the Relationships	248
Adding Some Constraints	251
Summary	252
Exercises	252
 Chapter 9: SQL Server Storage and Index Structures	 255
SQL Server Storage	255
The Database	255
The Extent	256
The Page	256
Rows	257
Understanding Indexes	257
B-Trees	258
How Data Is Accessed in SQL Server	262
Creating, Altering, and Dropping Indexes	270
The CREATE INDEX Statement	270
Creating XML Indexes	276
Implied Indexes Created with Constraints	277
Choosing Wisely: Deciding What Index Goes Where and When	277
Selectivity	277
Watching Costs: When Less Is More	278
Choosing That Clustered Index	278
Column Order Matters	281
Dropping Indexes	281
Use the Database Engine Tuning Wizard	282
Maintaining Your Indexes	282
Fragmentation	282
Identifying Fragmentation vs. Likelihood of Page Splits	283
Summary	286
Exercises	288

Chapter 10: Views	289
Simple Views	289
Views as Filters	293
More Complex Views	295
Using a View to Change Data — Before INSTEAD OF Triggers	298
Editing Views with T-SQL	301
Dropping Views	302
Creating and Editing Views in the Management Studio	302
Editing Views in the Management Studio	306
Auditing: Displaying Existing Code	306
Protecting Code: Encrypting Views	308
About Schema Binding	309
Making Your View Look Like a Table with VIEW_METADATA	310
Indexed (Materialized) Views	310
Summary	313
Exercises	314
 Chapter 11: Writing Scripts and Batches	 315
Script Basics	315
The USE Statement	316
Declaring Variables	317
Using @@IDENTITY	320
Using @@ROWCOUNT	324
Batches	325
Errors in Batches	327
When to Use Batches	327
SQLCMD	330
Dynamic SQL: Generating Your Code On-the-Fly with the EXEC Command	334
The Gotchas of EXEC	335
Summary	339
Exercises	340
 Chapter 12: Stored Procedures	 341
Creating the Sproc: Basic Syntax	342
An Example of a Basic Sproc	342
Changing Stored Procedures with ALTER	343
Dropping Sprocs	344

Parameterization	344
Declaring Parameters	344
Control-of-Flow Statements	349
The IF . . . ELSE Statement	349
The CASE Statement	360
Looping with the WHILE Statement	366
The WAITFOR Statement	367
TRY/CATCH Blocks	368
Confirming Success or Failure with Return Values	369
How to Use RETURN	369
Dealing with Errors	371
The Way We Were . . .	372
Handling Errors Before They Happen	378
Manually Raising Errors	381
Adding Your Own Custom Error Messages	385
What a Sproc Offers	388
Creating Callable Processes	389
Using Sprocs for Security	390
Sprocs and Performance	391
Extended Stored Procedures (XPs)	393
A Brief Look at Recursion	393
Debugging	396
Setting Up SQL Server for Debugging	396
Starting the Debugger	397
Parts of the Debugger	400
Using the Debugger Once It's Started	402
.NET Assemblies	406
Summary	407
Exercises	407
 Chapter 13: User Defined Functions	 409
What a UDF Is	409
UDFs Returning a Scalar Value	410
UDFs That Return a Table	414
Understanding Determinism	421
Debugging User-Defined Functions	423
.NET in a Database World	423
Summary	424
Exercise	424

Chapter 14: Transactions and Locks **425**

Transactions	425
BEGIN TRAN	426
COMMIT TRAN	427
ROLLBACK TRAN	427
SAVE TRAN	427
How the SQL Server Log Works	428
Failure and Recovery	429
Implicit Transactions	431
Locks and Concurrency	431
What Problems Can Be Prevented by Locks	432
Lockable Resources	435
Lock Escalation and Lock Effects on Performance	435
Lock Modes	436
Lock Compatibility	438
Specifying a Specific Lock Type — Optimizer Hints	439
Setting the Isolation Level	440
Dealing with Deadlocks (aka “A 1205”)	442
How SQL Server Figures Out There’s a Deadlock	443
How Deadlock Victims Are Chosen	443
Avoiding Deadlocks	443
Summary	445

Chapter 15: Triggers **447**

What Is a Trigger?	448
ON	449
WITH ENCRYPTION	450
The FOR AFTER vs. the INSTEAD OF Clause	450
WITH APPEND	452
NOT FOR REPLICATION	453
AS	453
Using Triggers for Data Integrity Rules	453
Dealing with Requirements Sourced from Other Tables	454
Using Triggers to Check the Delta of an Update	455
Using Triggers for Custom Error Messages	457
Other Common Uses for Triggers	457
Other Trigger Issues	458
Triggers Can Be Nested	458
Triggers Can Be Recursive	458

Triggers Don't Prevent Architecture Changes	458
Triggers Can Be Turned Off Without Being Removed	459
Trigger Firing Order	459
INSTEAD OF Triggers	461
Performance Considerations	462
Triggers Are Reactive Rather Than Proactive	462
Triggers Don't Have Concurrency Issues with the Process That Fires Them	462
Using IF UPDATE() and COLUMNS_UPDATED	463
Keep It Short and Sweet	465
Don't Forget Triggers When Choosing Indexes	465
Try Not to Roll Back Within Triggers	465
Dropping Triggers	466
Debugging Triggers	466
Summary	468
 Chapter 16: A Brief XML Primer	 469
XML Basics	470
Parts of an XML Document	471
Namespaces	479
Element Content	481
Being Valid vs. Being Well Formed — Schemas and DTDs	481
What SQL Server Brings to the Party	482
Retrieving Relational Data in XML Format	483
RAW	484
AUTO	486
EXPLICIT	487
PATH	503
OPENXML	508
A Brief Word on XSLT	514
Summary	516
 Chapter 17: Reporting for Duty, Sir!: A Look At Reporting Services	 517
Reporting Services 101	518
Building Simple Report Models	518
Data Source Views	523
Report Creation	529
Report Server Projects	532
Deploying the Report	537
Summary	538

Contents

Chapter 18: Getting Integrated With Integration Services	539
Understanding the Problem	539
Using the Import/Export Wizard to Generate Basic Packages	540
Executing Packages	547
Using the Execute Package Utility	547
Executing Within the Business Intelligence Development Studio	549
Executing Within the Management Studio	549
Editing the Package	550
Summary	553
Chapter 19: Playing Administrator	555
Scheduling Jobs	556
Creating an Operator	557
Creating Jobs and Tasks	558
Backup and Recovery	567
Creating a Backup: a.k.a. “A Dump”	567
Recovery Models	570
Recovery	571
Index Maintenance	572
ALTER INDEX	573
Archiving Data	575
Summary	576
Exercises	576
Appendix A: Exercise Solutions	577
Appendix B: System Functions	587
Appendix C: Finding the Right Tool	639
Appendix D: Very Simple Connectivity Examples	647
Appendix E: Installing and Using the Samples	651
Index	655

Introduction

What a long strange trip it's been. When I first wrote *Professional SQL Server 7.0 Programming* in early 1999, the landscape of both books and the development world was much different than it is today. At the time, .NET was as yet unheard of, and while Visual Studio 98 ruled the day as the most popular development environment, Java was coming on strong and alternative development tools such as Delphi were still more competitive than they typically are today. The so-called "dot com" era was booming, and the use of database management systems (DBMS) such as SQL Server was growing.

There was, however a problem. While one could find quite a few books on SQL Server, they were all oriented towards the administrator. They spent tremendous amounts of time and energy on things that the average developer did not give a proverbial hoot about. Something had to give, and as my development editor and I pondered the needs of the world, we realized that we could not solve world hunger or arms proliferation ourselves, but we could solve the unrealized need for a new kind of SQL book — one aimed specifically at developers.

At the time, we wrote *Professional SQL Server 7.0 Programming* to be everything to everyone. It was a compendium. It started at the beginning, and progressed to a logical end. The result was a very, very large book that filled a void for a lot of people (hooray!).

SQL Server 2005 represents the 2nd major revision to SQL Server since that time, and, as we did the planning for this cycle of books, we realized that we once again had a problem — it was too big. The new features of SQL Server 2005 created a situation where there was simply too much content to squeeze into one book, and so we made the choice to split the old Professional series title into a Beginning and a more targeted Professional pair of titles. You are now holding the first half of that effort.

My hope is that you find something that covers all of the core elements of SQL Server with the same success that we had in the previous *Professional SQL Server Programming* titles. When we're done, you should be set to be a highly functional SQL Server 2005 programmer, and, when you need it, be ready to move on to the more advanced Professional title.

Who This Book Is For

It is almost sad that the word "Beginner" is in the title of this book. Don't get me wrong; if you are a beginner, then this title is for you. But it is designed to last you well beyond your beginning days. What is covered in this book is necessary for the beginner, but there is simply too much information for you to remember all of it all the time, and so it is laid out in a fashion that should make a solid review and reference item even for the more intermediate, and, yes, even advanced user.

The beginning user will want to start right at the beginning. Things are designed such that just about everything in this book is a genuine "need to know" sort of thing. With the possible exception of the chapters on XML, Reporting Services and Integration Services, every item in this book is a fundamental item to you having the breadth of understanding you need to make well-informed choices on how you approach your SQL Server problems.

Introduction

For the intermediate user, you can probably skip perhaps as far as chapter 7 or 8 for starting. While I would still recommend scanning the prior chapters for holes in your skills or general review, you can probably skip ahead with little harm done and get to something that might be a bit more challenging for you.

Advanced users, in addition to utilizing this as an excellent reference resource, will probably want to focus on Chapter 12 and beyond. Virtually everything from that point forward should be of some interest (the new debugging, transactions, XML, Reporting Services and more!).

What This Book Covers

Well, if you're read the title, you're probably not shocked to hear that this book covers SQL Server 2005 with a definite bent towards the developer perspective.

SQL Server 2005 is the latest incarnation of a database management system that has now been around for what is slowly approaching two decades. It builds on the base redesign that was done to the product in version 7.0, and significantly enhances the compatibility and featureset surrounding XML, .NET, user defined datatypes as well as a number of extra services. This book focuses on core development needs of every developer regardless of skill level. The focus is highly orienting to just the 2005 version of the product, but there is regular mention of backward compatibility issues as they may affect your design and coding choices.

How This Book Is Structured

The book is designed to become progressively more advanced as you progress through it, but, from the very beginning, I'm assuming that you are already an experienced developer — just not necessarily with databases. In order to make it through this book you do need to already have understanding of programming basics such as variables, data types, and procedural programming. You do not have to have ever seen a query before in your life (though I suspect you have).

The focus of the book is highly developer-oriented. This means that we will, for the sake of both brevity and sanity, sometimes gloss over or totally ignore items that are more the purview of the database administrator than the developer. We will, however, remember administration issues as they either affect the developer or as they need to be thought of during the development process — we'll also take a brief look at several administration related issues in Chapter 19.

The book makes a very concerted effort to be language independent in terms of your client side development. VB, C#, C++, Java and other languages are generally ignored (we focus on the server side of the equation) and treated equally where addressed.

In terms of learning order, we start by learning the foundation objects of SQL, and then move onto basic queries and joins. From there, we begin adding objects to our database and discuss items that are important to the physical design — then it is on to the more robust code aspects of SQL Server scripting, stored procedures, user defined functions, and triggers. We then look at a few of the relative peripheral features of SQL server. Last but not least, we wrap things up with a discussion of administration meant to help you keep the databases you develop nice and healthy.

What You Need to Use This Book

In order to make any real viable use of this book, you will need an installation of SQL Server. The book makes extensive use of the actual SQL Server 2005 management tools, so I highly recommend that you have a version that contains the full product rather than just using SQL Server Express. That said, the book is focused on the kind of scripting required for developers, so even SQL Server Express users should be able to get the lion's share of learning out of most of the chapters.

A copy of Visual Studio is handy for working with this book, but most of the Visual Studio features needed are included in the Business Intelligence Studio that comes along with the SQL Server product.

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.

Try It Out

The *Try It Out* is an exercise you should work through, following the text in the book.

1. They usually consist of a set of steps.
2. Each step has a number.
3. Follow the steps through with your copy of the database.

How It Works

After each *Try It Out*, the code you've typed will be explained in detail.

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.

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 file names, 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 <http://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 0-7645-8433-2 (changing to 978-0-7645-8433-6 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 <http://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 <http://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 your 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.

Beginning SQL Server™ 2005 Programming

1

RDBMS Basics: What Makes Up a SQL Server Database?

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 could 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, or the *business rules* of the data.

Don't confuse what I'm calling the "business rules of data" with the more generalized business rules that drive your entire system (for example, someone can't see anything until they've logged on, or automatically adjusting the current period in an accounting system on the first of the month). Those types of rules can be enforced at virtually any level of the system. (These days, it's usually in the middle or client tier of an n-tier system). Instead, what we're talking about here are the business rules that specifically relate to the data. For example, you can't have a sales order with a negative amount. With an RDBMS, we can incorporate these rules right into the integrity of the database itself.

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. Therefore, 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	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, we 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 ourselves with here.)

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.

What types of tables do we store in just one database and what goes in a separate database? We'll discuss that in some detail later in the book, but for now we'll take the simple approach of saying that any data that is generally thought of as belonging to just one system, or is significantly related will be stored in a single database. An RDBMS, such as SQL Server, may have multiple user databases on just one server, or it may have only one. How many can reside on one SQL Server depends on such factors as capacity (CPU power, disk I/O limitations, memory, etc.), autonomy (you want one person to have management rights to the server this system is running on, and someone else to have admin rights to a different server), or just how many databases your company or client has. Many servers have only one production database; others may have many. Also keep in mind that with any version of SQL Server you're likely to find in production these days (SQL Server 2000 was already five years old by the time it

was replaced, so we'll assume most shops have that or higher), we have the ability to have multiple instances of SQL Server — complete with separate logins and management rights — all on the same physical server.

I'm sure many of you are now asking: Can I have different versions of SQL Server on the same box — say, SQL Server 2000 and SQL Server 2005? The answer is, yes. You can mix SQL Server 2000 and 2005 on the same box. Personally, I am not at all trusting of this configuration, even for migration scenarios, but, if you have the need, yes, it can be done.

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)

In addition to the system installed examples, this book makes extensive use of the older samples. (See Appendix F — online for more info on how to get these installed.)

- ☐ pubs
- ☐ Northwind

During the design of this book, much debate was had over whether to use the newer examples or stick with the tried and true older examples. I'm going to be very up front that Microsoft was not very happy about my choice to retain the older examples, but I'm not making any apologies about it.

The newer AdventureWorks database is certainly a much more robust example, and does a great job of providing examples of just about every little twist and turn you can make use of in SQL Server 2005. There is, however, a problem with that — complexity. The AdventureWorks database is excessively complex for a training database. It takes features that are likely to be used only in exception cases and uses them as a dominant feature. I polled several friends who teach and/or write books on SQL Server, and all of them shared my opinion on this: Northwind and pubs, while overly simplistic in many ways, make it relatively easy to understand the basic concepts at work in SQL Server. I'd much rather get you to understand the basics and move forward than overwhelm you in the unnecessary complexity that is AdventureWorks.

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

Chapter 1

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. They can enable you to determine whether certain objects exist before you perform operations on them. For example, if you try to create an object that already exists in any particular database, you will get an error. If you want to force the issue, you could test to see whether the table already has an entry in the `sysobjects` table for that database. If it does, you would delete that object before re-creating it.

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 meta data that is stored in the system tables.

All that said, there are still times when nothing else will do. We will discuss a few situations where you can't avoid using the system tables, but in general, you should consider them to be evil cannibals from another tribe and best left alone.

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 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% 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. Not only are the objects within it temporary; 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 here and there — more as you get to some of the more advanced features of the product.

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 it.

The pubs Database

Ahhhh `pubs`! It's almost like an old friend. `pubs` is now installed only as a separately downloaded sample from the Microsoft website and is available primarily to support training articles and books like this one. `pubs` has absolutely nothing to do with the operation of SQL Server. It's merely there to provide a consistent place for your training and experimentation. You make use of `pubs` occasionally in this book.

`pubs` can be installed, although it is a separate install, and deleted with no significant consequences.

The Northwind Database

If your past programming experience has involved Access or Visual Basic, = you are probably already somewhat familiar with the `Northwind` database. `Northwind` was new to SQL Server beginning in version 7.0, but is being removed from the basic installation as of SQL Server 2005. Much like `pubs`, it must be installed separately from the base SQL Server install. (Fortunately, it's part of the same sample download and install). The `Northwind` database serves as one of the major testing grounds for this book.

`pubs` and `Northwind` are only installed as part of a separate installation that can be downloaded from Microsoft. See Appendix F (online) for more information on how to get them installed on your practice system.

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 then writes several of them to the physical database file(s) at a time.

We'll take a much closer look at how things are logged in Chapter 14, "Transactions and Locks," 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.

RDBMS Basics: What Makes Up a SQL Server Database?

Let's take a look at the `publishers` table in the `pubs` database. (The view presented in Figure 1-1 is from the SQL Server Management Studio. This is a fundamental tool and we will look at how to make use of it in the next chapter.)

pub_id	pub_name	city	state	country
0736	New Moon Books	Boston	MA	USA
0877	Binnet & Hardley	Washington	DC	USA
1389	Algodata Infosy...	Berkeley	CA	USA
1622	Five Lakes Publis...	Chicago	IL	USA
1756	Ramona Publishers	Dallas	TX	USA
9901	GGG&G	München	NULL	Germany
9952	Scootney Books	New York	NY	USA
9999	Lucerne Publishing	Paris	NULL	France

Figure 1-1

The table in Figure 1-1 is made up of five columns of data. The number of columns remains constant regardless of how much data (even zero) is in the table. Currently, the table has eight records. The number of records will go up and down as we add or delete data, but the nature of the data in each record (or row) is described and restricted by the *data type* of the column.

I'm going to take this as my first opportunity to launch into a diatribe on the naming of objects. SQL Server has the ability to embed spaces in names and, in some cases, to use keywords as names. Resist the temptation to do this! Columns with embedded spaces in their name have nice headers when you make a `SELECT` statement, but there are other ways to achieve the same result. Using embedded spaces and keywords for column names is literally begging for bugs, confusion, and other disasters. I'll discuss later why Microsoft has elected to allow this, but for now, just remember to associate embedded spaces or keywords in names with evil empires, torture, and certain death. (This won't be the last time you hear from me on this one.)

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 were after.

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.

Chapter 1

Note that views that have indexes — or *indexed views* — must have at least one clustered index before it 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.

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 normalization and 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. See Appendix C for more on ERD and other tools.

Figure 1-2 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

RDBMS Basics: What Makes Up a SQL Server Database?

between two tables. We'll talk about relationships extensively in Chapters 7 and 8 and we'll look further into diagrams later in the book.

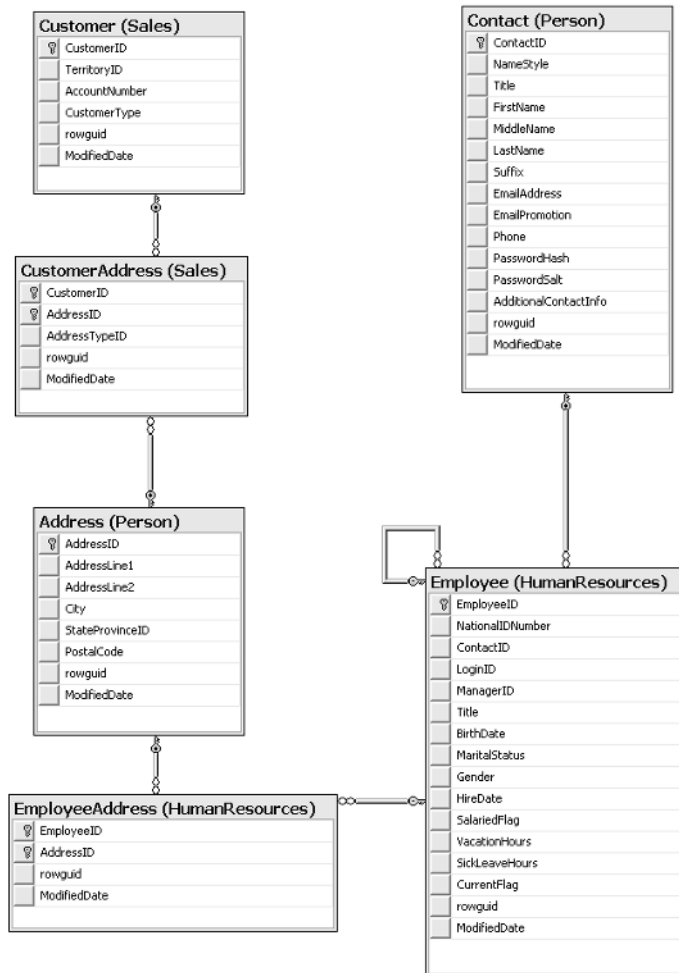


Figure 1-2

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.

Chapter 1

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.

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 couple of performance impacts (some positive, one negative):

- ❑ 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 10.

Stored Procedures

Stored procedures (or *sprocs*) are historically and, even in the .NET era, likely to continue 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, therefore 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 13.

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 login into 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.

Rules should be considered there for backward compatibility only and should be avoided in new development.

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 as 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 6.

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 tied at the hip to the tables and columns that they map, they are separate objects, and are therefore 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 <code>bit</code> 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 quintrillion.
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.

RDBMS Basics: What Makes Up a SQL Server Database?

Data Type Name	Class	Size in Bytes	Nature of the Data
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, Float (20)) that determines size and precision. Note that the argument is in bits, not bytes. Ranges from -1.79E + 308 to 1.79E + 308.
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 only takes up 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.)
UniqueIdentifier	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.

Table continued on following page

Chapter 1

Data Type Name	Class	Size in Bytes	Nature of the Data
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).
Ntext	Unicode	Varies	Like the Text data type, this is legacy support only. In this case, use <code>nvarchar(max)</code> . 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 <code>max</code> 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 <code>varbinary(max)</code> 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.

Data Type Name	Class	Size in Bytes	Nature of the Data
Sql_variant	Other	Special	This is loosely related to the <code>Variant</code> 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.

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. 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 specifically say what type you want to convert to.) A few can't be converted between at all. Figure 1-3 contains a chart that shows the various possible conversions:

To:\nFrom:	binary	varbinary	char	varchar	nchar	nvarchar	datetime	smalldatetime	decimal	numeric	float	real	bigint	int(INT4)	smallint(INT2)	tinyint(INT1)	money	smallmoney	bit	timestamp	uniqueidentifier	image	ntext	text	sql_variant	xml
binary																										
varbinary	●										○	○												○	○	
char	●	●																		●				○	○	
varchar	●	●																		●				○	○	
nchar	●	●	○																	●		○		○	○	
nvarchar	●	●	○																	●		○		○	○	
datetime	●	●				○						●			●	●	●	●		●			○	○	○	
smalldatetime	●	●							●			●			●	●	●	●		●						
decimal	○	○	○	○	○	○			*	*													○	○	○	
numeric	○	○	○	○	○	○			*	*													○	○	○	
float	○	○	○	○	○	○																	○	○	○	
real	○	○	○	○	○	○																		○	○	
bigint	○	○	○	○	○	○																		○	○	
int(INT4)	○	○	○	○	○	○																		○	○	
smallint(INT2)	○	○	○	○	○	○																		○	○	
tinyint(INT1)	○	○	○	○	○	○																		○	○	
money	○	○	○	○	○	○																		○	○	
smallmoney	○	○	○	○	○	○																		○	○	
bit	○	○	○		○	○												○							○	
timestamp	○	○	○		○	○															○				○	
uniqueidentifier	○	○	○	○	○	○																		○	○	
image	○	○	○	○	○	○																		○	○	
ntext	○	○	○	○	○	○			○	○	○	○					○	○	○	○				○	○	
text	○	○	○	○	○	○			○	○	○	○					○	○	○	○			○	○	○	
sql_variant	●	●	●	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○			○	○	
xml	●	●	●	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	

● Explicit conversion

● Implicit conversion

○ Conversion not allowed

★ Requires explicit CAST to prevent the loss of precision or scale that might occur in an implicit conversion

● Implicit conversions between XML datatypes are supported only if the source or target is untyped xml. Otherwise, it has to be explicit.

Figure 1-3

Why would we have to convert a data type? Well, let’s try a simple example. If I wanted to output the phrase, “Today’s date is ##/##/####”, where ##/##/#### is the current date, I could write it like this:

```
SELECT 'Today''s date is ' + GETDATE()
```

We will discuss Transact-SQL statements such as this in much greater detail later in the book, but the expected results of the previous example should be fairly obvious to you.

The problem is that this statement would yield the following result:

```
Msg 241, Level 16, State 1, Line 1
Syntax error converting datetime from character string.
```

Not exactly what we were after, is it? Now let’s try it with the CONVERT() function:

```
SELECT "Today's date is " + CONVERT(varchar(12), GETDATE(),101)
```

This yields something like:

```
-----  
Today's date is 01/01/2000  
  
(1 row(s) affected)
```

Date and time data types, such as the output of the `GETDATE()` function, aren't implicitly convertible to a string data type, such as "Today's date is ", yet we run into these conversions on a regular basis. Fortunately, the `CAST` and `CONVERT()` functions enable us to convert between many SQL Server data types. We will discuss the `CAST` and `CONVERT()` functions more in a later chapter.

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 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. You've even heard my first soapbox diatribe on column names. 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

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

As I mentioned earlier in the chapter, 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 than “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.
- ❑ 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 only acceptable as a delimiter for column names if you have SET QUOTED_IDENTIFIER ON. Using square brackets ([and]) avoids the chance that your users will have the wrong setting.

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

Tools of the Trade

Now that we know something about the many types of objects that exist on SQL Server, we probably should get to know something about how to find these objects, and how to monitor your system in general.

In this chapter, we will look into the tools that SQL Server has to offer. Some of them offer only a small number of highly specialized tasks; others do many different things. Most of them have been around in SQL Server for a long time. One thing is certain: 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, and, for people just starting out, I would say they have largely succeeded. (Of course, stodgy old SQL Server users like me find ourselves saying, “where is everything?”)

The tools we will look at in this chapter will be:

- ☐ SQL Server Books Online
- ☐ The SQL Server Computer Manager
- ☐ SQL Server Management Workbench
- ☐ SQL Server Integration Services (SSIS), including the Import/Export Wizard
- ☐ The Database Engine Tuning Advisor
- ☐ The Report Manager
- ☐ The Bulk Copy Program (bcp)
- ☐ Profiler
- ☐ sqlcmd

Be careful if you're getting help from friends that may be experienced, but are using SQL Server 2000 or an older version rather than SQL Server 2005. The toolset received a very major rework for this release, and many of the things that the "old timers" of SQL Server are used to and have names for have been moved, had their names changed, or were eliminated in favor of integration with another tool.

Most all of the concepts are still in there somewhere, but may have moved around.

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.

My general philosophy about books or any other reference material related to programming is that I can't have enough of it. I first began programming in 1980 or so, and back then it was possible to remember most things (but not everything). Today, it's simply impossible. If you have any diversification at all (something that is, in itself, rather difficult these days), there are just too many things to remember, and the things you don't use everyday get lost in dying brain cells.

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

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, and Visual Studio):

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 deselect 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 having that quick reference available wherever you are running SQL Server can save you a fortune in time. (On my machine, Books Online takes up 100MB of space.)

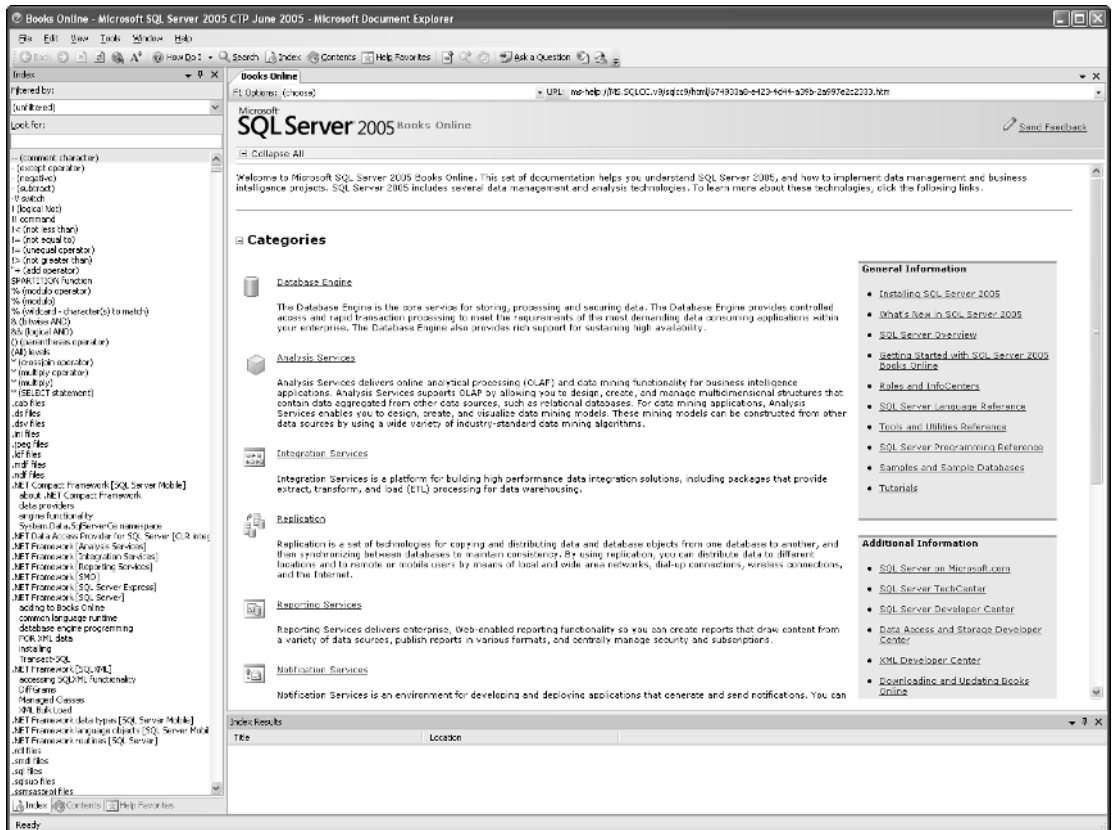


Figure 2-1

The SQL Server Configuration Manager

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

The SQL Server Computer 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

SQL Server is a large product, and the various pieces of it utilize a host of services that run in the background on your server. A full installation will encompass seven services, and these can all be managed from this part of the SQL Server Computer Manager.

The services available for management here include:

- ❑ **Analysis Services** — This powers the Analysis Services engine.
- ❑ **Full Text** — Again, just what it sounds like. it powers the Full Text Search Engine
- ❑ **Report Server** — The underlying engine that supports 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 to them and can even branch into 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. These are dynamic-link libraries (DLLs) that SQL Server uses to communicate with certain network protocols. NetLibs serve as something of an insulator between your client application and the network protocol, which is essentially the language that one network card uses to talk to another, 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

VIA is a special net library that is made for use with some very special (and expensive) hardware. If you're running in a VIA environment, you'll know about the special requirements associated with it. For those of you that aren't running in that environment, it suffices to say that VIA offers a very fast but expensive solution to high-speed communication between servers. It would not usually be used for a normal client.

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.

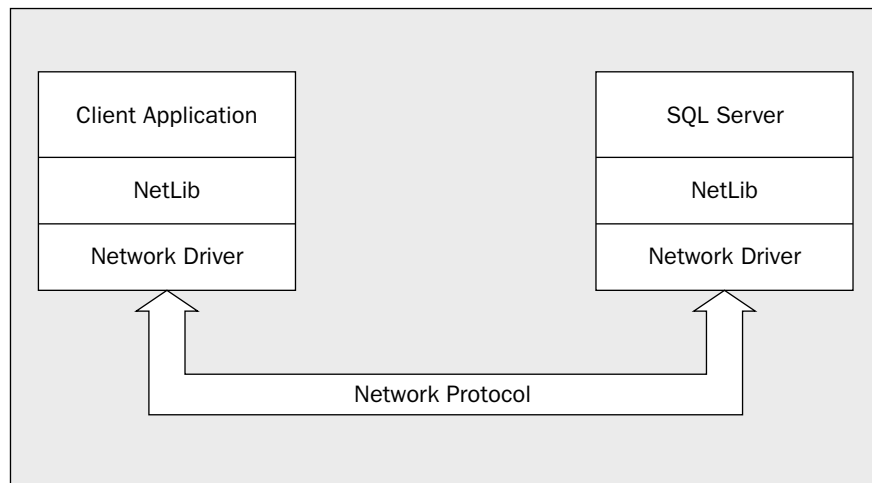


Figure 2-2

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

In case you're familiar with TCP/IP, the default port that the IP NetLib will listen on is 1433. A port can be thought of as being like a channel on the radio — signals are bouncing around on all sorts of different frequencies, but they only do you any good if you're "listening" on the right channel.

The Protocols

Let's start off with that "What are the available choices?" question. If you run the Computer Management Utility and open the Server Network Configuration tree, you'll see something like Figure 2-3:

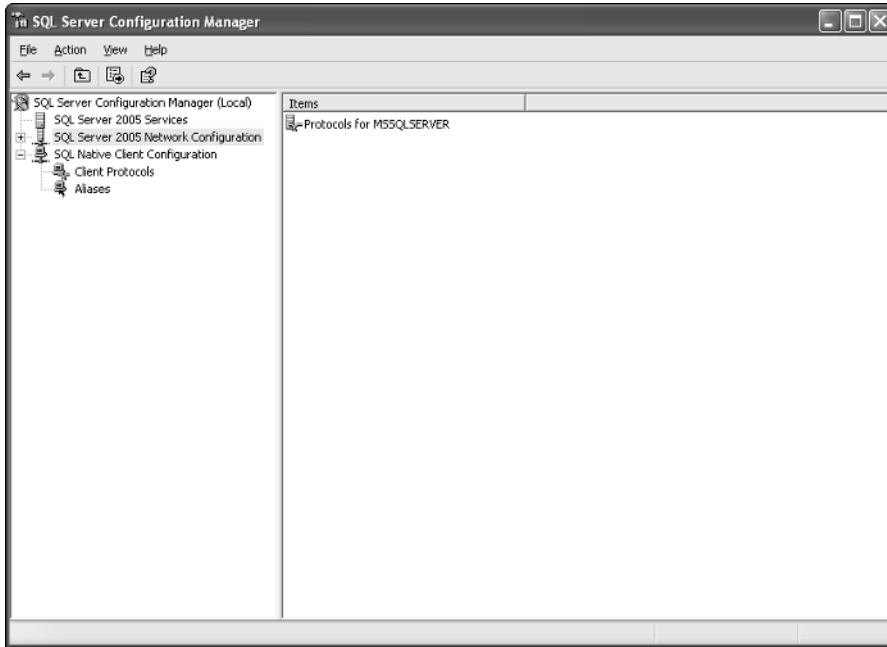


Figure 2-3

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

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

Now let's see what our server *could* be listening for by expanding the Protocols for MSSQLSERVER tree under Server Network Configuration, as shown in Figure 2-4:

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 chose enable.

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?).

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

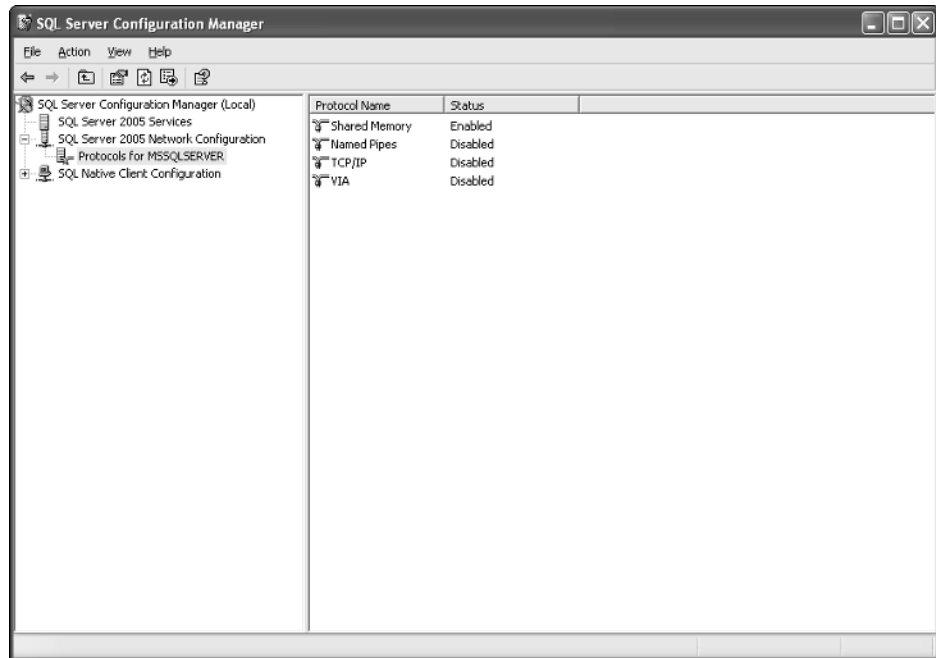


Figure 2-4

Named Pipes

Named Pipes can be very useful when TCP/IP is not available, or there is no Domain Name Service (DNS) server to allow the naming of servers under TCP/IP.

Technically speaking, you can connect to a SQL Server running TCP/IP by using its IP address in the place of the name. This works all the time, even if there is no DNS service, as long as you have a route from the client to the server. (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 only IP.

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 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 — 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's only useful 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.

On to the Client

Now, we've seen all the possible protocols and we know how to choose which ones to offer. 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, but let's take a look at what we've got. Expand the Client Network Configuration tree and select the Client Protocols node, as shown in Figure 2-5:

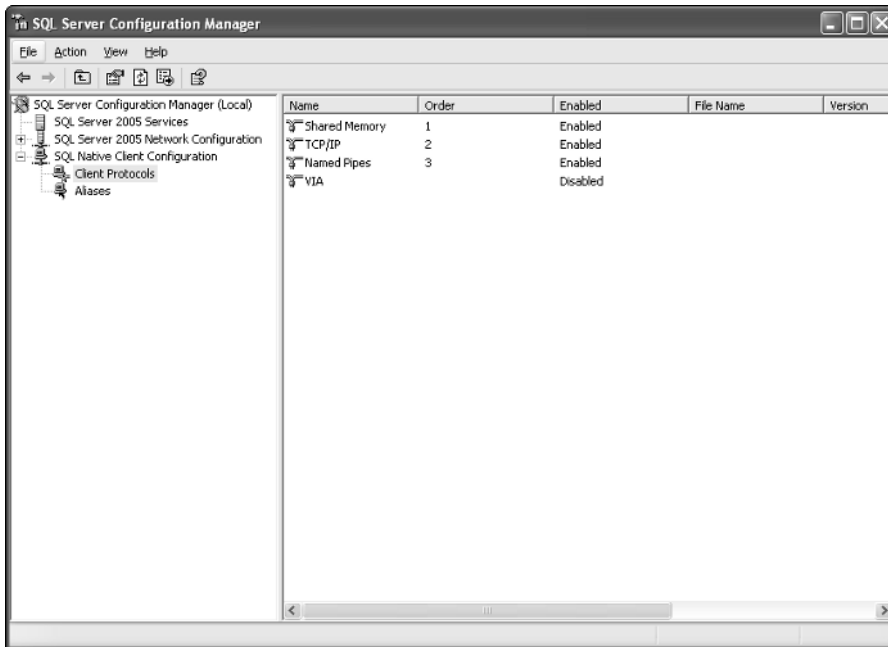


Figure 2-5

Beginning with SQL Server 2000, Microsoft added the ability for the client to start with one protocol, then, if that didn't work, move on to another. In the dialog above, 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 is 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 on which 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. Figure 2-6 shows a client configured to use the Named Pipes NetLib for requests from the server named `ARISTOTLE`, and to use whatever is set up as the default for contact with any other SQL Server:

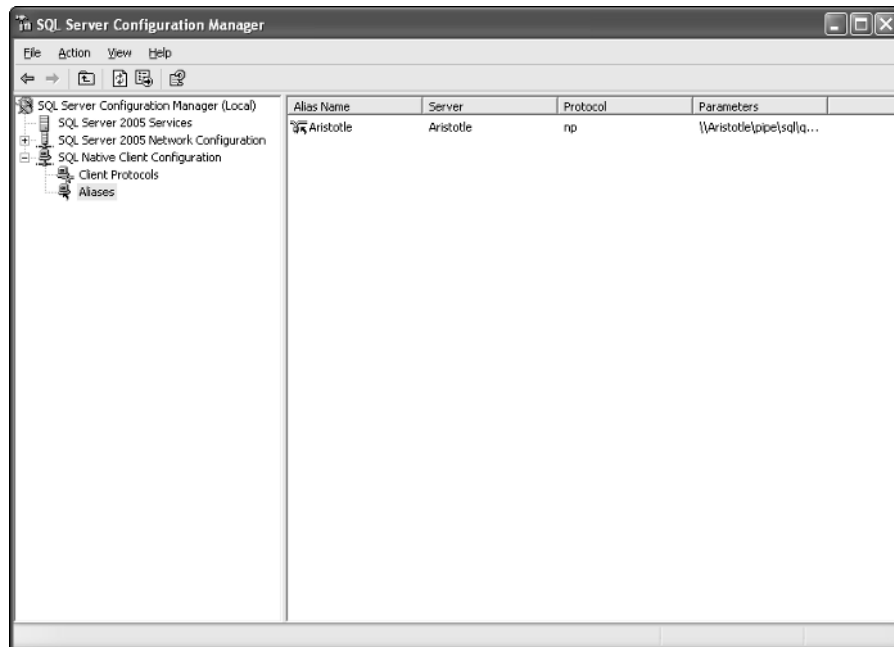


Figure 2-6

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

If you are connecting to your SQL Server over the Internet (which is a very bad idea from a security standpoint, but people do it), you'll probably want to use the Server's actual IP address rather than the name of the server. This gets around some name resolution issues that may occur when dealing with SQL Server and the Internet. Keep in mind, however, that you'll need to change the IP address manually if the server gets a new IP; you won't be able to count on DNS to take care of it for you.

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.

The 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 take a quick run down 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 publishing and subscribing databases for replication

We will be seeing a great deal of the 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 the Management Studio, you are presented with a Connection dialog box similar to the one in Figure 2-7:



Figure 2-7

Your login screen may look a little bit different from this, depending on whether you've logged in before, which machine you logged into, and what login name you used. Most of the options on the login screen are pretty self-descriptive, but let's look at a couple in more depth.

Server Type

This relates to which of the various subsystems 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. In our illustration, we have chosen (local). This doesn't mean that there is a server named (local), but rather that we want to log into the default instance of SQL Server that is on this same machine, regardless of what this machine is named. Selecting (local) not only automatically identifies which server (and instance) you want to use, but also how you're going to get there. You can also use a simple period (.) as a shortcut for (local).

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 or ARISTOTLE in many of the examples in this book) followed by a dollar sign, and the instance name, for example, ARISTOTLE\$POMPEII.

If you select (local), your system uses the shared memory NetLib regardless of which NetLib you 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 which 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, 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.

Now, what if you can't remember what the server's name is? Just click the down arrow to the right of the server box to get a list of recently connected servers. If you scroll down, you'll see a <Browse for more...> option. If you choose this option, SQL Server will poll the network for any servers that are "advertising" to the network; essentially, this is a way for a server to let itself be known to other systems on the network. You can see from Figure 2-8 that you get two tabs: one that displays local servers (all of the instances of SQL Server on the same system you're running on), and another that shows other SQL Servers on the network:

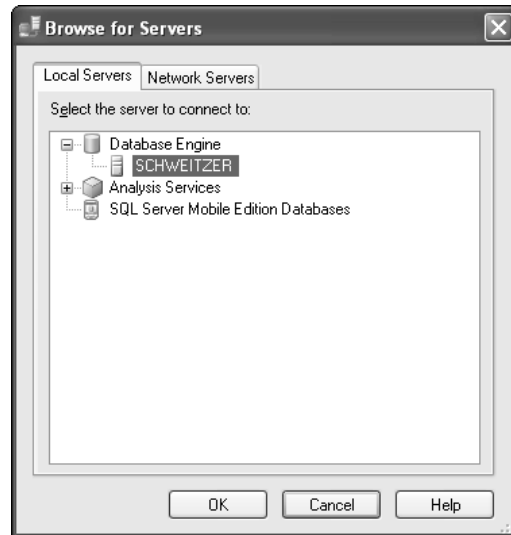


Figure 2-8

You can select one of these servers and click OK.

Watch out when using the Server selection dialog box. Although it's usually pretty reliable, there are ways of configuring a SQL Server so that it doesn't broadcast. When a server has been configured this way, it won't show up in the list. Also, servers that are only listening on the TCP/IP NetLib and don't have a DNS entry will not show up. You must, in this case, already know your IP address and refer to the server using it.

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 only acceptable to the system if you have specifically turned SQL Server Authentication on.

Windows Authentication

Windows authentication is just as it sounds. You have Windows 2000+ users and groups. 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 don't have to fill in anything to log in; it just takes the login information from the way you're currently logged into the Windows network. Additionally, the administrator has to administer users in only one place. The downside is that mapping this process can get complex and, to administer the Windows user side of things, you must 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 explicitly set up in SQL Server. The authentication process doesn't take into account the current network login at all; instead, the user provides a SQL Server-specific login and password.

This can be nice because the administrator for a given SQL Server doesn't need to be a domain administrator (or even have a username on your network for that matter) 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.

Try It Out Making the Connection

Let's get logged on. If you are starting up your SQL Server for the first time, set everything just as it is in our example screen.

1. Choose the (local) option for the SQL Server.
2. Select SQL Server authentication.
3. Select a Login name of **sa**, which stands for System Administrator and remember it. Alternatively, you may log in as a different user as long as that user has system administrator privileges.
4. Enter the sa password that was set to when you installed SQL Server. On case-sensitive servers, the login is also case-sensitive, so make sure you enter it in lowercase.

If you're connecting to a server that has been installed by someone else, or where you have changed the default information, you need to provide login information that matches those changes. After you click OK, you should see the main `Query` window screen shown in Figure 2-9:

Be careful with the password for the sa user. This and any other user who is a sysadmin is a super-user with full access to everything.

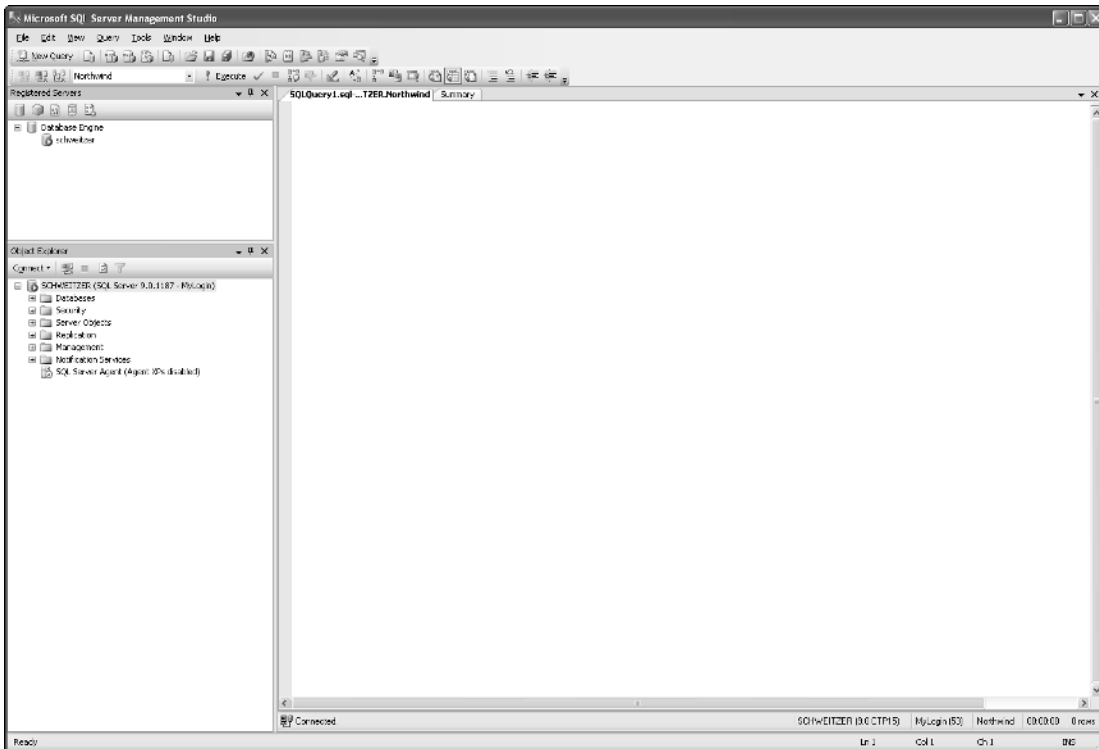


Figure 2-9

How It Works

The Login dialog gathers all the information needed to create the connection. Once it has that, it assembles the connection information into a single *connection string*, and sends that to the server. The connection is then either accepted or rejected, and, if it is accepted, a connection handle is given to the *Query* window so that the connection can be used over and over again for many queries as long as you do not disconnect.

You see more about how connection strings are created and formatted in Appendix X.

Again, many of the items here (New, Open, Save, Cut, Paste, and so on) are things that you have seen plenty of times in other Windows applications, and should be familiar with, but there's also a fair amount that's specific to SQL Server. The main thing to notice for now is that the menus in the management studio are context sensitive—that is, you'll see a change in what menus are available and what they contain based on what window is active in the studio. Be sure to explore the different context menus you get as you explore different parts of the Management Studio.

Query Window

This part of the Management Studio takes the place of a separate tool in previous versions that was called *Query Analyzer*. It's your tool for interactive sessions with a given SQL Server. It's 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's 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 only support ANSI to entry-level.

Personally, I'm not all that thrilled with this new version of the tool; I find that, due to the number of things this one tool does, the user interface is cluttered and it can be hard to find what you're looking for. That said, Microsoft is hoping that new users will actually find it more intuitive to use within the larger Management Studio.

Because the *Query window* is where we will spend a fair amount of time in this book, let's take a more in-depth look at this tool and get familiar with how to use it.

Getting Started

Well, I've been doing plenty of talking about things in this book, and it's high time we started doing something. To that end, open a new *Query window* by clicking the New Query button towards the top-left of the Management Studio, or choosing File→New→New Query With Current Connection from the File menu. When the *Query window* opens, you'll get menus that largely match those in *Query Analyzer* back when that was a separate tool. We'll look at the specifics, but let's get our very first query out of the way.

Type the following code into the main window of the *Query window*:

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

Notice the coloring of words and phrases as you type them into the code window. 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 represents another simple debugging item, which quickly parses the query for you without actually attempting to run the statement. If there are any syntax errors, this should catch them before you see error messages. A debugger is available as another way to find errors. We'll look at that in depth in Chapter 12, "Stored Procedures."

Now click the execute button — with the red exclamation point next to it — on the toolbar. The *Query window* changes a bit, as shown in Figure 2-10:

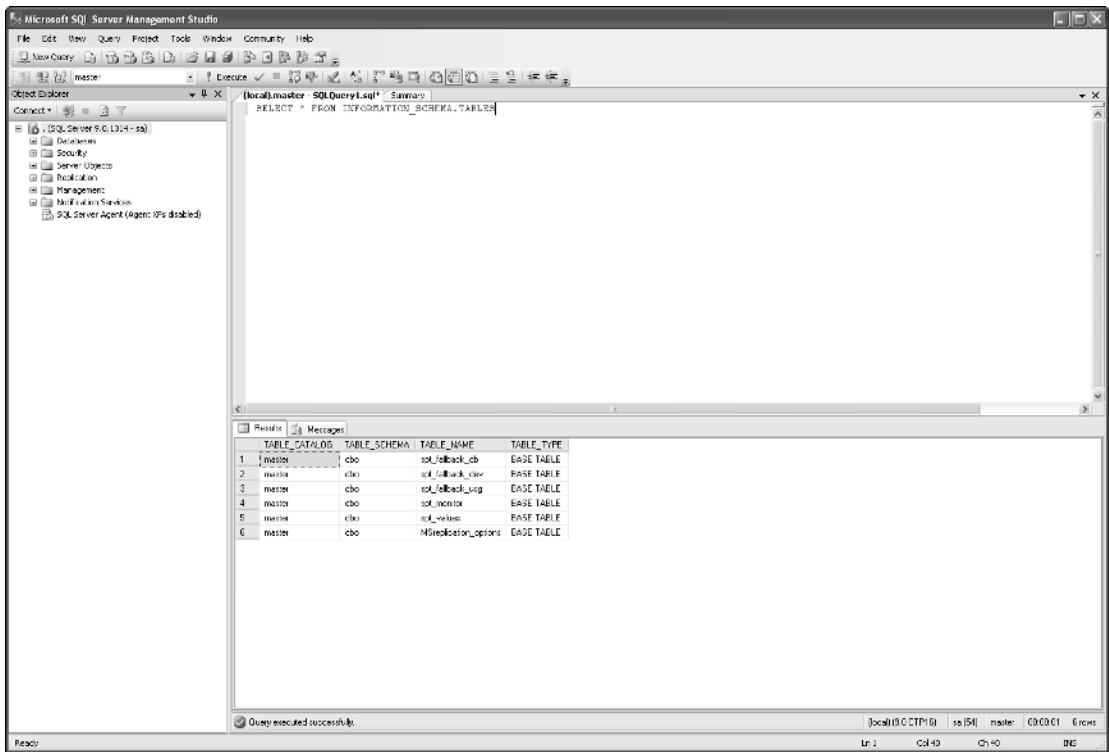


Figure 2-10

Notice that the main window has been automatically divided into two panes. The top is your original query text; the bottom is called the *results pane*. In addition, notice that the results pane has a tab at the top of it. Later on, after we’ve run queries that return multiple sets of data, you’ll see that we can get each of these results on separate tabs; this can be rather handy because you often don’t know how long each set of data, or *result set*, is.

The terms result set and recordset are frequently used to refer to a set of data that is returned as a result of some command being run. You can think of these words as interchangeable.

Now, 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 a set of three icons, highlighted in Figure 2-11:

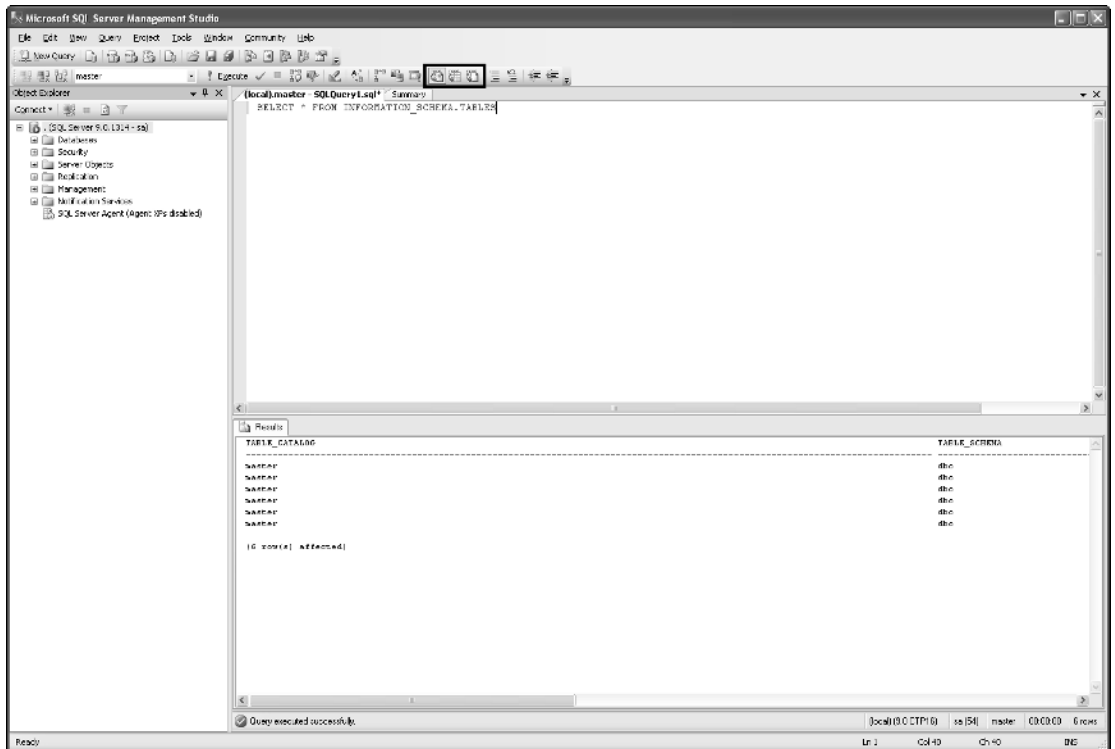


Figure 2-11

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 submenu.

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 only by the available memory in your system).

Before discussing this further, rerun that last query using this option and see what you get. Choose the Results in Text option and rerun the previous query by clicking the green arrow, as shown in Figure 2-12:

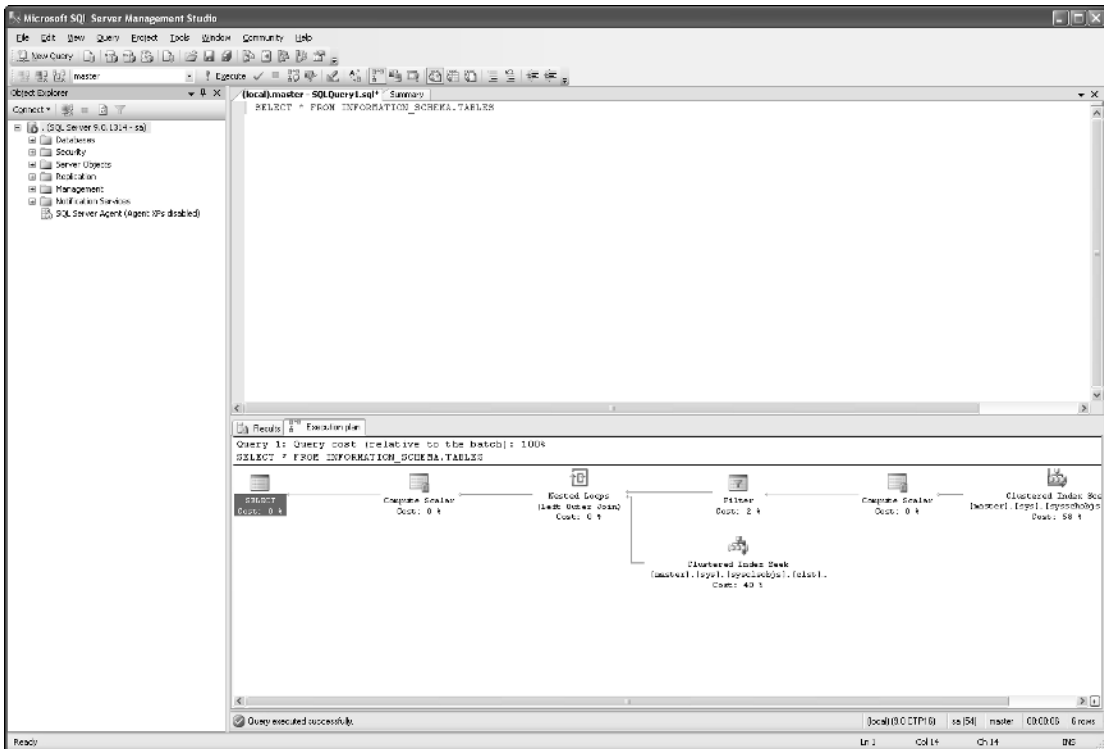


Figure 2-12

The data that you get back is exactly the same as before. It's just given to you in a different format. I use this output method in several scenarios:

- ☐ When I'm only getting one result set and the results have only fairly narrow columns
- ☐ When I want to be able to 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 option divides the columns and rows into a grid arrangement. Following is a list of specific things that this option gives you that the Results in Text doesn't:

- ❑ 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 the right border results in the autofit 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 is 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 the inner rows have every column selected; you can select only in the middle of the row for the first and last row selected).

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

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 the best way to run your query to balance fast results with minimum impact to other users. When you use the Show Estimated Execution Plan option, you receive a graphical representation and additional information about 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 you will occasionally see differences here due to changes that the optimizer decided to make while it was running the query as well as 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 the Include Actual Execution Plan option, and execute the query again, as shown in Figure 2-13:

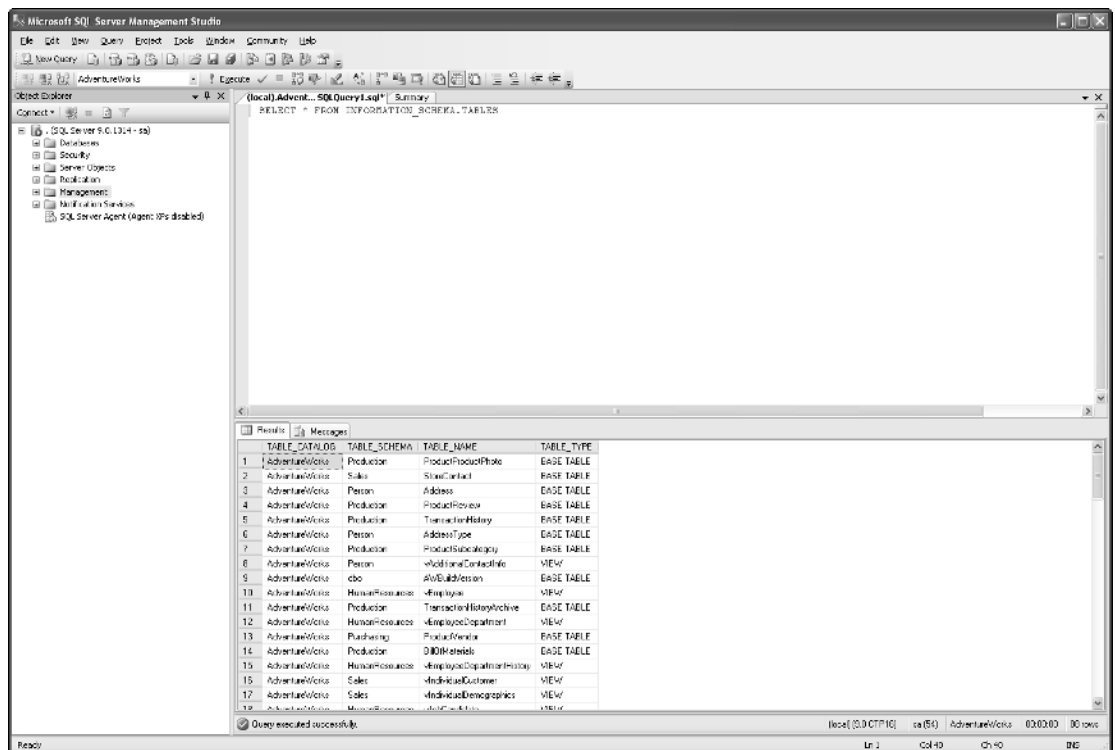


Figure 2-13

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

- ❑ You get the plan immediately rather than after your query executes.
- ❑ Although 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 is physically executed and the cost information you get is actual rather than estimated.

The DB Combo Box

Finally, 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. Since we’re using the sa user ID, every database on the current server should have an entry in the DB combo box.

Let’s change our current database to AdventureWorks and re-run the query, as shown in Figure 2-14:

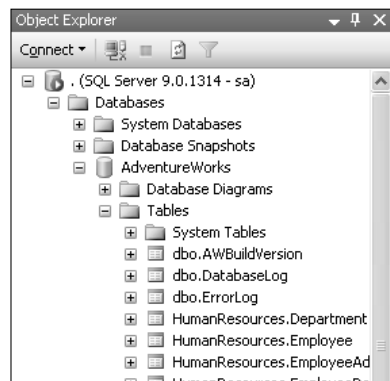


Figure 2-14

As you can see, the data has changed to represent the data from the newly queried database.

The Object Explorer

This useful little tool enables you to navigate your database, look up object names, and even perform actions like scripting and looking at the underlying data.

The database node can be extended all the way down to the listing of tables in the AdventureWorks Database. You can drill down even farther to see individual columns, including datatype and similar properties, of the tables—a very handy tool for browsing your database.

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. To give you a touch of perspective here, I’ve done a couple of Decision Support Systems (DSS) projects over the years. (These are

usually systems that don't have online data going in and out, but instead pull data together to help management make decisions.) A DSS project gathers data from a variety of sources and pumps it into one centralized database to be used for centralized reporting.

These projects can get very expensive very quickly, as they attempt to deal with the fact that not every system calls what is essentially the same data by the same name. There can be an infinite number of issues to be dealt with. These can include data integrity (what if the field has a `NULL` and we don't allow `NULLS`?) or differences in business rules (one system deals with credits by allowing a negative order quantity, another doesn't allow this and has a separate set of tables to deal with credits). The list can go on and on, so can the expense.

With SSIS, a tremendous amount of the coding, usually in some client-side language, that had to be done to handle these situations can be eliminated or, at least, simplified. SSIS enables 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.

Be aware that there is a special OLE DB provider for ODBC. This provider allows you to map your OLE DB access directly to an ODBC driver. That means anything that ODBC can access can also be accessed by OLE DB (and, therefore, SSIS).

While we're at it, it's also worth pointing out that SSIS, although part of SQL Server, can work against any OLE DB source and any OLE DB destination. That means that SQL Server doesn't need to be involved in the process at all other than to provide the data pump. You could, for example, push data from Oracle to Excel, or even DB/2 to MySQL.

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 example I gave earlier of taking data from a field that allows nulls and moving it to a table that doesn't allow nulls. With SSIS, you can automatically change 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.)

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 really appreciate when we do.

bcp is a command-line program that's 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 for 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.

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) tends to use everyday, 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 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 1000 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.

This is the tool to use when you want to include SQL commands and management tasks in command-line batch files. Prior to version 7.0 and the advent of what was then called DTS (now SSIS), sqlcmd was often used in conjunction with the Bulk Copy Program (bcp), to manage the import of data from external systems. This type of use is decreasing as administrators and developers everywhere learn the power and simplicity of SSIS. Even so, 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 use files that contain the scripts you want to run under sqlcmd. Keep in mind, however, that there are usually tools that can accomplish what sqlcmd can 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, just 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 aren't ones you'll use every day. Indeed, for the average developer, only SQL Server Management Studio will get daily use. Nevertheless it's 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

The Foundation Statements of T-SQL

At last! We've finally disposed of the most boring stuff. It doesn't get any worse than basic objects and tools, does it? Unfortunately, we have to lay down a foundation before we can build the house. The nice thing is that the foundation is now down. Having used the clichéd example of building a house, I'm going to turn it all upside down by talking about the things that let you enjoy living in it before we've even talked about the plumbing. You see, when working with databases, you have to get to know how data is going to be accessed before you can learn all that much about the best ways to store it.

In this chapter, we will discuss the most fundamental *Transact-SQL* (or *T-SQL*) statements. 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 going to be using T-SQL, and T-SQL remains our core language for doing things in SQL Server.

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*. Because 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), you'll find that these will become like old friends in no time at all.

In addition, SQL provides for many operators and keywords that help refine your queries. We'll learn some of the most common of these 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 vs. 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.

Getting Started with a Basic SELECT Statement

If you haven't used SQL before, or don't really feel like you've really understood it yet — pay attention here! 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)>]
[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])]
```

Wow — that's a lot to decipher, so let's look at the parts.

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. What we are selecting is identified by an expression or column list immediately following the SELECT — you'll see what I mean by this in a moment.

Next, we add in more specifics, such as from where we are getting this data. 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 ***** may seem odd, but it actually 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 meta data 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 meta data of your system have been put there so you won't have to use what are called "system tables."

Try It Out The SELECT Statement

Let's play around with this some more. Change the current database to be the AdventureWorks database. Recall that, to do this, you need only select the AdventureWorks entry from the combo box in the toolbar at the top of the Query window in the Management Studio, as shown in Figure 3-1.

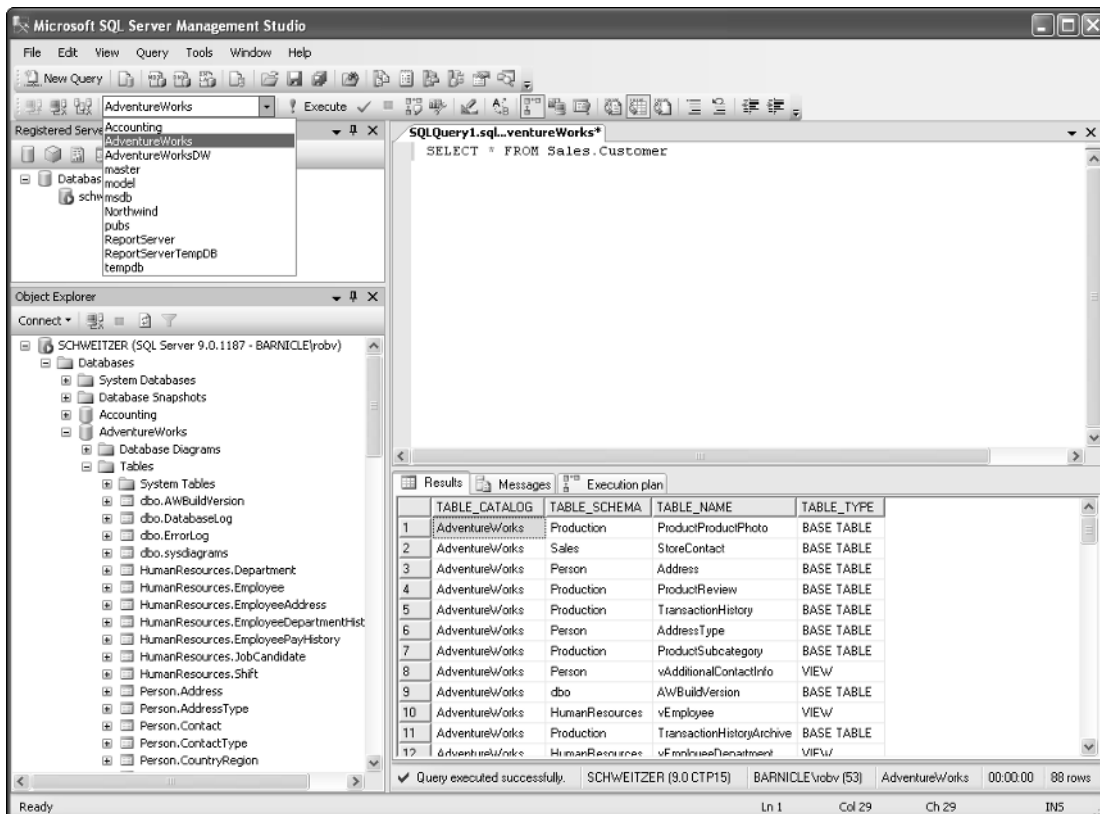


Figure 3-1

If you're having difficulty finding the combo box that lists the various databases, try clicking once in the Query window. The SQL Server Management Studio toolbars are context sensitive — that is, they will change by whatever it thinks is the current thing you are doing. If you don't have a Query window as the currently active window, you may have a different set of toolbars up (one that is more suitable to some other task). As soon as a Query window is active, it should switch to a set of toolbars that are suitable to query needs.

Now that we have the AdventureWorks database selected, let's start looking at some real data from our database. Try this query:

```
SELECT * FROM Sales.Customer
```

After you have that in the Query window, just click on Execute on the toolbar and watch SQL Server give you your results. This query will list every row of data in every column of the Sales.Customer table in the current database (in our case, AdventureWorks). If you haven't altered any of the settings on your system or the data in the AdventureWorks database before you ran this query, then you should see the following information if you click on the messages tab:

```
(19185 row(s) affected)
```

For a SELECT statement, the number shown here is the number of rows that your query returned.

How It Works

Let's look at a few specifics of our SELECT statement. Notice that I capitalized the SELECT and FROM. This is not a requirement of SQL Server — we could run them as SeLeCt and frOM and they would work just fine. I capitalized them purely for purposes of convention and readability. You'll find that many SQL coders will use the convention of capitalizing all commands and keywords, while using mixed case for table, column, and non-constant variable names. The standards you choose or have forced upon you may vary, but live by at least one rule — be consistent.

OK, time for my next soapbox diatribe. Nothing is more frustrating for the person that has to read your code or remember your table names than lack of consistency. When someone looks at your code or, more important, uses your column and table names, it shouldn't take him or her long to guess most of the way you do things just by experience with the parts that he or she has already worked with. Being consistent is one of those incredibly simple things that have been missed to at least some degree in almost every database I've ever worked with. Break the trend — be consistent.

The SELECT is telling the Query Window what we are doing and the * is saying what we want (remember that * = every column). Then comes the FROM.

A FROM clause does just what it says — that is, it defines the place from which our data should come. Immediately following the FROM will be the names of one or more tables. In our query, all of the data came from the table called Customer.

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:

```
SELECT LastName FROM Person.Contact
```

Your results should 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 19,972 rows there. Since the name of each customer is all that we want, that's all that we've selected.

*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.*

Let's try another simple query. How about:

```
SELECT Name FROM Production.Product
```

Again, assuming that you haven't modified the data that came with the sample database, SQL Server should respond by returning a list of 504 different products that are available in the AdventureWorks database:

```
Name
-----
Adjustable Race
Bearing Ball
BB Ball Bearing
...
...
Road-750 Black, 44
Road-750 Black, 48
Road-750 Black, 52
```

The columns that you have chosen right after your SELECT clause are known as the *SELECT list*. In short, the SELECT list is made up of the columns that you have requested to be output from your query.

The WHERE Clause

Well, things are starting to get boring again, aren't they? So let's add in 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 specified has been included in our results. Unrestricted queries such as these 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 US, you might provide them with a list that limits the domain of choices to just the fifty states. That way, you can be sure that the option selected will be a valid one. We will see this concept of domains further when we begin talking about database design.

Now we want to try looking for more specific information. We don't want a listing of product names — we want information on a specific product. Try this — see if you can come up with a query that returns 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.

Now, we'll take a look at the tables that are available. Since we've already used the Production.Product table once before, we know that it's there (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 3-2, you will see each of the columns along with its datatype and nullability options. Again, we'll see some other methods of finding this information a little later in the chapter.

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, save for the missing space in between the two words, just as easy to identify.

Therefore, 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. After all, we can look through a small list ourselves — right? But that's a pretty big "if" there. In any significant system, very few of your tables will have small record counts. You don't want to have to go scrolling through 10,000 records. What if you had 100,000 or 1,000,000? Even if you felt like scrolling through them all, the time before the results were back would be increasing dramatically. Finally, what do you do when you're designing this into your application and you need a quick result that gets straight to the point?

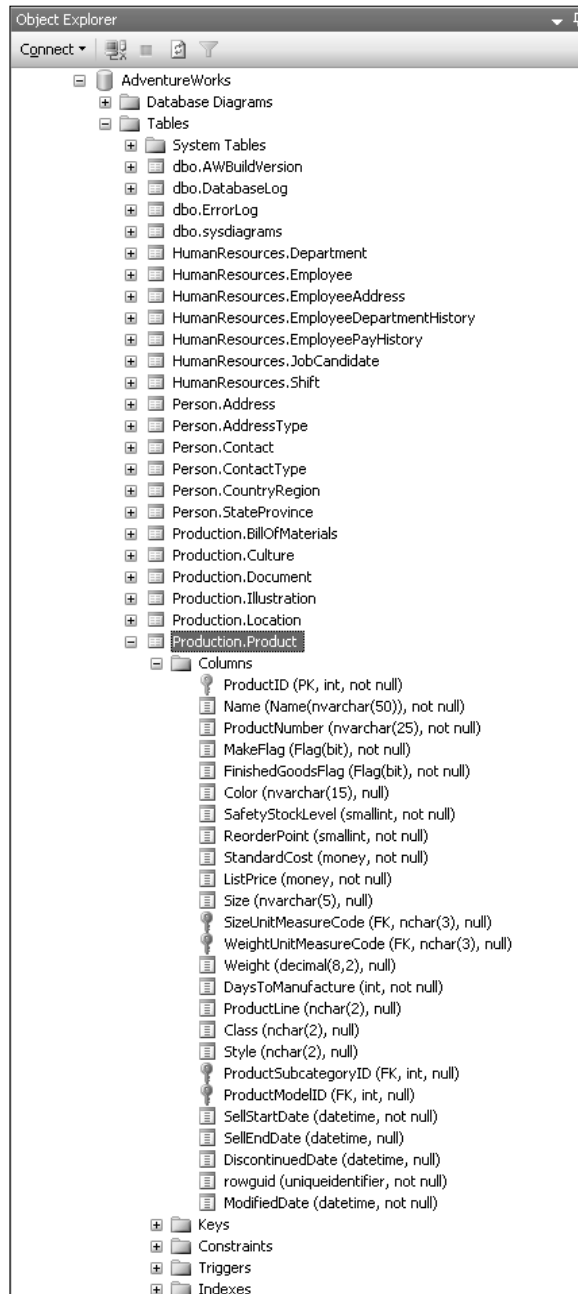


Figure 3-2

Chapter 3

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: 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.
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.

Operator	Example Usage	Effect
LIKE	<Column1> LIKE "ROM%"	Uses the % and _ characters for wildcarding. % indicates a value of any length can replace the % character. _ indicates any one character can replace the _ character. Enclosing characters in [] symbols indicates any single character within the [] is OK ([a-c] means a, b, and c are OK. [ab] indicates a or b are OK). ^ 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 16.
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 16.

ORDER BY

In the queries that we've run thus far, most of them have come out in something resembling alphabetical order. Is this by accident? It will probably come as somewhat of a surprise to you, but the answer to that is yes. If you don't say 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 how 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 SQL Server used to find your data.

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
```

Chapter 3

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. Were we to 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 therefore might sort 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.

We can also do our sorting using numeric fields. Let's try our first query against the Northwind database.

This will be the first time that we've utilized any of the databases that are not "built-in" to the installation routine. In order to run these next queries and many others scattered throughout the book, you will need to make sure that you have installed the Northwind and Pubs database samples from the SQL Server 2000 Samples Install, which is downloadable from Microsoft. How to install these samples is described in Appendix E.

We will be going between samples fairly interchangeably as the book progresses, and will eventually even experiment with utilizing data from more than one database at the same time.

We utilize this mix of databases for a variety of reasons, not the least of which is to expose you to databases that are designed with somewhat different approaches. This issue of difference in design approach will be addressed more fully in our design chapter.

This next query is still on a table containing products, but with a slight difference in name (notice the "s," and a different schema and database:

```
SELECT ProductID, ProductName, UnitsInStock, UnitsOnOrder
FROM Products
WHERE UnitsOnOrder > 0
AND UnitsInStock < 10
ORDER BY UnitsOnOrder DESC
```

This one results in:

ProductID	ProductName	UnitsInStock	UnitsOnOrder
-----	-----	-----	-----
66	Louisiana Hot Spiced Okra	4	100
31	Gorgonzola Telino	0	70
45	Rogede sild	5	70
21	Sir Rodney's Scones	3	40
32	Mascarpone Fabioli	9	40
74	Longlife Tofu	4	20
68	Scottish Longbreads	6	10

(7 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 multiple WHERE clause conditions and we also have an ORDER BY clause in place. 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.)

OK, 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.

Chapter 3

Suppose, for example, that we want to get a listing of every order that was placed between December 10 and 20 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 CustomerIDs sorted in descending order.

Our query would look like this:

```
SELECT OrderDate, CustomerID
FROM Orders
WHERE OrderDate BETWEEN '12-10-1996' AND '12-20-1996'
ORDER BY OrderDate, CustomerID DESC
```

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

OrderDate	CustomerID
-----	-----
1996-12-10 00:00:00.000	FOLKO
1996-12-11 00:00:00.000	QUEDE
1996-12-12 00:00:00.000	LILAS
1996-12-12 00:00:00.000	HUNGO
1996-12-13 00:00:00.000	ERNSH
1996-12-16 00:00:00.000	BERGS
1996-12-16 00:00:00.000	AROUT
1996-12-17 00:00:00.000	SPLIR
1996-12-18 00:00:00.000	SANTG
1996-12-18 00:00:00.000	FAMIA
1996-12-19 00:00:00.000	SEVES
1996-12-20 00:00:00.000	BOTTM

(12 row(s) affected)

Our dates, since we didn't say anything to the contrary, were still sorted in ascending order (the default), but, if you look at the 16th as an example, you can see that our CustomerIDs were indeed sorted last to first—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 that 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 OrderID, Quantity
FROM [Order Details]
WHERE OrderID BETWEEN 11000 AND 11002
```

Note the use of the square brackets in this query. Remember back from Chapter 2 that if an object name (a table in this case) has embedded spaces in it, we must delimit the name by using either square brackets or single quotes — this lets SQL Server know where the start and the end of the name is. Again, I highly recommend *against* the use of embedded spaces in your names in real practice.

This yields a result set of:

OrderID	Quantity
-----	-----
11000	25
11000	30
11000	30
11001	60
11001	25
11001	25
11001	6
11002	56
11002	15
11002	24
11002	40

(11 row(s) affected)

Even though we've asked only 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().

Chapter 3

```
SELECT OrderID, SUM(Quantity)
FROM [Order Details]
WHERE OrderID BETWEEN 11000 AND 11002
GROUP BY OrderID
```

This gets us what we were looking for:

```
OrderID
-----
11000      85
11001     116
11002     135

(3 row(s) affected)
```

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, and 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.

Let's say, for example, that we're looking for the number of orders each employee has taken for customers with CustomerIDs between A and AO. We can use both the EmployeeID and CustomerID columns in our GROUP BY (I'll explain how to use the COUNT() function shortly):

```
SELECT CustomerID, EmployeeID, COUNT(*)
FROM Orders
WHERE CustomerID BETWEEN 'A' AND 'AO'
GROUP BY CustomerID, EmployeeID
```

This gets us counts, but the counts are pulled together based on how many orders a given employee took from a given customer:

```
CustomerID      EmployeeID
-----
ALFKI           1           2
ANTON           1           1
ALFKI           3           1
ANATR           3           2
ANTON           3           3
ALFKI           4           2
ANATR           4           1
ANTON           4           1
ALFKI           6           1
ANATR           7           1
ANTON           7           2

(11 row(s) affected)
```

Note that, once we use a GROUP BY, every column in the SELECT list has to either be part of the GROUP BY or it must be an aggregate. 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 queries above, 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. A wide range of aggregates is 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 a specific column.

AVG

This one is for computing averages. Let's try running that same query we ran before, but now we'll modify it to return the average quantity per order rather than the total for each order:

```
SELECT OrderID, AVG(Quantity)
FROM [Order Details]
WHERE OrderID BETWEEN 11000 AND 11002
GROUP BY OrderID
```

Notice that our results changed substantially:

```
OrderID
-----
11000      28
11001      29
11002      33

(3 row(s) affected)
```

You can check the math—on order number 11,000 there were three line items totaling 85 altogether. $85 \div 3 = 28.33$. I can just hear some of you out there squealing right now. You're probably saying something like, "Hey, if it's 28.33, then why did it round the value to 28?" Good question. The answer lies in the rules of *casting*.

Casting is covered in more detail later, but suffice to say that it is a process where the system automatically converts between different datatypes. In this case, the numbers the system started with were integers, so it made sure to give you an integer back (thus losing the decimal data).

Chapter 3

MIN/MAX

Bet you can guess these two. Yes, these grab the minimum and maximum amounts for each grouping for a selected column. Again, let's use that same query modified for the MIN function:

```
SELECT OrderID, MIN(Quantity)
FROM [Order Details]
WHERE OrderID BETWEEN 11000 AND 11002
GROUP BY OrderID
```

Which gives us the following results:

OrderID	
11000	25
11001	6
11002	15

(3 row(s) affected)

Modify it one more time for the MAX function:

```
SELECT OrderID, MAX(Quantity)
FROM [Order Details]
WHERE OrderID BETWEEN 11000 AND 11002
GROUP BY OrderID
```

And you come up with this:

OrderID	
11000	30
11001	60
11002	56

(3 row(s) affected)

What if, however, we wanted both the MIN and the MAX? Simple! Just use both in your query:

```
SELECT OrderID, MIN(Quantity),MAX(Quantity)
FROM [Order Details]
WHERE OrderID BETWEEN 11000 AND 11002
GROUP BY OrderID
```

Now, this will yield you an additional column and a bit of a problem:

OrderID		
11000	25	30
11001	6	60
11002	15	56

(3 row(s) affected)

Can you spot the issue here? We've gotten back everything that we've asked for, but now that we have more than one aggregate column, we have a problem identifying which column is which. Sure, in this particular example, we can be sure that the columns with the largest numbers are the columns generated by the MAX and the smallest by the MIN, but the answer to which column is which is not always so apparent. So let's make use of an *alias*. An alias allows you to change the name of a column in the result set, and you can create it by using the AS keyword:

```
SELECT OrderID, MIN(Quantity) AS Minimum, MAX(Quantity) AS Maximum
FROM [Order Details]
WHERE OrderID BETWEEN 11000 AND 11002
GROUP BY OrderID
```

Now our results are somewhat easier to make sense of:

OrderID	Minimum	Maximum
11000	25	30
11001	6	60
11002	15	56

(3 row(s) affected)

It's worth noting that the AS keyword is actually optional. Indeed, there was a day (prior to version 6.5 of SQL Server) when it wasn't even a valid keyword. If you like, you can execute the same query as before, except remove the two AS keywords from the query — you'll see that you wind up with exactly the same results. It's also worth noting that you can alias any column (and even, as we'll see in the next chapter, table names) — not just aggregates.

Let's re-run this last query, but this time we'll not use the AS keyword in some places, and we'll alias every column:

```
SELECT OrderID AS "Order Number", MIN(Quantity) Minimum, MAX(Quantity) Maximum
FROM [Order Details]
WHERE OrderID BETWEEN 11000 AND 11002
GROUP BY OrderID
```

Despite the AS keyword being missing in some places, we've still changed the name output for every column:

Order Number	Minimum	Maximum
11000	25	30
11001	6	60
11002	15	56

(3 row(s) affected)

I must admit that I usually don't include the AS keyword in my aliasing, but I would also admit that it's a bad habit on my part. I've been working with SQL Server since before the AS keyword was available and have unfortunately got set in my ways about it (I simply forget to use it). I would, however, strongly encourage you to go ahead and make use of this "extra" word. Why? Well, first, because it reads somewhat more clearly, and second because it's the ANSI standard way of doing things.

Chapter 3

So then, why did I even tell you about it? Well, I got you started doing it the right way — with the `AS` keyword — but I want you to be aware of alternate ways of doing things so that you aren't confused when you see something that looks a little different.

`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 Employees
WHERE EmployeeID = 5
```

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 Employees
```

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

```
-----
9

(1 row(s) affected)
```

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

Now, we're just getting started! If you look back at the header for this section (the `COUNT` section), you'll see that there are two different ways of using `COUNT`. We've already discussed using `COUNT` with the `*` option. 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 Customers
```

This is a slightly larger table, so you get a higher count:

```
-----  
91  
  
(1 row(s) affected)
```

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

```
SELECT COUNT(Fax)  
FROM Customers
```

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

```
-----  
69  
  
(1 row(s) affected)
```

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

This new result brings with it a question: why, since the Fax column exists for every row, is there a different count for Fax than there is for the row count in general? The answer is fairly obvious when you stop to think about it—there isn't a value, as such, for the Fax column in every row. In short, the `COUNT`, when used in any form other than `COUNT(*)`, ignores `NULL` values. Let's verify that `NULL` values are the cause of the discrepancy:

```
SELECT COUNT(*)  
FROM Customers  
WHERE Fax IS NULL
```

This should yield you the following recordset:

```
-----  
22  
  
(1 row(s) affected)
```

Now, let's do the math:

$69 + 22 = 91$

Chapter 3

That's 69 records with a defined value in the Fax field and 22 rows where the value in the Fax field is NULL, making a total of 91 rows.

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 7, 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 by thinking about the way that queries are likely to be run against the database and how you want your aggregates to work.

Before we leave the `COUNT` function, we had better see it in action with the `GROUP BY` clause.

Let's say our boss has asked us to find out the number of employees that report to each manager. The statements that we've done this far would either count up all the rows in the table (`COUNT(*)`) or all the rows in the table that didn't have null values (`COUNT(ColumnName)`). When we add a `GROUP BY` clause, these aggregators perform exactly as they did before, except that they return a count for each grouping rather than the full table—we can use this to get our number of reports:

```
SELECT ReportsTo, COUNT(*)
FROM Employees
GROUP BY ReportsTo
```

Notice that we are grouping only by the `ReportsTo`—the `COUNT()` function is an aggregator, and therefore does not have to be included in the `GROUP BY` clause.

```
ReportsTo
-----  --
NULL      1
2         5
5         3

(3 row(s) affected)
```

Our results tell us that the manager with 2 as his/her `ManagerID` has five people reporting to him or her, and that three people report to the manager with `ManagerID` 5. We are also able to tell that one `Employees` record had a NULL value in the `ReportsTo` field—this employee apparently doesn't report to anyone (hmmm, president of the company I suspect?).

It's probably worth noting that we, technically speaking, could use a `GROUP BY` clause without any kind of aggregator, but this wouldn't make sense. Why not? Well, SQL Server is going to wind up doing work on all the rows in order to group them up, but, functionally speaking, you would get the same result with a `DISTINCT` option (which we'll look at shortly), and it would operate much faster.

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

Up to now, all of our conditions have been against specific rows. If a given column in a row doesn't have a specific value or isn't within a range of values, then the entire row is left out. All of this happens before the groupings are really even thought about.

What if we want to place conditions on what the groups themselves look like? In other words, what if we want every row to be added to a group, but then we want to say that only after the groups are fully accumulated are we ready to apply the condition. Well, that's where the **HAVING** clause comes in.

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 they even have a chance to become part of a group, the **HAVING** clause is applied to the aggregated value for that group.

Let's start off with a slight modification to the **GROUP BY** query we used at the end of the last section — the one that tells us the number of employees assigned to each manager's **EmployeeID**:

```
SELECT ReportsTo AS Manager, COUNT(*) AS Reports
FROM Employees
GROUP BY ReportsTo
```

Now let's look at the results again:

Manager	Reports
-----	-----
NULL	1
2	5
5	3

(3 row(s) affected)

In the next chapter, we'll learn how to put names on the **EmployeeIDs** that are in the **Manager** column. For now though, we'll just note that there appear to be two different managers in the company. Apparently, everyone reports to these two people except for one person who doesn't have a manager assigned — that is probably our company's president (we could write a query to verify that, but we'll just trust in our assumptions for the time being).

We didn't put a **WHERE** clause in this query, so the **GROUP BY** was operating on every row in the table and every row is included in a grouping. To test out what would happen to our counts, let's add a **WHERE** clause:

```
SELECT ReportsTo AS Manager, COUNT(*) AS Reports
FROM Employees
WHERE EmployeeID != 5
GROUP BY ReportsTo
```

This yields us one slight change that we probably expected:

Manager	Reports
-----	-----
NULL	1
2	4
5	3

(3 row(s) affected)

The groupings were relatively untouched, but one row was eliminated before the `GROUP BY` clause was even considered. You see, the `WHERE` clause filtered out the one row where the `EmployeeID` was 5. As it happens, `EmployeeID` 5 reports to `ManagerID` 2. When `EmployeeID` 5 was no longer part of the query, the number of rows that were eligible to be in `ManagerID` 2's group was reduced by one.

I want to look at things a bit differently though. See if you can work out how to answer the following question. Which managers have more than four people reporting to them? You can look at the query without the `WHERE` clause and tell by the count, but how do you tell programmatically? That is, what if we need this query to return only the managers with more than four people reporting to them? If you try to work this out with a `WHERE` clause, you'll find that there isn't a way to return rows based on the aggregation — the `WHERE` clause is already completed by the system before the aggregation is executed. That's where our `HAVING` clause comes in:

```
SELECT ReportsTo AS Manager, COUNT(*) AS Reports
FROM Employees
GROUP BY ReportsTo
HAVING COUNT(*) > 4
```

Try it out and you'll come up with something a little bit more like what we were after:

Manager	Reports
-----	-----
2	5

(1 row(s) affected)

There is only one manager that has more than four employees reporting to him or her.

As I mentioned before — we could have gone and picked this out of the original listing fairly quickly, but the list is not always so short, and when dealing with things programmatically, you often need an exact answer that requires no further analysis.

Let's try a somewhat larger recordset, and then we'll leave this topic until we look at multi-table queries in the next chapter. If we want a query that will look at the total quantity of items ordered on each order in the system, it's a reasonably easy query:

```
SELECT OrderID, SUM(Quantity) AS Total
FROM [Order Details]
GROUP BY OrderID
```

OrderID	Total
-----	-----
10248	27
...	
...	
11075	42

```

11076          50
11077          72

(830 row(s) affected)

```

Unfortunately, it's somewhat difficult to do analysis on such a large list. So, let's have SQL Server do some paring down of this list to help us do our analysis. Assume that we're only interested in larger order quantities. Can you modify the query to return the same information, but limit it to orders where the total quantity of product ordered was over 300? It's as easy as adding the `HAVING` clause:

```

SELECT OrderID, SUM(Quantity) AS Total
FROM [Order Details]
GROUP BY OrderID
HAVING SUM(Quantity) > 300

```

Now we get a substantially shorter list:

```

OrderID      Total
-----
10895        346
11030        330

(2 row(s) affected)

```

As you can see, we can very quickly pare the list down to just the few in which we are most interested. We could perform additional queries now, specifically searching for OrderIDs 10895 and 11030, or as you'll learn in later chapters, we can `JOIN` the information from this query with additional information to yield information that is even more precise.

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 they deliver 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 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.

The *DISTINCT* and *ALL* Predicates

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.

Let's say, for example, that we wanted a list of the IDs of the suppliers of all of the products that we have in stock currently. We can easily get that information from the `Products` table with the following query:

```
SELECT SupplierID
FROM Products
WHERE UnitsInStock > 0
```

What we get back is one row matching the `SupplierID` for every row in the `Products` table:

```
SupplierID
-----
1
1
1
2
3
3
3
...
...
12
23
12

(72 row(s) affected)
```

While this meets our needs from a technical standpoint, it doesn't really meet our needs from a reality standpoint. Look at all those duplicate rows! As we've seen in other queries in this chapter, this particular table is small, but the number of rows returned and the number of duplicates can quickly become overwhelming. Like the problems we've discussed before — we have an answer. It comes in the form of the `DISTINCT` predicate on your `SELECT` statement.

Try re-running the query with a slight change:

```
SELECT DISTINCT SupplierID
FROM Products
WHERE UnitsInStock > 0
```

Now you come up with a true list of the `SupplierIDs` from which we currently have stock:

```
SupplierID
-----
1
2
3
...
...
27
```

28

29

```
(29 row(s) affected)
```

As you can see, this cut down the size of our list substantially and made the contents of the list more relevant. Another side benefit of this query is that it will actually perform better than the first one. Why? Well, we'll go into that later in the book when we discuss performance issues further, but for now, suffice it to say that not having to return every single row means that SQL Server doesn't have to do quite as much work in order to meet the needs of this query.

As the old commercials on television go, "But wait! There's more!" We're not done with `DISTINCT` yet. Indeed, the next example is one that you might be able to use as a party trick to impress your programmer friends. You see, this is one that an amazing number of SQL programmers don't even realize you can do — `DISTINCT` can be used as more than just a predicate for a `SELECT` statement. It can also be used in the expression for an aggregate. What do I mean? Let's compare three queries.

First, grab a row count for the Order Details table in Northwind:

```
SELECT COUNT(*)  
FROM [Order Details]
```

If you haven't modified the Order Details table, this should yield you 2,155 rows.

Now run the same query using a specific column to count:

```
SELECT COUNT(OrderID)  
FROM [Order Details]
```

Since the `OrderID` column is part of the key for this table, it can't contain any `NULLs` (more on this in the chapter in indexing). Therefore, the net count for this query is always going to be the same as the `COUNT(*)` — in this case, it's 2,155.

Key is a term used to describe a column or combination of columns that can be used to identify a row within a table. There are actually several different kinds of keys (we'll see much more on these in Chapters 7 through 9), but when the word "key" is used by itself, it is usually referring to the table's primary key. A primary key is a column or group of columns that is effectively the unique name for that row — when you refer to a row using its primary key, you can be certain that you will get back only one row because no two rows are allowed to have the same primary key within the same table.

Now for the fun part. Modify the query again:

```
SELECT COUNT(DISTINCT OrderID)  
FROM [Order Details]
```

Now we get a substantially different result:

```
-----  
830  
  
(1 row(s) affected)
```

All duplicate rows were eliminated before the aggregation occurred, so you have substantially fewer rows.

Note that you can use `DISTINCT` with any aggregate function, although 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.

That takes us to the `ALL` predicate. With one exception, it is a very rare thing indeed to see someone actually including an `ALL` in a statement. `ALL` is perhaps best understood as being the opposite of `DISTINCT`. Where `DISTINCT` is used to filter out duplicate rows, `ALL` says to include every row. `ALL` is the default for any `SELECT` statement except for situations where there is a `UNION`. We will discuss the impact of `ALL` in a `UNION` situation in the next chapter, but for now, realize that `ALL` is happening any time you don't ask for a `DISTINCT`.

Adding Data with the `INSERT` Statement

By now, you should have pretty much got the hang of basic `SELECT` statements. We would be doing well to stop here save for a pretty major problem — we wouldn't have very much data to look at if we didn't have some way of getting it into the database in the first place. That's where the `INSERT` statement comes in.

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. As we go through this section, try a few of the statements both with and without the `INTO` keyword. It's a little less typing if you leave it out, but it's also quite a bit stranger to read — it's up to you.

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 (1st value to 1st column, 2nd value to 2nd 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, but for now, we'll focus on single line inserts that use data that you explicitly provide. 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 must exactly match the number of columns in the column list. The datatype 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.

Whew! That's confusing, so let's practice with this some. Let's make use of the pubs database. This is a yet another database we'll be using throughout the book, and it should have been added to your system when you installed the Microsoft samples that included Northwind. Don't forget to make the change in your Query window or perform the USE Pubs command.

OK, most of the inserts we're going to do in this chapter will be to the stores table (it's a very nice, simple table to get started with), so let's take a look at the properties for that table, shown in Figure 3-3. To do this, expand the Tables node of the pubs database in the Object Explorer within the Management Studio. Then, also expand the columns node as showed in Figure 3-3.

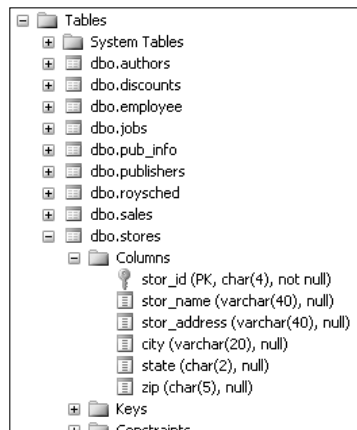


Figure 3-3

Chapter 3

In this table, every column happens to be a char or varchar.

For our first insert, we'll eliminate the optional column list and allow SQL Server to assume we're providing something for every column:

```
INSERT INTO stores
VALUES
('TEST', 'Test Store', '1234 Anywhere Street', 'Here', 'NY', '00319')
```

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. After executing this query, you should see a statement that tells you that one row was affected by your query. Now, just for fun, try running the exact same query a second time. You'll get the following error:

```
Msg 2627, Level 14, State 1, Line 1
Violation of PRIMARY KEY constraint 'UPK_storeid'. Cannot insert duplicate key in
object 'dbo.stores'.
The statement has been terminated.
```

Why did it work the first time and not the second? Because this table has a primary key that does not allow duplicate values for the stor_id field. As long as we changed that one field, we could have left the rest of the columns alone and it would have taken the new row. We'll see more of primary keys in the chapters on design and constraints.

So let's see what we inserted:

```
SELECT *
FROM stores
WHERE stor_id = 'TEST'
```

This query yields us exactly what we inserted:

stor_id	stor_name	stor_address	city	state	zip
-----	-----	-----	----	-----	-----
TEST	Test Store	1234 Anywhere Street	Here	NY	00319

(1 row(s) affected)

Note that I've trimmed a few spaces off the end of each column to help it fit on a page neatly, but the true data is just as we expected it to be.

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

```
INSERT INTO stores
(stor_id, stor_name, city, state, zip)
VALUES
('TST2', 'Test Store', 'Here', 'NY', '00319')
```

Note that, on the line with the data values, we've changed just two things. First, we've changed the value we are inserting into the primary key column so it won't generate an error. Second, we eliminated the value that was associated with the stor_address column since we have omitted that column in our

column list. There are a few different instances where we can skip a column in a column list and not provide any data for it in the `INSERT` statement. For now, we're just taking advantage of the fact that the `stor_address` column is not a required column—that is, it accepts `NULL`s. Since we're not providing a value for this column and since it has no default (we'll see more on defaults later on), this column will be set to `NULL` when we perform our insert. Let's verify that by re-running our test `SELECT` statement with one slight modification:

```
SELECT *
FROM stores
WHERE stor_id = 'TST2'
```

Now we see something a little different:

stor_id	stor_name	stor_address	city	state	zip
TST2	Test Store	NULL	Here	NY	00319

(1 row(s) affected)

Notice that a `NULL` was inserted for the column that we skipped.

Note that the columns have to be *nullable* in order to do this. What does that mean? Pretty much what it sounds like—it means that you are allowed to have `NULL` values for that column. Believe me, we will be discussing the nullability of columns at great length in this book, but for now, just realize that some columns allow `NULL`s and some don't. We can always skip providing information for columns that allow `NULL`s.

If, however, the column is not nullable, then one of three conditions must exist, or we will receive an error and the `INSERT` will be rejected:

- ❑ The column has been defined with a *default value*. A default is a constant value that is inserted if no other value is provided. We will learn how to define defaults in Chapter 7.
- ❑ The column is defined to receive some form of system-generated value. The most common of these is an `IDENTITY` value (covered more in the design chapter)—where the system typically starts counting at one first row, increments to 2 for the second, and so on. These aren't really "row numbers", as rows may be deleted later on and numbers can, under some conditions, get skipped, but they serve to make sure each row has its own identifier.
- ❑ We supply a value for the column.

Just for completeness, let's perform one more `INSERT` statement. This time, we'll insert a new sale into the sales table. To view the properties of the sales table, we can either open its Properties dialog as we did with the stores table, or we can run a system stored procedure called `sp_help`. `sp_help` will report information about any database object, user-defined datatype, or SQL Server datatype. The syntax for using `sp_help` is as follows:

```
EXEC sp_help <name>
```

To view the properties of the sales table, we just have to type the following into the Query Analyzer:


```
EXEC sp_help sales
```

Which returns (among other things):

Column_name	Type	Length	Nullable
-----	-----	-----	-----
stor_id	char	4	no
ord_num	varchar	20	no
ord_date	datetime	8	no
qty	smallint	2	no
payterms	varchar	12	no
title_id	tid	6	no

The sales table has six columns in it, but pay particular attention to the qty and ord_date columns — they are of types that we haven’t done inserts with up to this point (the title_id column is of type tid, but that is actually just a user-defined type that is still a character type with a length of 6).

What you need to pay attention to in this query is how to format the types as you’re inserting them. We do *not* use quotes for numeric values as we have with our character data. However, the datetime datatype does require quotes (essentially, it goes in as a string and it then gets converted to a datetime).

```
INSERT INTO sales
(stor_id, ord_num, ord_date, qty, payterms, title_id)
VALUES
('TEST', 'TESTORDER', '01/01/1999', 10, 'NET 30', 'BU1032')
```

This gets us back the now familiar (1 row(s) affected) message.

Note that, while I’ve used the MM/DD/YYYY format that is popular in the US, you can use a wide variety of other formats (such as the internationally more popular YYYY-MM-DD) with equal success. The default for your server will vary depending on if you purchase a localized copy of SQL Server or if the setting has been changed on the server.

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
- ☐ From 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 11.

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

USE Northwind

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

DECLARE @MyTable Table
(
    OrderID      int,
    CustomerID   char(5)
)

/* 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 OrderID, CustomerID
    FROM Northwind.dbo.Orders
    WHERE OrderID BETWEEN 10240 AND 10250

-- 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:

```
(3 row(s) affected)

OrderID      CustomerID
-----      -
10248        VINET
10249        TOMSP
10250        HANAR

(3 row(s) affected)
```

The first (3 row(s) affected) we see is the effect of the `INSERT...SELECT` statement at work — our `SELECT` statement returns three 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 11 through 13.

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`, although 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, but only one table at a time can be the subject of the update action. Don’t sweat this one too much — we haven’t looked at joining multiple tables yet (next chapter folks!), so we won’t get into complex `UPDATE` statements here. For now, we’ll look at simple updates.

Let's start off by doing some updates to the data that we inserted in the `INSERT` statement section. Let's re-run that query to look at one row of inserted data (don't forget to switch back to the pubs database):

```
SELECT *
FROM stores
WHERE stor_id = 'TEST'
```

Which returns the following to us:

stor_id	stor_name	stor_address	city	state	zip
TEST	Test Store	1234 Anywhere Street	Here	NY	00319

Let's update the value in the city column:

```
UPDATE stores
SET city = 'There'
WHERE stor_id = 'TEST'
```

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:

stor_id	stor_name	stor_address	city	state	zip
TEST	Test Store	1234 Anywhere Street	There	NY	00319

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 updated both columns:

```
UPDATE stores
SET city = 'There', state = 'CA'
WHERE stor_id = 'TEST'
```

If we choose, we can use an expression for the `SET` clause instead of the explicit values we've used thus far. For example, take a look at a few records from the titles table of the pubs database:

```
SELECT title_id, price
FROM titles
WHERE title_id LIKE 'BU%'
```

Our `LIKE` operator used here is going to provide us with the rows that start with BU, but which have any value after that (since we've used a `%`). Assuming you haven't been playing around with the data in the pubs database, you should end up with results similar to these:

title_id	price
BU1032	19.9900
BU1111	11.9500
BU2075	2.9900

Chapter 3

```
BU7832          19.9900
```

```
(4 row(s) affected)
```

Now that we've seen what the data looks like, let's try a somewhat different update by using an expression in our UPDATE statement:

```
UPDATE titles
SET price = price * 1.1
WHERE title_id LIKE 'BU%'
```

After executing that update, run the SELECT statement again:

```
SELECT title_id, price
FROM titles
WHERE title_id LIKE 'BU%'
```

You should see the price increased by 10 percent for every title ID that starts with BU:

```
title_id      price
-----
BU1032        21.9890
BU1111        13.1450
BU2075         3.2890
BU7832        21.9890
```

```
(4 row(s) affected)
```

Let's take this a little further to show you how much we can manipulate our results. For example, let's say that we have a business rule that says our prices need to be evenly payable with US currency. The prices we came up with don't meet our criteria, so we need to do something to get our prices rounded to the nearest whole cent (0.01 dollars). From the point that we're at, we could round to the nearest cent by running another update that does the rounding, but let's go back to the beginning. First, let's undo our last update:

```
UPDATE titles
SET price = price / 1.1
WHERE title_id LIKE 'BU%'
```

Notice that we only had to change just the one line of code. After you execute this, the SELECT statement should yield you the results with which we started:

```
title_id      price
-----
BU1032        19.9900
BU1111        11.9500
BU2075         2.9900
BU7832        19.9900
```

```
(4 row(s) affected)
```

Now we're ready to start from the beginning with a more advanced query. This time, we're going to perform pretty much the same update, but we'll round the updated data:

```
UPDATE titles
SET price = ROUND(price * 1.1, 2)
WHERE title_id LIKE 'BU%'
```

We've actually performed two mathematical operations before the `UPDATE` is actually written to each record. First, we perform the equivalent of our first query (increasing the price by 10 percent). Then we round it to match our business rule (must be to the cent) by indicating that our `ROUND()` function should round data off to two decimal places (hence the number 2 right after our 1.1,). The great thing is that we've been able to do this in just one operation rather than two.

Let's verify that result:

title_id	price
BU1032	21.9900
BU1111	13.1500
BU2075	3.2900
BU7832	21.9900

(4 row(s) affected)

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).

For example, the `stor_id` field in the `stores` table of the `pubs` database is a primary key. If we decide to change `stor_id` 10 to 35 in `stores`, then any data in the `sales` table that relates to that store may be orphaned and lost to us if the `stor_id` value in all of the records relating to `stor_id` 10 is not also updated to 35. As it happens, there is a constraint that references the `stores` table, so SQL Server would prevent such an orphaning situation in this case (we'll investigate constraints in Chapter 7), but updating primary keys is risky at best.

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 <table_name>
[WHERE <search condition>]
```

Chapter 3

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).

Because this is so easy, we'll perform only a couple of quick deletes that are focused on cleaning up the inserts that we performed earlier in the chapter. First, let's run a `SELECT` to make sure the first of those rows is still there:

```
SELECT *
FROM stores
WHERE stor_id = 'TEST'
```

If you haven't already deleted it, you should come up with a single row that matches what we added with our original `INSERT` statement. Now let's get rid of it:

```
DELETE stores
WHERE stor_id = 'TEST'
```

Note that we've run into a situation where SQL Server is refusing to delete this row because of referential integrity violations:

```
Msg 547, Level 16, State 1, Line 1
DELETE statement conflicted with COLUMN REFERENCE constraint
'FK_sales_stor_id_1BFD2C07'. The conflict occurred in database 'pubs', table
'sales', column 'stor_id'.
The statement has been terminated.
```

SQL Server won't let us delete a row if it is referenced as part of a foreign key constraint. We'll see much more on foreign keys in Chapter 7, but for now, just keep in mind that, if one row references another row (either in the same or a different table—it doesn't matter) using a foreign key, then the referencing row must be deleted before the referenced row can be deleted. Our last `INSERT` statement inserted a record into the sales table that had a `stor_id` of `TEST`—this record is referencing the record we have just attempted to delete.

Before we can delete the record from our stores table, we must delete the record it is referencing in the sales table:

```
DELETE sales
WHERE stor_id = 'TEST'
```

Now we can successfully re-run the first `DELETE` statement:

```
DELETE stores
WHERE stor_id = 'TEST'
```

You can do two quick checks to verify that the data was indeed deleted. The first happens automatically when the `DELETE` statement is executed—you should get a message telling you that one row was affected. The other quick check is to re-run the `SELECT` statement—you should get zero rows back.

For one more easy practice `DELETE`, we'll also kill off that second row by making just a slight change:

```
DELETE stores  
WHERE stor_id = 'TST2'
```

That's it for simple deletes! Like the other statements in this chapter, we'll come back to the `DELETE` statement when we're ready for more complex search conditions.

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.

In this chapter, you have gained a solid taste of making use of single table statements in T-SQL, but the reality is that you often need information from more than one table. In the next chapter, we will learn how to make use of JOINS to allow us to use multiple tables.

Exercises

1. Write a query that outputs all of the columns and all of the rows from the authors table of the pubs database.
2. Modify the query in Exercise 1 so it filters down the result to just the authors from the state of Utah. (HINT: There are 2.)
3. Add a new row into the authors table in the pubs database.
4. Remove the row you just added.

4

JOINS

Feel like a seasoned professional yet? Let me dash that feeling right away (just kidding)! While we've now got the basic statements under our belt, they are only a small part of the bigger picture of the statements we will run. To put it simply, there is often not that much you can do with just one table — especially in a highly normalized database.

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.

We will be looking into the concepts of normalization extensively in Chapter 8. For now, though, just keep in mind that the more normalized your database is, the more likely that you're going to have to join multiple tables together in order to get all the data you want.

In this chapter, we're going to introduce you to the process of combining tables into one result set by using the various forms of the `JOIN` clause. These will include:

- ☐ `INNER JOIN`
- ☐ `OUTER JOIN` (both `LEFT` and `RIGHT`)
- ☐ `FULL JOIN`
- ☐ `CROSS JOIN`

We'll also learn that there is more than one syntax available to use for joins, and that one particular syntax is the right choice. In addition, we'll take a look at the `UNION` operator, which allows us to combine the results of two queries into one.

JOINS

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

Chapter 4

returned is in one table, but the information we want to place conditions on is in another table. In these situations, this is where the `JOIN` clause comes in.

A `JOIN` does just what it sounds like—it puts the information from two tables together into one result set. We can think of a result set as being a “virtual” table. It has both columns and rows, and the columns have datatypes. Indeed, in Chapter 7, we’ll see how to treat a result set as if it were a table and use it for other queries.

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

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

This `JOIN` (at least apparently) joins records in a one-to-one relationship. We have one `Films` record joining to one `Actors` record.

Let’s expand things just a bit and see if you can see what’s happening. I’ve added another record to the `Actors` table:

FilmID	FirstName	LastName
1	Rex	Harrison
1	Audrey	Hepburn

Now let’s see what happens when we join that to the very same (only one record) `Films` table:

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

As you can see, the result has changed a bit — we are no longer seeing things as being one-to-one, but rather one-to-two, or more appropriately, what we would call one-to-many. We can use that single record in the `Films` table as many times as necessary in order to have complete (joined) information about the matching records in the `Actors` table.

Have you noticed how they are matching up? It is, of course, by matching up the `FilmID` field from the two tables to create one record out of two.

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 JOIN`s are far and away the most common kind of `JOIN`. They match records together based on one or more common fields, as do most `JOIN`s, 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 4

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

The preferred code for an `INNER JOIN` looks something like this:

```
SELECT <select list>
FROM <first_table>
<join_type> <second_table>
    [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 proprietary syntax required up to and prior to version 6.0 (and still used by many developers today). We'll take a look at the other syntax later in the chapter.

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

```
SELECT *
FROM Products
INNER JOIN Suppliers
    ON Products.SupplierID = Suppliers.SupplierID
```

The results of this query are too wide to print in this book, but if you run this, you should get something in the order of 77 rows back. There are several things worth noting about the results:

- ❑ The `SupplierID` column appears twice, but there's nothing to say which one is from which table.
- ❑ All columns were returned from both tables.
- ❑ The first columns listed were from the first table listed.

We can figure out which `SupplierID` is which just by looking at what table we selected first and matching it with the first `SupplierID` column that shows up, but this is tedious at best, and at worst, prone to errors. That's one of many reasons why using the plain `*` operator in `JOINS` is ill advised. In the case of an `INNER JOIN`, however, it's not really that much of a problem because we know that both `SupplierID` columns, even though they came from different tables, will be exact duplicates of each other. How do we know that? Think about it — since we're doing an `INNER JOIN` on those two columns, they have to match or the record wouldn't have been returned! Don't get in the habit of counting on this, however. When we look at other `JOIN` types, we'll find that we can't depend on the `JOIN` values being equal.

As for all columns being returned from both tables, that is as expected. We used the `*` operator, which as we've learned before is going to return all columns to us. As I mentioned earlier, the use of the `*` operator in joins is a bad habit. It's quick and easy, but it's also dirty—it is error-prone and can result in poor performance.

One good principle to adopt early on is this to select what you need, and need what you select. What I'm getting at here is that every additional record or column that you return takes up additional network bandwidth, and often, additional query processing on your SQL Server. The upshot is that selecting unnecessary information hurts performance not only for the current user, but also for every other user of the system and for users of the network on which the SQL Server resides.

Select only the columns that you are going to be using and make your WHERE clause as restrictive as possible.

If you must insist on using the `*` operator, you should use it only for the tables from which you need all the columns. That's right—the `*` operator can be used on a per-table basis. For example, if we want all of our product information, but only the name of the supplier, we could have changed our query to read:

```
SELECT Products.*, CompanyName
FROM Products
INNER JOIN Suppliers
    ON Products.SupplierID = Suppliers.SupplierID
```

If you scroll over to the right in the results of this query, you'll see that most of the supplier information is now gone. Indeed, we also only have one instance of the `SupplierID` column. What we got in our result set was all the columns from the `Products` table (since we used the `*` qualified for just that table—our one instance of `SupplierID` came from this part of the `SELECT` list) and the only column that had the name `CompanyName` (which happened to be from the `Suppliers` table). Now let's try it again, with only one slight change:

```
SELECT Products.*, SupplierID
FROM Products
INNER JOIN Suppliers
    ON Products.SupplierID = Suppliers.SupplierID
```

Uh, oh—this is a problem. We get an error back:

```
Msg 209, Level 16, State 1, Line 1
Ambiguous column name 'SupplierID'.
```

Why did `CompanyName` work and `SupplierID` not work? For just the reason SQL Server has indicated—our column name is ambiguous. While `CompanyName` exists only in the `Suppliers` table, `SupplierID` appears in both tables. SQL Server has no way of knowing which one we want. All the instances where we have returned `SupplierID` up to this point have been resolvable: that is, SQL Server could figure out which table was which. In the first query (where we used a plain `*` operator), we asked SQL Server to return everything—that would include *both* `SupplierID` columns, so no name resolution was necessary. In our second example (where we qualified the `*` to be only for `Products`), we again said nothing specifically about which `SupplierID` column to use—instead, we said pull everything from the `Products` table and `SupplierID` just happened to be in that list. `CompanyName` was resolvable because there was only one `CompanyName` column, so that was the one we wanted.

Chapter 4

When we want to refer to a column where the column name exists more than once in our `JOIN` result, we must *fully qualify* the column name. We can do this in one of two ways:

- ❑ Provide the name of the table that the desired column is from, followed by a period and the column name (`Table.ColumnName`)
- ❑ Alias the tables, and provide that alias, followed by a period and the column name (`Alias.ColumnName`)

The task of providing the names is straightforward enough — we’ve already seen how that works with the qualified `*` operator, but let’s try our `SupplierID` query again with a qualified column name:

```
SELECT Products.*, Suppliers.SupplierID
FROM Products
INNER JOIN Suppliers
    ON Products.SupplierID = Suppliers.SupplierID
```

Now things are working again and we have the `SupplierID` from the `Suppliers` table added back to the far right-hand side of the result set.

Aliasing the table is only slightly trickier but can cut down on the wordiness and help the readability of your query. It works almost exactly the same as aliasing a column in the simple `SELECT`s that we did in the last chapter — right after the name of the table, we simply state the alias we want to use to refer to that table. Note that, just as with column aliasing, we can use the `AS` keyword (but for some strange reason, this hasn’t caught on as much in practice):

```
SELECT p.*, s.SupplierID
FROM Products p
INNER JOIN Suppliers s
    ON p.SupplierID = s.SupplierID
```

Run this code and you’ll see that we receive the exact same results as we did in the last query.

Be aware that using an alias is an all-or-nothing proposition. Once you decide to alias a table, you must use that alias in every part of the query. This is on a table-by-table basis, but try running some mixed code and you’ll see what I mean:

```
SELECT p.*, Suppliers.SupplierID
FROM Products p
INNER JOIN Suppliers s
    ON p.SupplierID = s.SupplierID
```

This seems like it should run fine, but it will give you an error:

```
Msg 4104, Level 16, State 1, Line 1
The multi-part identifier "Suppliers.SupplierID" could not be bound.
```

Again, you can mix and match which tables you choose to use aliasing on and which you don’t, but once you make a decision, you have to be consistent.

Think back to those bullet points we saw a few pages earlier; we noticed that the columns from the first table listed in the `JOIN` were the first columns returned. Take a break for a moment and think about why that is, and what you might be able to do to control it.

SQL Server always uses a column order that is the best guess it can make at how you want the columns returned. In our first query, we used one global `*` operator, so SQL Server didn't have much to go on. In that case, it goes on the small amount that it does have—the order of the columns as they exist physically in the table, and the order of tables that you specified in your query. The nice thing is that it is extremely easy to reorder the columns—we just have to be explicit about it. The simplest way to reorder the columns would be to change which table is mentioned first, but we can actually mix and match our column order by simply explicitly stating the columns that we want (even if it is every column), and the order in which we want them.

Try It Out A Simple JOIN

Let's try a small query to demonstrate the point:

```
SELECT p.ProductID, s.SupplierID, p.ProductName, s.CompanyName
FROM Products p
INNER JOIN Suppliers s
    ON p.SupplierID = s.SupplierID
WHERE p.ProductID < 4
```

This yields a pretty simple result set:

ProductID	SupplierID	ProductName	CompanyName
-----	-----	-----	-----
1	1	Chai	Exotic Liquids
2	1	Chang	Exotic Liquids
3	1	Aniseed Syrup	Exotic Liquids

(3 row(s) affected)

How It Works

Unlike when we were non-specific about what columns we wanted (when we just used the `*`), this time we were specific about what we wanted, and thus SQL Server knew exactly what to give us—the columns have come out in exactly the order that we've specified in our `SELECT` list.

How an INNER JOIN Is Like a WHERE Clause

In the `INNER JOIN`s that we've done so far, we've really been looking at the concepts that will work for any `JOIN` type—the column ordering and aliasing is exactly the same for any `JOIN`. The part that makes an `INNER JOIN` different from other `JOIN`s is that it is an *exclusive join*—that is, it excludes all records that don't have a value in both tables (the first named, or left table, and the second named, or right table).

Our first example of this was seen with our imaginary `Films` and `Actors` tables. Bogey was left out because he didn't have a matching movie in the `Films` table. Let's look at a real example or two to show how this works.

We have a `Customers` table, and it is full of customer names and addresses. This does not mean, however, that the customers have actually ordered anything. Indeed, I'll give you a hint up front and tell you that there are some customers that have *not* ordered anything, so let's take a question and turn it into a query—the question I've picked calls for an `INNER JOIN`, but we'll see how slight changes to the question will change our choice of `JOIN`s later on.

Chapter 4

Here’s a question you might get from a sales manager: “Can you show me all the customers who have placed orders with us?”

You can waste no time in saying, “Absolutely!” and starting to dissect the parts of the query. What are the things we need? Well, the sales manager asked about both customers and orders, so we can take a guess that we will need information from both of those tables. The sales manager asked only for a list of customers, so the `CompanyName` and perhaps the `CustomerID` are the only columns we need. Note that while we need to include the `Orders` table to figure out whether a customer has ordered anything or not, we do not need to return anything from it to make use of it (that’s why it’s not in the `SELECT` list that follows). The sales manager has asked for a list of customers where there has been an order, so the question calls for a solution where there is both a `Customers` record and an `Orders` record — that’s our `INNER JOIN` scenario, so we should now be ready to write the query:

```
SELECT DISTINCT c.CustomerID, c.CompanyName
FROM Customers c
INNER JOIN Orders o
    ON c.CustomerID = o.CustomerID
```

If you haven’t altered any data in the `Northwind` database, this should give you 89 rows. Note that we used the `DISTINCT` keyword because we only need to know that the customers have made orders (once was sufficient), not how many orders. Without the `DISTINCT`, a customer who ordered multiple times would have had a separate record returned for each `Orders` record to which it joined.

Now let’s see if we got all the customers back. Try running a simple `COUNT` query:

```
SELECT COUNT(*) AS "No. Of Records" FROM Customers
```

And you’ll get back a different count on the number of rows:

```
No. Of Records
-----
91

(1 row(s) affected)
```

Where did the other two rows go? As expected, they were excluded from the result set because there were no corresponding records in the `Orders` table. It is for this reason that an `INNER JOIN` is comparable to a `WHERE` clause. Just as an `INNER JOIN` will exclude rows because they had no corresponding match in the other table, the `WHERE` clause also limits the rows returned to those that match the criteria specified.

Just for a little more proof and practice, consider the following tables from the `pubs` database:

authors	Titles	title`author
au_id	title_id	au_id
au_lname	Title	title_id
au_fname	Type	au_ord
Phone	pub_id	royaltyper

authors	Titles	title`author
address	Price	
City	Advance	
State	Royalty	
Zip	ytd_sales	
contract	notes	
	pubdate	

What we’re looking for this time is a query that returns all the authors that have written books and the titles of the books that they have written. Try coming up with this query on your own for a few minutes; then we’ll dissect it a piece at a time.

The first thing to do is to figure out what data we need to return. Our question calls for two different pieces of information to be returned: the author’s name and the book’s title. The author’s name is available (in two parts) from the authors table. The book’s title is available in the titles table, so we can write the first part of our `SELECT` statement:

```
SELECT au_lname + ', ' + au_fname AS "Author", title
```

Like many languages, the “+” operator can be used for concatenation of strings as well as the addition of numbers. In this case, we are just connecting the last name to the first name with a comma separator in between.

What we need now is something to join the two tables on, and that’s where we run into our first problem—there doesn’t appear to be one. The tables don’t seem to have anything in common on which we can base our `JOIN`.

This brings us to the third table listed. Depending on which database architect you’re talking to, a table such as `titleauthor` will be called a number of different things. The most common names that I’ve come across for this type of table are *linking table* or *associate table*.

A linking table (also sometimes called an associate or merge table) is any table for which the primary purpose is not to store its own data, but rather to relate the data stored in other tables. You can consider it to be “linking” or “associating” the two or more tables. These tables are used to get around the common situation where you have what is called a “many-to-many” relationship between the tables. This is where two tables relate, but either table can have many records that potentially match many records in the other table. SQL Server can’t implement such relationships directly, so the linking table breaks down the many-to-many relationship into two “one-to-many” relationships—which SQL Server can handle. We will see much more on this subject in Chapter 8.

Chapter 4

This particular table doesn't meet the criteria for a linking table in the strictest sense of the term, but it still serves that general purpose, and I, not being a purist, consider it such a table. By using this third table, we are able to indirectly join the authors and titles tables by joining each to the linking table, titleauthor. authors can join to titleauthor based on au_id, and titles can join to titleauthor based on title_id.

Try It Out

More Complex JOINS

Adding this third table into our JOIN is no problem — we just keep on going with our FROM clause and JOIN keywords (don't forget to switch the database to pubs):

```
SELECT a.au_lname + ', ' + a.au_fname AS "Author", t.title
FROM authors a
JOIN titleauthor ta
    ON a.au_id = ta.au_id
JOIN titles t
    ON t.title_id = ta.title_id
```

Notice that, because we've used aliases on the tables, we had to go back and change our SELECT clause to use the aliases, but our SELECT statement with a three-table join is now complete! If we execute this (I'm using the grid mode here), we get:

Author	Title
Bennet, Abraham	The Busy Executive's Database Guide
Blotchet-Halls, Reginald	Fifty Years in Buckingham Palace Kitchens
Carson, Cheryl	But Is It User Friendly?
DeFrance, Michel	The Gourmet Microwave
del Castillo, Innes	Silicon Valley Gastronomic Treats
Dull, Ann	Secrets of Silicon Valley
Green, Marjorie	The Busy Executive's Database Guide
Green, Marjorie	You Can Combat Computer Stress!
Gringlesby, Burt	Sushi, Anyone?
Hunter, Sheryl	Secrets of Silicon Valley
Karsen, Livia	Computer Phobic AND Non-Phobic Individuals: Behavior Variations
Locksley, Charlene	Net Etiquette
Locksley, Charlene	Emotional Security: A New Algorithm
MacFeather, Stearns	Cooking with Computers: Surreptitious Balance Sheets
MacFeather, Stearns	Computer Phobic AND Non-Phobic Individuals: Behavior Variations

Author	Title
O'Leary, Michael	Cooking with Computers: Surreptitious Balance Sheets
O'Leary, Michael	Sushi, Anyone?
Panteley, Sylvia	Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean
Ringer, Albert	Is Anger the Enemy?
Ringer, Albert	Life Without Fear
Ringer, Anne	The Gourmet Microwave
Ringer, Anne	Is Anger the Enemy?
Straight, Dean	Straight Talk About Computers
White, Johnson	Prolonged Data Deprivation: Four Case Studies
Yokomoto, Akiko	Sushi, Anyone?

Note that your sort order may differ from what you see here — remember, SQL Server makes no promises about the order your results will arrive in unless you use an `ORDER BY` clause — since we didn't use `ORDER BY`, the old adage “Actual results may vary” comes into play.

How It Works

If we were to do a simple `SELECT *` against the authors table, we would find that several authors were left out because, although they have been entered into the authors table, they apparently haven't written any matching books (at least not that we have in our database). Indeed, we've even left out one title (*The Psychology of Computer Cooking*) because we can't match it up with an author! Once again, the key to `INNER JOIN`s is that they are exclusive.

Other than that, our third table worked with the first two just like the second did with the first. We could keep adding tables, and each new table would work exactly the same way.

Notice that we did not use the `INNER` keyword in this last query. That is because an `INNER JOIN` is the default `JOIN` type. Schools of thought vary on this, but I believe that because leaving the `INNER` keyword out has dominated the way code has been written for so long, that it is almost more confusing to put it in — that's why you won't see me use it again in this book.

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:

- ❑ We, 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 nested subqueries (which we will look into in Chapter 7).

Earlier in this chapter, we introduced the concept of a `JOIN` having 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.

To learn how to construct `OUTER JOINS` correctly, we're going to use two syntax illustrations. The first deals with the simple scenario of a two-table `OUTER JOIN`. The second will deal with the more complex scenario of mixing `OUTER JOINS` with any other `JOIN`.

The Simple `OUTER JOIN`

The first syntax situation is the easy part — most people get this part just fine.

```
SELECT <SELECT list>
FROM <the table you want to be the "LEFT" table>
<LEFT|RIGHT> [OUTER] JOIN <table you want to be the "RIGHT" table>
ON <join condition>
```

In the examples, you'll find that I tend to use the full syntax — that is, I include the `OUTER` keyword (for example `LEFT OUTER JOIN`). Note that the `OUTER` keyword is optional — you need only include the `LEFT` or `RIGHT` (for example `LEFT JOIN`).

What I'm trying to get across here is that the table that comes before the `JOIN` keyword is considered to be the `LEFT` table, and the table that comes after the `JOIN` keyword is considered to be the `RIGHT` table.

`OUTER JOINS` are, as we've said, inclusive in nature. What specifically gets included depends on which side of the join you have emphasized. A `LEFT OUTER JOIN` includes all the information from the table on the left, and a `RIGHT OUTER JOIN` includes all the information from the table on the right. Let's put this into practice with a small query so that you can see what I mean.

Let's say we want to know what all our discounts are, the amount of each discount, and which stores use them. Looking over our pubs database, we have tables called `discounts` and `stores` as follows:

discounts	stores
discounttype	stor_id
stor_id	stor_name
Lowqty	stor_address
Highqty	city
Discount	state
	zip

We can directly join these tables based on the `stor_id`. If we did this using a common `INNER JOIN`, it would look something like:

```
SELECT discounttype, discount, s.stor_name
FROM discounts d
JOIN stores s
  ON d.stor_id = s.stor_id
```

This yields us just one record:

```
discounttype      discount      stor_name
-----
Customer Discount    5.00      Bookbeat

(1 row(s) affected)
```

Think about this though. We wanted results based on the discounts we have — not which ones were actually in use. This query only gives us discounts that we have matching stores for — it doesn't answer the question!

What we need is something that's going to return every discount and the stores where applicable.

Try It Out Outer JOINS

In order to return every discount, and the stores where applicable, we need to change only the `JOIN` type in the query:

```
SELECT discounttype, discount, s.stor_name
FROM discounts d
LEFT OUTER JOIN stores s
  ON d.stor_id = s.stor_id
```

This yields us somewhat different results:

```
discounttype      discount      stor_name
-----
Initial Customer    10.50      NULL
```

```
Volume Discount      6.70      NULL
Customer Discount    5.00      Bookbeat

(3 row(s) affected)
```

If you were to perform a `SELECT *` against the discounts table, you’d quickly find that we have included every row from that table.

How It Works

We are doing a `LEFT JOIN`, and the discounts table is on the left side of the `JOIN`. But what about the stores table? If we are joining, and we don’t have a matching record for the stores table, then what happens? Since it is not on the inclusive side of the join (in this case, the `LEFT` side), SQL Server will fill in a `NULL` for any value that comes from the opposite side of the join if there is no match with the inclusive side of the join. In this case, all but one of our rows has a `stor_name` that is `NULL`. What we can discern from that is that two of our discounts records (the two with `NULLs` in the column from the stores table) do not have matching store records — that is, no stores are using that discount type.

We’ve answered the question then; of the three discount types, only one is being used (Customer Discount) and it is being used only by one store (Bookbeat).

Try It Out RIGHT OUTER JOINS

Now, let’s see what happens if we change the join to a `RIGHT OUTER JOIN`:

```
SELECT discounttype, discount, s.stor_name
FROM discounts d
RIGHT OUTER JOIN stores s
ON d.stor_id = s.stor_id
```

Even though this seems like a very small change, it actually changes our results rather dramatically:

```
discounttype      discount      stor_name
-----
NULL              NULL          Eric the Read Books
NULL              NULL          Barnum's
NULL              NULL          News & Brews
NULL              NULL          Doc-U-Mat: Quality Laundry and Books
NULL              NULL          Fricative Bookshop
Customer Discount  5.00          Bookbeat

(6 row(s) affected)
```

How It Works

If you were to perform a `SELECT *` on the stores table now, you would find that all of the records from stores have been included in the query. Where there is a matching record in discounts, the appropriate discount record is displayed. Everywhere else, the columns that are from the discounts table are filled in with `NULLs`. Assuming that we always name the discounts table first, and the stores table second, then we would use a `LEFT JOIN` if we want all the discounts, and a `RIGHT JOIN` if we want all the stores.

Finding Orphan or Non-Matching Records

We can actually use the inclusive nature of `OUTER JOINS` to find non-matching records in the exclusive table. What do I mean by that? Let's look at an example.

Let's change our discount question. We want to know the store name for all the stores that do not have any kind of discount record. Can you come up with a query to perform this based on what we know this far? Actually, the very last query we ran has us 90 percent of the way there. Think about it for a minute; an `OUTER JOIN` returns a `NULL` value in the discounts-based columns wherever there is no match. What we are looking for is pretty much the same result set as we received in the last query, except that we want to filter out any records that do have a discount, and we want only the store name. To do this, we simply change our `SELECT` list and add a `WHERE` clause. To make it a touch prettier to give to our manager, we also alias the `stor_name` field to be the more expected "Store Name":

```
SELECT s.stor_name AS "Store Name"
FROM discounts d
RIGHT OUTER JOIN stores s
      ON d.stor_id = s.stor_id
WHERE d.stor_id IS NULL
```

As expected, we have exactly the same stores that had `NULL` values before:

```
Store Name
-----
Eric the Read Books
Barnum's
News & Brews
Doc-U-Mat: Quality Laundry and Books
Fricative Bookshop

(5 row(s) affected)
```

There is one question you might be thinking at the moment that I want to answer in anticipation, so that you're sure you understand why this will always work. The question is: "What if the discount record really has a `NULL` value?" Well, that's why we built a `WHERE` clause on the same field that was part of our join. If we are joining based on the `stor_id` columns in both tables, then only three conditions can exist:

- ❑ If the `stores.stor_id` column has a non-`NULL` value, then, according to the `ON` operator of the `JOIN` clause, if a `discounts` record exists, then `discounts.stor_id` must also have the same value as `stores.stor_id` (look at the `ON d.stor_id = s.stor_id`).
- ❑ If the `stores.stor_id` column has a non-`NULL` value, then, according to the `ON` operator of the `JOIN` clause, if a `discounts` record does not exist, then `discounts.stor_id` will be returned as `NULL`.
- ❑ If the `stores.stor_id` happens to have a `NULL` value, and `discounts.stor_id` also has a `NULL` value, there will be no join, and `discounts.stor_id` will return `NULL` because there is no matching record.

A value of `NULL` does not join to a value of `NULL`. Why? Think about what we've already said about comparing `NULL`s — a `NULL` does not equal `NULL`. Be extra careful of this when coding. One of the more common questions I am asked is, "Why isn't this working?" in a situation where people are using an "equal to" operation on a `NULL` — it simply doesn't work because they are not equal. If you want to test this, try executing some simple code:

Chapter 4

```
IF (NULL=NULL)
    PRINT 'It Does'
ELSE
    PRINT 'It Doesn't'
```

If you execute this, you'll get the answer to whether your SQL Server thinks a NULL equals a NULL — that is "It Doesn't."

This was actually a change of behavior that began in SQL Server 7.0. Be aware that if you are running in SQL Server 6.5 compatibility mode (should be rare given it's vintage at this point, but you never know), or if you have ANSI_NULLS set to off (through server options or a SET statement), then you will get a different answer (your server will think that a NULL equals a NULL). This is considered non-standard at this point. It is a violation of the ANSI standard, and it is no longer compatible with the default configuration for SQL Server. (Even the Books Online will tell you a NULL is not equal to a NULL.)

Let's use this notion of being able to identify non-matching records to locate some of the missing records from one of our earlier INNER JOINS. Remember these two queries, which we ran against Northwind?

```
SELECT DISTINCT c.CustomerID, c.CompanyName
FROM Customers c
INNER JOIN Orders o
    ON c.CustomerID = o.CustomerID
```

And . . .

```
SELECT COUNT(*) AS "No. Of Records" FROM Customers
```

The first was one of our queries where we explored the INNER JOIN. We discovered by running the second query that the first had excluded (by design) some rows. Now let's identify the excluded rows by using an OUTER JOIN.

We know from our SELECT COUNT(*) query that our first query is missing some records from the Customers table. (It may also be missing records from the Orders table, but we're not interested in that at the moment.) The implication is that there are records in the Customers table that do not have corresponding Orders records. While our manager's first question was about all the customers that had placed orders, it would be very common to ask just the opposite: "What customers haven't placed an order?" That question is answered with the same result as asking, "What records exist in Customers that don't have corresponding records in the Orders table?" The solution has the same structure as our query to find stores without discounts:

```
USE Northwind

SELECT c.CustomerID, CompanyName
FROM Customers c
LEFT OUTER JOIN Orders o
    ON c.CustomerID = o.CustomerID
WHERE o.CustomerID IS NULL
```

Just that quick we are able to not only find out how many customers haven't placed orders, but now we know which customers they are (I suspect the sales department will contact them shortly . . .):

CustomerID	CompanyName
PARIS	Paris spécialités
FISSA	FISSA Fabrica Inter. Salchichas S.A.

(2 row(s) affected)

Note that whether you use a `LEFT` or a `RIGHT JOIN` doesn't matter as long as the correct table or group of tables is on the corresponding side of the `JOIN`. For example, we could have run the preceding query using a `RIGHT JOIN` as long as we also switched which sides of the `JOIN` the `Customers` and `Orders` tables were on. For example this would have yielded exactly the same results:

```
SELECT c.CustomerID, CompanyName
FROM Orders o
RIGHT OUTER JOIN Customers c
            ON c.CustomerID = o.CustomerID
WHERE o.CustomerID IS NULL
```

When we take a look at even more advanced queries, we'll run into a slightly more popular way of finding records that exist in one table without there being corresponding records in another table. Allow me to preface that by saying that using `JOINS` is usually our best bet in terms of performance. There are exceptions to the rule that we will cover as we come across them, but in general, the use of `JOINS` will be the best when faced with multiple options.

Dealing with More Complex OUTER JOINS

Now we're on to our second illustration and how to make use of it. This scenario is all about dealing with an `OUTER JOIN` mixed with some other `JOIN` (no matter what the variety).

It is when combining an `OUTER JOIN` with other `JOINS` that the concept of sides becomes even more critical. What's important to understand here is that everything to the "left" — or before — the `JOIN` in question will be treated just as if it was a single table for the purposes of inclusion or exclusion from the query. The same is true for everything to the "right" — or after — the `JOIN`. The frequent mistake here is to perform a `LEFT OUTER JOIN` early in the query and then use an `INNER JOIN` late in the query. The `OUTER JOIN` includes everything up to that point in the query, but the `INNER JOIN` may still create a situation where something is excluded! My guess is that you will, like most people (including me for a while), find this exceptionally confusing at first, so let's see what we mean with some examples. Because none of the databases that come along with SQL Server has any good scenarios for demonstrating this, we're going to have to create a database and sample data of our own.

If you want to follow along with the examples, the example database called `Chapter4DB` can be created by running `Chapter4DB.sql` from the downloaded source code.

What we are going to do is to build up a query step-by-step and watch what happens. The query we are looking for will return a vendor name and the address of that vendor. The example database only has a few records in it; so let's start out by selecting all the choices from the central item of the query — the vendor. We're going to go ahead and start aliasing from the beginning, since we will want to do this in the end:

Chapter 4

```
SELECT v.VendorName
FROM Vendors v
```

This yields us a scant three records:

```
VendorName
-----
Don's Database Design Shop
Dave's Data
The SQL Sequel

(3 row(s) affected)
```

These are the names of every vendor that we have at this time. Now let's add in the address information—there are two issues here. First, we want the query to return every vendor no matter what, so we'll make use of an `OUTER JOIN`. Next, a vendor can have more than one address and vice versa, so the database design has made use of a linking table. This means that we don't have anything to directly join the `Vendors` and `Address` table—we must instead join both of these tables to our linking table, which is called `VendorAddress`. Let's start out with the logical first piece of this join:

```
SELECT v.VendorName
FROM Vendors v
LEFT OUTER JOIN VendorAddress va
    ON v.VendorID = va.VendorID
```

Because `VendorAddress` doesn't itself have the address information, we're not including any columns from that table in our `SELECT` list. `VendorAddress`'s sole purpose in life is to be the connection point of a many-to-many relationship (one vendor can have many addresses, and, as we've set it up here, an address can be the home of more than one vendor). Running this, as we expect, gives us the same results as before:

```
VendorName
-----
Don's Database Design Shop
Dave's Data
The SQL Sequel

(3 row(s) affected)
```

Let's take a brief time-out from this particular query to check on the table against which we just joined. Try selecting out all the data from the `VendorAddress` table:

```
SELECT *
FROM VendorAddress
```

Just two records are returned:

```
VendorID  AddressID
-----
1         1
2         3

(2 row(s) affected)
```

We know, therefore, that our `OUTER JOIN` is working for us. Since there are only two records in the `VendorAddress` table, and three vendors are returned, we must be returning at least one row from the `Vendors` table that didn't have a matching record in the `VendorAddress` table. While we're here, we'll just verify that by briefly adding one more column back to our vendors query:

```
SELECT v.VendorName, va.VendorID
FROM Vendors v
LEFT OUTER JOIN VendorAddress va
    ON v.VendorID = va.VendorID
```

Sure enough, we wind up with a `NULL` in the `VendorID` column from the `VendorAddress` table:

VendorName	VendorID
-----	-----
Don's Database Design Shop	1
Dave's Data	2
The SQL Sequel	NULL

(3 row(s) affected)

The vendor named "The SQL Sequel" would not have been returned if we were using an `INNER` or `RIGHT JOIN`. Our use of a `LEFT JOIN` has ensured that we get all vendors in our query result.

Now that we've tested things out a bit, let's return to our original query and then add in the second `JOIN` to get the actual address information. Because we don't care if we get all addresses, no special `JOIN` is required — at least, it doesn't appear that way at first . . .

```
SELECT v.VendorName, a.Address
FROM Vendors v
LEFT OUTER JOIN VendorAddress va
    ON v.VendorID = va.VendorID
JOIN Address a
    ON va.AddressID = a.AddressID
```

We get back the address information as expected, but there's a problem:

VendorName	Address
-----	-----
Don's Database Design Shop	1234 Anywhere
Dave's Data	567 Main St.

(2 row(s) affected)

Somehow, we've lost one of our vendors. That's because SQL Server is applying the rules in the order that we've stated them. We have started with an `OUTER JOIN` between `Vendors` and `VendorAddress`. SQL Server does just what we want for that part of the query — it returns all vendors. The issue comes when it applies the next set of instructions. We have a result set that includes all the vendors, but we now apply that result set as part of an `INNER JOIN`. Because an `INNER JOIN` is exclusive to both sides of the `JOIN`, only records where the result of the first `JOIN` has a match with the second `JOIN` will be included. Because only two records match up with a record in the `Address` table, only two records are returned in the final result set. We have two ways of addressing this:

Chapter 4

- ❑ Add yet another `OUTER JOIN`
- ❑ Change the ordering of the `JOINS`

Let's try it both ways. We'll add another `OUTER JOIN` first:

```
SELECT v.VendorName, a.Address
FROM Vendors v
LEFT OUTER JOIN VendorAddress va
    ON v.VendorID = va.VendorID
LEFT OUTER JOIN Address a
    ON va.AddressID = a.AddressID
```

And now we get to our expected results:

VendorName	Address
Don's Database Design Shop	1234 Anywhere
Dave's Data	567 Main St.
The SQL Sequel	NULL

(3 row(s) affected)

Now let's do something slightly more dramatic and reorder our original query:

```
SELECT v.VendorName, a.Address
FROM VendorAddress va
JOIN Address a
    ON va.AddressID = a.AddressID
RIGHT OUTER JOIN Vendors v
    ON v.VendorID = va.VendorID
```

And we still get our desired result:

VendorName	Address
Don's Database Design Shop	1234 Anywhere
Dave's Data	567 Main St.
The SQL Sequel	NULL

(3 row(s) affected)

The question you should be asking now is, "Which way is best?" Quite often in SQL, there are several ways of executing the query without one having any significant advantage over the other — this is *not* one of those times.

I would most definitely steer you to the second of the two solutions.

The rule of thumb is to get all of the `INNER JOINS` you can out of the way first, you will then find yourself using the minimum number of `OUTER JOINS`, and decreasing the number of errors in your data.

The reason has to do with navigating as quickly as possible to your data. If you keep adding `OUTER JOIN`s not because of what's happening with the current table you're trying to add in, but because you're trying to carry through an earlier `JOIN` result, you are much more likely to include something you don't intend, or make some sort of mistake in your overall logic. The second solution addresses this by using only the `OUTER JOIN` where necessary — just once. You can't always create a situation where the `JOINS` can be moved around to this extent, but you often can.

I can't stress enough how often I see errors with `JOIN` order. It is one of those areas that just seem to give developers fits. Time after time I get called in to look over a query that someone has spent hours verifying each section of, and it seems that at least half the time I get asked whether I know about this SQL Server "bug." The bug isn't in SQL Server in this case — it's with the developer. If you take anything away from this section, I hope it is that `JOIN` order is one of the first places to look for errors when the results aren't coming up as you expect.

Seeing Both Sides with `FULL JOIN`s

Like many things in SQL, a `FULL JOIN` (also known as a `FULL OUTER JOIN`) is basically what it sounds like — it is a matching up of data on both sides of the `JOIN` with everything included no matter what side of the `JOIN` it is on.

`FULL JOIN`s are one of those things that seem really cool at the time you learn them and then almost never get used. You'll find an honest politician more often than you'll find a `FULL JOIN` in use. Their main purpose in life is to look at the complete relationship between data without giving preference to one side or the other. You want to know about every record on both sides of the equation — with nothing left out.

A `FULL JOIN` is perhaps best as what you would get if you could do a `LEFT JOIN` and a `RIGHT JOIN` in the same `JOIN`. You get all the records that match, based on the `JOIN` field(s). You also get any records that exist only in the left side, with `NULL`s being returned for columns from the right side. Finally, you get any records that exist only in the right side, with `NULL`s being returned for columns from the left side. Note that, when I say "finally," I don't mean to imply that they'll be last in the query. The result order you get will (unless you use an `ORDER BY` clause) depend entirely on what SQL Server thinks is the least costly way to retrieve your records.

Try It Out `FULL JOIN`s

Let's just get right to it by looking back at our last query from our section on `OUTER JOIN`s:

```
SELECT v.VendorName, a.Address
FROM VendorAddress va
JOIN Address a
    ON va.AddressID = a.AddressID
RIGHT OUTER JOIN Vendors v
    ON v.VendorID = va.VendorID
```

What we want to do here is take it a piece at a time again, and add some fields to the `SELECT` list that will let us see what's happening. First, we'll take the first two tables using a `FULL JOIN`:

```
SELECT a.Address, va.AddressID
FROM VendorAddress va
FULL JOIN Address a
      ON va.AddressID = a.AddressID
```

As it happens, a `FULL JOIN` on this section doesn’t yield us any more than a `RIGHT JOIN` would have:

Address	AddressID
-----	-----
1234 Anywhere	1
567 Main St.	3
999 1st St.	NULL
1212 Smith Ave	NULL
364 Westin	NULL

(5 row(s) affected)

But wait — there’s more! Now let’s add in the second `JOIN`:

```
SELECT a.Address, va.AddressID, v.VendorID, v.VendorName
FROM VendorAddress va
FULL JOIN Address a
      ON va.AddressID = a.AddressID
FULL JOIN Vendors v
      ON va.VendorID = v.VendorID
```

Now we have everything:

Address	AddressID	VendorID	VendorName
-----	-----	-----	-----
1234 Anywhere	1	1	Don's Database Design Shop
567 Main St.	3	2	Dave's Data
999 1st St.	NULL	NULL	NULL
1212 Smith Ave	NULL	NULL	NULL
364 Westin	NULL	NULL	NULL
NULL	NULL	3	The SQL Sequel

(6 row(s) affected)

How It Works

As you can see, we have the same two rows that we would have had with an `INNER JOIN` clause. Those are then followed by the three `Address` records that aren’t matched with anything in either table. Last, but not least, we have the one record from the `Vendors` table that wasn’t matched with anything.

Again, use a `FULL JOIN` when you want all records from both sides of the `JOIN` — matched where possible, but included even if there is no match.

CROSS JOINS

`CROSS JOINS` are very strange critters indeed. A `CROSS JOIN` differs from other `JOINS` in that there is no `ON` operator, and 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. Here's a quick example:

```
SELECT v.VendorName, a.Address
FROM Vendors v
CROSS JOIN Address a
```

Think back now — we had three records in the `Vendors` table, and five records in the `Address` table. If we're going to match every record in the `Vendors` table with every record in the `Address` table, then we should end up with $3 \times 5 = 15$ records in our CROSS JOIN:

VendorName	Address
-----	-----
Don's Database Design Shop	1234 Anywhere
Don's Database Design Shop	567 Main St.
Don's Database Design Shop	999 1st St.
Don's Database Design Shop	1212 Smith Ave
Don's Database Design Shop	364 Westin
Dave's Data	1234 Anywhere
Dave's Data	567 Main St.
Dave's Data	999 1st St.
Dave's Data	1212 Smith Ave
Dave's Data	364 Westin
The SQL Sequel	1234 Anywhere
The SQL Sequel	567 Main St.
The SQL Sequel	999 1st St.
The SQL Sequel	1212 Smith Ave
The SQL Sequel	364 Westin

(15 row(s) affected)

Indeed, that's exactly what we get.

Every time I teach a SQL class, I get asked the same question about CROSS JOINS, "Why in the world would you use something like this?" I'm told there are scientific uses for it — this makes sense to me since I know there are a number of high-level mathematical functions that make use of Cartesian products. I presume that you could read a large number of samples into table structures, and then perform your CROSS JOIN to create a Cartesian product of your sample. There is, however, a much more frequently occurring use for CROSS JOINS — the creation of test data.

When you are building up a database, that database is quite often part of a larger scale system that will need substantial testing. A reoccurring problem in testing of large-scale systems is the creation of large amounts of test data. By using a CROSS JOIN, you can do smaller amounts of data entry to create your test data in two or more tables, and then perform a CROSS JOIN against the tables to produce a much larger set of test data. You have a great example in our last query — if you needed to match a group of addresses up with a group of vendors, then that simple query yields 15 records from 8. Of course, the numbers can become far more dramatic. For example, if we created a table with 50 first names, and then created a table with 250 last names, we could CROSS JOIN them together to create a table with 12,500 unique name combinations. By investing in keying in 300 names, we suddenly get a set of test data with 12,500 names.

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.

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 they 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 Products
INNER JOIN Suppliers
    ON Products.SupplierID = Suppliers.SupplierID
```

This got us 77 rows back (again, assuming that Northwind is still as it was when it was shipped with SQL Server). 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 Products, Suppliers
WHERE Products.SupplierID = Suppliers.SupplierID
```

It’s a piece of cake, and it yields us the same 77 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

The alternative syntax for OUTER JOINS works 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:

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 discounttype, discount, s.stor_name
FROM discounts d
LEFT OUTER JOIN stores s
    ON d.stor_id = s.stor_id
```

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

```
SELECT discounttype, discount, s.stor_name
FROM discounts d, stores s
WHERE d.stor_id *= s.stor_id
```

Sure enough, we wind up with the same results as before:

```
discounttype      discount      stor_name
-----
Initial Customer   10.50        NULL
Volume Discount    6.70         NULL
Customer Discount  5.00         Bookbeat

(3 row(s) affected)
```

A RIGHT JOIN looks pretty much the same:

```
SELECT discounttype, discount, s.stor_name
FROM discounts d, stores s
WHERE d.stor_id =* s.stor_id
```

Again, we come up with the same six rows we would have under the ANSI syntax.

This is where we start to see some breakdown, as outer join support was really chaotic prior to the ANSI syntax definition. The preceding "mostly" works, but it varies from system to system and version to version.

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 v.VendorName, a.Address
FROM Vendors v
CROSS JOIN Address a
```

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

```
SELECT v.VendorName, a.Address
FROM Vendors v, Address a
```

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

VendorName	Address
-----	-----
Don's Database Design Shop	1234 Anywhere
Don's Database Design Shop	567 Main St.
Don's Database Design Shop	999 1st St.
Don's Database Design Shop	1212 Smith Ave
Don's Database Design Shop	364 Westin
Dave's Data	1234 Anywhere
Dave's Data	567 Main St.
Dave's Data	999 1st St.
Dave's Data	1212 Smith Ave
Dave's Data	364 Westin
The SQL Sequel	1234 Anywhere
The SQL Sequel	567 Main St.
The SQL Sequel	999 1st St.
The SQL Sequel	1212 Smith Ave
The SQL Sequel	364 Westin
(15 row(s) affected)	

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

The UNION

OK, enough with all the "old syntax" vs. "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 than 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 4-1.

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(s).
- ❑ 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.

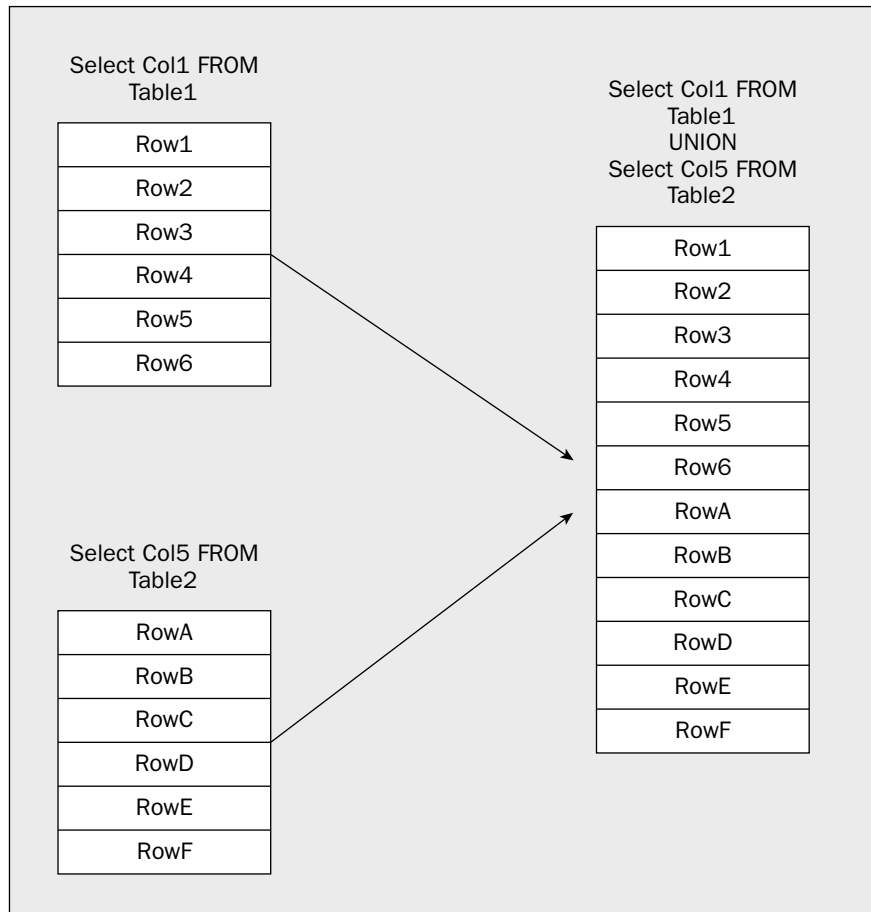


Figure 4-1

- ❑ 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 vs. explicit conversions can be found in Chapter 2). 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 look at this with an example or two.

Try It Out UNION

First, let's look at a UNION that has some practical use to it. (It's something I could see happening in the real world — albeit not all that often.) For this example, we're going to assume that it's time for the holidays, and we want to send out a New Year's card to everyone that's involved with Northwind. We want to return a list of full addresses to send our cards to including our employees, customers, and suppliers. We can do this in just one query with something like this:

```
USE Northwind

SELECT CompanyName AS Name,
       Address,
       City,
       Region,
       PostalCode,
       Country
FROM Customers

UNION

SELECT CompanyName,
       Address,
       City,
       Region,
       PostalCode,
       Country
FROM Suppliers

UNION

SELECT FirstName + ' ' + LastName,
       Address,
       City,
       Region,
       PostalCode,
       Country
FROM Employees
```

This gets us back just one result set, but it has data from all three queries:

Name	Address	City	Region	PostalCode	Country
Alfreds Futterkiste	Obere Str. 57	Berlin	NULL	12209	Germany
Ana Trujillo Emparedados y helados	Avda. de la Constitución 2222	México D.F.	NULL	5021	Mexico
Andrew Fuller	908 W. Capital Way	Tacoma	WA	98401	USA
...					
...					
...					
Wilman Kala	Keskuskatu 45	Helsinki	NULL	21240	Finland
Wolski Zajazd	ul. Filtrowa 68	Warszawa	NULL	01-012	Poland
Zaanse Snoepfabriek	Verkoop Rijnweg 22	Zaandam	NULL	9999 ZZ	Netherlands

We've got something for everyone here. Alfreds is a customer, Andrew Fuller is an employee, and Zaanse is a supplier.

As I played with this, I got some rather inconsistent results on the sorting of the query, so don't be surprised if the order of your query looks a lot different from mine. The big thing is that you should have approximately 129 rows depending on what modifications you've made to the Northwind database previously. If you want the results to be returned in a specific order, then don't forget the ORDER BY clause—for UNION statements, the ORDER BY clause needs to be part of the last query being UNIONed.

How It Works

We have our one result set from what would have been three.

SQL Server has run all three queries and essentially stacked the results one on top of the other to create one combined result set. Again, notice that the headings for the returned columns all came from the SELECT list of the first of the queries.

Moving on to a second example, I want to show you how a UNION deals with duplicate rows—it's actually just the inverse of a normal query in that it assumes you want to throw out duplicates. (In our previous queries, the assumption was that you wanted to keep everything unless you used the DISTINCT keyword.) This demo has no real-world potential, but it's quick and easy to run and see how things work.

Chapter 4

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 only return 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')

INSERT INTO UnionTest2
VALUES
    ('CCC')

INSERT INTO UnionTest2
VALUES
    ('DDD')

INSERT INTO UnionTest2
VALUES
    ('EEE')

SELECT col2
FROM UnionTest1

UNION

SELECT col4
FROM UnionTest2

PRINT 'Divider Line-----'

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" in there — just ignore them until you get to where the results of your query are visible):

```

col2
-----
DDD
EEE
AAA
BBB
CCC

(5 row(s) affected)

Divider Line-----
col2
-----
AAA
BBB
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.

The second return changed things a bit. This time we used a `UNION ALL` and the `ALL` keyword ensured that we get every row back. As such, our eliminated row from the last query suddenly reappears.

Summary

In an RDBMS, the data we want is quite frequently spread across more than one table. `JOINS` allow us to combine the data from multiple tables in a variety of ways:

- ❑ Use an `INNER JOIN` when you want to exclude non-matching fields.
- ❑ Use an `OUTER JOIN` when you want to retrieve matches wherever possible, but also want a fully inclusive data set on one side of the `JOIN`.

- ❑ Use a `FULL JOIN` when you want to retrieve matches wherever possible, but also want a fully inclusive data set of both sides of the `JOIN`.
- ❑ Use a `CROSS JOIN` when you want a Cartesian product based on the records in two tables. This is typically used in scientific environments and when you want to create test data.
- ❑ Use a `UNION` when you want the combination of the result of a second query appended to the first query.

There are two different forms of `JOIN` syntax available for `INNER` and `OUTER JOINS`. We provided the legacy syntax here to help you deal with legacy code, but the newer ANSI format presented through most of this chapter is highly preferable, as it is more readable, is not prone to the ambiguities of the older syntax, and will be supported in SQL Server for the indefinite future.

Over the course of the next few chapters, we will be learning how to build our own tables and “relate” them to each other. As we do this, the concepts of what columns to join on will become even clearer.

Exercises

1. Write a query against the Northwind database that returns one column called “SupplierName” and contains the name of the supplier of the product called Chai.
2. Using a `JOIN`, write a query that will list the name of every territory in the Northwind database that does not have an employee assigned to it. (HINT: Use an outer join!)

5

Creating and Altering Tables

Every time I teach the T-SQL code for creating databases, tables, keys, and constraints, I am asked the same question, “Can’t you just do this in the GUI tool?” The answer is an unequivocal Yes! Therefore, the next question usually follows quite shortly behind, “Then why are we spending all this time learning stuff I’ll never use?” The answer is just as unequivocal — you will use the regular syntax on a quasi-regular basis. The reality is you probably won’t actually write the code from scratch that often, but you’ll verify and edit it on the majority of all larger database projects you work on — that means that you had better know how it works.

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 Management Console to help us 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 we’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 at only a very narrow scope when trying to identify and locate the objects you name in your queries and other statements. For example, we’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 we start creating objects, it’s a good idea for us to get a solid handle on each part of the name. So let’s move from the object name left.

Schema Name (aka Ownership)

If you're utilizing schemas (most older database 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 two objects with the same name, but residing in different schemas. If you want 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, let's 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 schemas to be on the safe side.

This is another time where 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 change to be made to a user's default schema or to some other alias 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 pay if later you 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, had 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 such as 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 its support of schemas, but 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 schema. (Way, WAY too much if you ask me.) Schema also becomes important in dealing with some other facets of SQL Server such as Notification Services.

Let's focus, for now, on what schema is and how it works.

For prior releases, ownership (as it was known then) was actually a great deal like what it sounds—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 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.

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, let's 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, because 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, let's 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 an identical `CREATE` statement 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 the default schema (it could be anything). Weird but true!

*In chats I had with a few old friends at Microsoft, they seemed to be somewhat on the schema bandwagon and happy for the changes. I too am happy for the changes, but mostly because they make what you **can** do easier, not because they offer a feature that I think everyone should rush to use.*

The addition of schemas adds complexity to your database no matter what you do. While they can address organizational problems in your design, those problems can usually be dealt with in other ways that produce a much more user-friendly database. In addition, schemas, while an ANSI-compliant notion, are not supported in the same way across every major RDBMS product. This means using schemas is going to have an impact on you if you're trying to write code that can support multiple platforms.

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 the Northwind database, 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 drop-down 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.

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). We'll see a bit more about linked servers later in the book, but for now, just realize that there is one more level in our naming hierarchy, that it lets you access different servers, and that it works pretty much like the database and ownership levels work.

Now, let's just add to our previous example. If we want 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`.

Reviewing the Defaults

So let's look one last time at how the defaults work at each level of the naming hierarchy from right to left:

- ❑ **Object Name:** There isn't a default — you must supply an object name.
- ❑ **Ownership:** You can leave this off, in which case it will resolve first using the current user's name, and then, if the object name in question doesn't exist with the current user as owner, then it will try the `dbo` as the owner.

- ❑ **Database Name:** This can also be left off unless you are providing a Server Name—in which case you must provide the Database Name for SQL Servers (other server types vary depending on the specific kind of server).
- ❑ **Server Name:** You can provide the name of a linked server here, but most of the time you'll just leave this off, which will cause SQL Server to default to the server you are logged into.

If you want to skip the object owner, but still provide information regarding the database or server, then you must still provide the extra “.” for the position where the owner would be. For example, if we are logged in using the `Northwind` database on our local server, but want to refer to the `Orders` table in the `Northwind` database on a linked server called `MyOtherServer`, then we could refer to that table by using `MyOtherServer.Northwind.Orders`. Since we didn't provide a specific owner name, it will assume that either the user ID that is used to log on to the linked server or the `dbo` (in that order) is the owner of the object you want (in this case, `Orders`).

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.

Let's 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

For this part of things, we'll need to create a database called `Accounting` that we will also use when we start to create tables. The most basic syntax for the `CREATE DATABASE` statement looks like the example above.

```
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 that created it, the system administrator, and the database owner (which, if the object created was a database, is the same as the person that created it). This allows you to create things and make whatever adjustments you need to make before you explicitly allow access to your object.

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

Chapter 5

This will yield a database that looks exactly like your `model` database (we discussed the `model` database in Chapter 1). The reality of what you want is almost always different, so let's 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 terabytes>]
   [, MAXSIZE = size in kilobytes, megabytes, gigabytes, or terabytes>]
   [, FILEGROWTH = <kilobytes, megabytes, gigabytes, or terabytes/percentage>]]]
[LOG ON
  ([NAME = <'logical file name'> ,]
   FILENAME = <'file name'>
   [, SIZE = <size in kilobytes, megabytes, gigabytes, or terabytes>]
   [, MAXSIZE = size in kilobytes, megabytes, gigabytes, or terabytes>]
   [, FILEGROWTH = <kilobytes, megabytes, gigabytes, or terabytes/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>]
[;]
```

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 let's break down 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. The use of filegroups is a fairly advanced concept and is outside the scope of this book.

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. By default, your file will be located in the `\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 we're dealing with the physical database file, it will be named the same as your database with an `.mdf` extension. If we're dealing with the log, it will be named the same as the database file but 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 side note, 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 vs. 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). We'll see how to create alerts in Chapter 19.

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

Where `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 1.44, 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 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 we'll see later, 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

Hmmm. How to address this one in a beginning kinda way . . . Well, suffice to say this is a toughie, and is in no way a “beginning” kind of concept. With that in mind, here's the abridged version of what this relates to . . .

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 what are called “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 depending 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 switch is new to add an extra layer of security around access to system resources and files outside of the SQL Server context. For example, you may run a .NET assembly that touches files on your network — if so, you must identify the database that the assembly is part of as being Trustworthy.

By default this is turned off for security reasons — be certain you understand exactly what you're doing and why before you set this to on.

Building a Database

At this point, we're ready to begin building our database. Below is the statement to create it, but keep in mind that 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
```

Now is a good time to start learning about some of the informational utilities that are available with SQL Server. We saw `sp_help` in Chapter 4, but in this case, let's try running a command called `sp_helpdb`. This one is especially tailored for database structure information, and often provides better information if we'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, IsAutoCreateStatistics, IsAutoUpdateStatistics, 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:

name	fileid	Filename	filegroup	size	maxsize	growth	usage
Accounting	1	C:\Program Files\Microsoft SQL Server\mssql\data\AccountingData.mdf	PRIMARY	10240 KB	51200 KB	5120 KB	data only
AccountingLog	2	C:\Program Files\Microsoft SQL Server\mssql\data\AccountingLog.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, datatypes, 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

What's in a name? Frankly — a lot. You may recall that one of my first soapbox diatribes was back in Chapter 2 and was about names. I promised then that it wouldn't be the last you heard from me on the subject, and this won't be either.

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

Chapter 5

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 your objects — not specific rules of what SQL Server will and won't accept for names, but how you want to go about naming your 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 everyone. 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 nit pick), 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. If you recall my very first soapbox diatribe back in Chapter 1, you'll know that isn't really much of an option — 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.

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). Regardless of what you're doing in your naming, make it apply to the entire database consistently — consider having a standards document or style guide to make sure other developers utilize the same rules you do. 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 datatypes are as I described them in Chapter 2. You just need to provide a datatype immediately following the column name — there is no default datatype.

DEFAULT

We'll cover this in much more detail in our chapter on constraints, 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 datatype.

IDENTITY

The concept of an *identity* value is very important in database design. We will cover how to use identity columns in some detail in our chapters on 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, however, 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 datatype.

The usage is pretty simple; you simply include the `IDENTITY` keyword right after the datatype 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. This can, however, create havoc if you're not careful or if people are still trying to use the system as you do this, so tread carefully.

Chapter 5

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 we come to the chapter on replication.

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 to an identity column. We'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 upward 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.

"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, we can have two copies of our table running, and have them both assigned an identical identity value. While this is just fine to start with, it causes big problems when we try to bring the rows from both tables together as one replicated table. A GUID is unique across both space and time.

GUIDs are actually in increasingly widespread use 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 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 Windows 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 in terms of scope (here, we 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_dbcmtlevel` 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

We have a whole chapter coming up on constraints, so we won't spend that much time on it here. Still, it seems like a good time to review the question of what column constraints are—in short, they are restrictions and rules that you place on individual columns about the data that can be inserted into that column.

For example, if you have a column that's supposed to store the month of the year, you might define that column as being of type `tinyint`—but that wouldn't prevent someone from inserting the number 54 in

that column. Since 54 would give us bad data (it doesn't refer to a month), we might provide a constraint that says that data in that column must be between 1 and 12. We'll see how to do this in our next chapter.

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.

For example, let's say that we're working on an invoicing system. We want to store information on the quantity of an item we have sold, and at what price. It used to be fairly commonplace to go ahead and add columns to store this information, along with another column that stored the extended value (price times quantity). However, that leads to unnecessary wasting of disk space and maintenance hassles associated with when the totals and the base values get out of synch with each other. With a computed column, we can get around that by defining the value of our computed column to be whatever multiplying price by quantity creates.

Let's look at the specific syntax:

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

The first item is a little different — we're providing a column name to go with our value. This is simply the alias that we'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. Therefore, in our example of price and quantity, we might define this column as:

```
ExtendedPrice AS Price * Quantity
```

For an example using a literal, let's say that we always charge a fixed markup on our goods that is 20 percent over our cost. We could simply keep track of cost in one column, and then use a computed column for the ListPrice column:

```
ListPrice AS Cost * 1.2
```

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. For SQL Server 2005, you can now use a computed column in constraints (you must flag the computed column as persisted if you do this however).
- ❑ Another problem for previous versions (but added back in SQL Server 2000) is the ability to create indexes on computed columns. You can create indexes on computed columns, but there are special steps you must take to do so. We will discuss each of these changes to computed columns further when we explore constraints in Chapter 6 and indexing in Chapter 9.

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

I'm actually surprised that I haven't heard much debate about the use of computed columns. Rules for normalization of data say that we should not have a column in our table for information that can be derived from other columns — that's exactly what a computed column is!

I'm glad the religious zealots of normalization haven't weighed into this one much, as I like computed columns as something of a compromise. By default, you aren't storing the data twice, and you don't have issues with the derived values not agreeing with the base values because they are calculated on the fly directly from the base values. However, you still get the end result you wanted. Note that, if you index the computed column, you are indeed actually storing the data (you have to for the index). This, however, has its own benefits when it comes to read performance.

This isn't the way to do everything related to derived data, but it sure is an excellent helper for many situations.

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, we 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), it is dependent on multiple columns, then you have what would be referred to as a table constraint.

ON

Remember when we were dealing with database creation, and we said we 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 it will be placed on whatever the default filegroup is (which will be the `PRIMARY` unless you've set it to something else). We will be looking at this usage extensively in our chapter on performance tuning.

TEXTIMAGE_ON

This one is basically the same as the `ON` clause we just looked at, except that it lets you move a very specific part of the table to yet a different filegroup. This clause is only valid 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

All right, we've seen plenty; we're ready for some action, so let's build a few tables.

When we started this section, we looked at our 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
```

Our `Customers` table is going to be the first table in a database we will be putting together to track our company's accounting. We'll be looking at designing a database in a couple of chapters, but we'll go ahead and get started on our database by building a couple of tables to learn our `CREATE TABLE` statement. We'll look at most of the concepts of table construction in this section, but we'll save a few for later on in the book. That being said, let's get started building the first of several tables.

I'm going to add in a `USE <database name>` line prior to my `CREATE` code so that I'm sure that, when I 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.

Any script you create for regular use with a particular database should include a `USE` command with the name of that database. This ensures that you really are creating, altering, and dropping the objects in the database you intend. More than once have I been the victim of my own stupidity when I blindly opened up a script and executed it only to find that the wrong database was current, and any tables with the same name had been dropped (thus losing all data) and replaced by a new layout. You can also tell when other people have done this by taking a look around the master database—you'll often find several extraneous tables in that database from people running `CREATE` scripts that were meant to go somewhere else.

```
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,
    Contact       varchar(25)    NOT NULL,
    Phone        char(15)       NOT NULL,
```

```
FedIDNo      varchar(9)          NOT NULL,
DateInSystem smalldatetime      NOT NULL
)
```

This is a somewhat simplified table vs. what we would probably use in real life, but there's plenty of time to change it later (and we will).

Once we've built the table, we want to verify that it was indeed created, and that it has all the columns and types that we expect. To do this, 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 `EXEC` command is used in two different ways. This rendition is used to execute a stored procedure—in this case, a system stored procedure. We'll see the second version later when we are dealing with advanced query topics and stored procedures.

Technically speaking, you can execute a stored procedure by simply calling it (without using the `EXEC` keyword). The problem is that this only works consistently if the sproc being called is the first statement of any kind in the batch. Just having `sp_help Customers` would have worked in the place of the code above, but if you tried to run a `SELECT` statement before it—it would blow up on you. Not using `EXEC` leads to very unpredictable behavior and should be avoided.

Try executing the command, and you'll find that you get back several result sets one after another. The information retrieved includes separate result sets for:

- ❑ Table name, schema, type of table (system vs. user), and creation date
- ❑ Column names, datatypes, nullability, size, and collation
- ❑ The identity column (if one exists) including the *initial* seed and increment values
- ❑ The RowGUIDCol (if one exists)
- ❑ Filegroup information
- ❑ Index names (if any exist), types, and included columns
- ❑ Constraint names (if any), types, and included columns
- ❑ Foreign key (if any) names and columns
- ❑ The names of any schema-bound views (more on this in Chapter 10) that depend on the table

Now that we're certain that we have our table created, let's take a look at creating yet another table—the `Employees` table. This time, let's talk about what we want in the table first, and then see how you do trying to code the `CREATE` script for yourself.

Chapter 5

The `Employees` table is another fairly simple table. It should include information on:

- ☐ The employee's ID—this should be automatically generated by the system
- ☐ First name
- ☐ Optionally, middle initial
- ☐ Last name
- ☐ Title
- ☐ Social Security Number
- ☐ Salary
- ☐ The previous salary
- ☐ The amount of the last raise
- ☐ Date of hire
- ☐ Date terminated (if there is one)
- ☐ The employee's manager
- ☐ Department

Start off by trying to figure out a layout for yourself.

Before we start looking at this together, let me tell you not to worry too much if your layout isn't exactly like mine. There are as many database designs as there are database designers—and that all begins with table design. We all can have different solutions to the same problem. What you want to look for is whether you have all the concepts that needed to be addressed. That being said, let's take a look at one way to build this table.

We have a special column here. The `EmployeeID` is to be generated by the system and therefore is an excellent candidate for either an identity column or a `RowGUIDCol`. There are several reasons you might want to go one way or the other between these two, but we'll go with an identity column for a couple of reasons:

- ☐ It's going to be used by an average person. (Would you want to have to remember a GUID?)
- ☐ It incurs lower overhead.

We're now ready to start constructing our script:

```
CREATE TABLE Employees
(
    EmployeeID      int          IDENTITY NOT NULL,
```

For this column, the `NOT NULL` option has essentially been chosen for us by virtue of our use of an `IDENTITY` column. You cannot allow `NULL` values in an `IDENTITY` column. Note that, depending on our server settings, we will, most likely, still need to include our `NOT NULL` option (if we leave it to the default we may get an error depending on whether the default allows `NULL`s).

Next up, we want to add in our name columns. I usually allow approximately 25 characters for names. Most names are far shorter than that, but I've bumped into enough that were rather lengthy (especially since hyphenated names have become so popular) that I allow for the extra room. In addition, I make use of a variable-length datatype for two reasons:

- ❑ To recapture the space of a column that is defined somewhat longer than the actual data usually is (retrieve blank space)
- ❑ To simplify searches in the `WHERE` clause — fixed-length columns are padded with spaces, which requires extra planning when performing comparisons against fields of this type

For the code that you write directly in T-SQL, SQL Server will automatically adjust to the padded spaces issue — that is, an 'xx' placed in a `char(5)` will be treated as being equal (if compared) to an 'xx' placed in a `varchar(5)` — this is not, however, true in your client APIs such as ADO and ADO.NET. If you connect to a `char(5)` in ADO, then an 'xx' will evaluate to `xx` with three spaces after it — if you compare it to 'xx', it will evaluate to `False`. An 'xx' placed in a `varchar(5)`, however, will automatically have any trailing spaces trimmed, and comparing it to 'xx' in ADO will evaluate to `True`.

The exception in this case is the middle initial. Since we really only need to allow for one character here, recapture of space is not an issue. Indeed, a variable-length datatype would actually use more space in this case, since a `varchar` needs not only the space to store the data, but also a small amount of overhead space to keep track of how long the data is. In addition, ease of search is also not an issue since, if we have any value in the field at all, there isn't enough room left for padded spaces.

Since a name for an employee is a critical item, we will not allow any `NULL` values in the first and last name columns. Middle initial is not nearly so critical (indeed, some people in the U.S. don't have a middle name at all, while my editor tells me that it's not uncommon for Brits to have several), so we will allow a `NULL` for that field only:

FirstName	<code>varchar(25)</code>	<code>NOT NULL,</code>
MiddleInitial	<code>char(1)</code>	<code>NULL,</code>
LastName	<code>varchar(25)</code>	<code>NOT NULL,</code>

Next up is the employee's title. We must know what they are doing if we're going to be cutting them a paycheck, so we will also make this a required field:

Title	<code>varchar(25)</code>	<code>NOT NULL,</code>

In that same paycheck vein, we must know their Social Security Number (or similar identification number outside the U.S.) in order to report for taxes. In this case, we'll use a `varchar` and allow up to 11 characters, as these identification numbers are different lengths in different countries. If you know your application is only going to require SSNs from the U.S., then you'll probably want to make it `char(11)` instead:

SSN	<code>varchar(11)</code>	<code>NOT NULL,</code>

We must know how much to pay the employees — that seems simple enough — but what comes next is a little different. When we add in the prior salary and the amount of the last raise, we get into a situation

where we could use a computed column. The new salary is the sum of the previous salary and the amount of the last raise. The `Salary` amount is something that we might use quite regularly — indeed we might want an index on it to help with ranged queries., but for various reasons I don’t want to do that here (we’ll talk about the ramifications of indexes on computed columns in Chapter 9), so I’m going to use `LastRaise` as my computed column:

<code>Salary</code>	<code>money</code>	<code>NOT NULL,</code>
<code>PriorSalary</code>	<code>money</code>	<code>NOT NULL,</code>
<code>LastRaise AS</code>	<code>Salary - PriorSalary,</code>	

If we hired them, then we must know the date of hire — so that will also be required:

<code>HireDate</code>	<code>smalldatetime</code>	<code>NOT NULL,</code>

Note that I’ve chosen to use a `smalldatetime` datatype rather than the standard `datetime` to save space. The `datetime` datatype will store information down to additional fractions of seconds, plus it will save a wider range of dates. Since we’re primarily interested in the date of hire, not the time, and since we are dealing with a limited range of calendar dates (say, back 50 years and ahead a century or so), the `smalldatetime` will meet our needs and take up half the space.

Date and time fields are somewhat of a double-edged sword. On one hand, it’s very nice to save storage space and network bandwidth by using the smaller datatype. On the other hand, you’ll find that `smalldatetime` is incompatible with some other language datatypes (including Visual Basic). Even going with the normal `datetime` is no guarantee of safety from this last problem though — some data access models pretty much require you to pass a date in as a `varchar` and allow for implicit conversion to a `datetime` field.

The date of termination is something we may not know (we’d like to think that some employees are still working for us), so we’ll need to leave it nullable:

<code>TerminationDate</code>	<code>smalldatetime</code>	<code>NULL,</code>

We absolutely want to know who the employee is reporting to (somebody must have hired them!) and what department they are working in:

<code>ManagerEmpID</code>	<code>int</code>	<code>NOT NULL,</code>
<code>Department</code>	<code>varchar(25)</code>	<code>NOT NULL</code>
<code>)</code>		

So, just for clarity, let’s look at the entire script to create this table:

```
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
)

```

Again, I would recommend executing `sp_help` on this table to verify that the table was created as you expected.

The ALTER Statement

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

Much like the `CREATE` statement, our `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. We'll see the beginnings of issues with this statement right away, and things will get really interesting (read: convoluted and confusing!) when we deal with this even further in our next chapter (when we deal with constraints).

ALTER DATABASE

Let's get right into it by taking a look at changing our database. We'll actually make a couple of changes just so we 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, let's take a look again at what we already have:

```
EXEC sp_helpdb Accounting
```

Notice that I didn't put the quotation marks in this time as I did when we used this stored proc earlier. That's because this system procedure, like many of them, accepts a special datatype called `sysname`. As long as what you pass in is a name of a valid object in the system, the quotes are optional for this datatype.

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

Chapter 5

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, IsAutoCreateStatistics, IsAutoUpdateStatistics, 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

Let's say we want to change things a bit. For example, let's say that we know that we are going to be doing a large import into our database. Currently, our database is only 15MB in size — that doesn't hold much these days. Since we have Autogrow turned on, we could just start our 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 we 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 we know that we're going to be getting up to 100MB of data, why not just do it in one shot? To do this, we 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 us (just kidding!).

So, after looking at all that gobbledygook, let's just worry about what we need to expand our database out to 100MB:

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

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

```
The command(s) completed successfully.
```

Gee—how informative . . . So, we'd better check on things for ourselves:

```
EXEC sp_helpdb Accounting
```

name	db_size	Owner	dbid	created	status	compatibility_level
Accounting	105.00 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, IsAutoCreateStatistics, IsAutoUpdateStatistics, IsFullTextEnabled	90

name	fileid	Filename	filegroup	size	maxsize	growth	usage
Accounting	1	c:\Program Files\Microsoft SQL Server\MSSQL.1\mssql\data\AccountingData.mdf	PRIMARY	102400 KB	102400 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

As you can see, we've succeeded in increasing our size up to 100MB. One thing worth noticing is that, even though we exceeded the previous maximum size of 51,200KB, we didn't get an error. This is because we *explicitly* increased the size. It was, therefore, implied that we must have wanted to increase the maximum, too. If we had done things our original way of just letting SQL Server expand things as necessary, our 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 our 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 us the ability to gracefully force other users out of the database so that we may complete our 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. As such, we will consider further discussion out of the scope of this book.

ALTER TABLE

A far, far more common need is the situation where we need to change the makeup of our table. This can range from simple things like adding a new column to more complex issues such as changing a datatype.

Let’s 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 let's start an example of using this by looking back at our `Employees` table in the Accounting database:

```
EXEC sp_help Employees
```

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

Column_name	Type	Computed	Length	Prec	Scale	Nullable
EmployeeID	int	no	4	10	0	no
FirstName	varchar	no	25			no
MiddleInitial	char	no	1			yes
LastName	varchar	no	25			no
Title	varchar	no	25			no
SSN	varchar	no	11			no
Salary	money	no	8	19	4	no
PriorSalary	money	no	8	19	4	no
LastRaise	money	yes	8	19	4	yes

Column_name	Type	Computed	Length	Prec	Scale	Nullable
HireDate	smalldatetime	no	4			no
TerminationDate	smalldatetime	no	4			yes
ManagerEmpID	int	no	4	10	0	no
Department	varchar	no	25			no

Let's say that we've decided we'd like to keep previous employer information on our employees (probably so we know who will be trying to recruit the good ones back!). That just involves adding another column, and really isn't all that tough. The syntax looks much like it did with our `CREATE TABLE` statement except that it has obvious alterations to it:

```
ALTER TABLE Employees
    ADD
        PreviousEmployer    varchar(30)    NULL
```

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

```
ALTER TABLE Employees
    ADD
        DateOfBirth        datetime    NULL,
        LastRaiseDate       datetime    NOT NULL
        DEFAULT '2005-01-01'
```

Notice the `DEFAULT` I slid in here. We haven't really looked at these yet (they are in our 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. We 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 our table.

Before we go away from this topic for now, let's take a look at what we've added:

```
EXEC sp_help Employees
```

Column_name	Type	Computed	Length	Prec	Scale	Nullable
EmployeeID	int	no	4	10	0	no
FirstName	varchar	no	25			no
MiddleInitial	char	no	1			yes
LastName	varchar	no	25			no
Title	varchar	no	25			no
SSN	varchar	no	11			no

Table continued on following page

Column_name	Type	Computed	Length	Prec	Scale	Nullable
Salary	money	no	8	19	4	no
PriorSalary	money	no	8	19	4	no
LastRaise	money	yes	8	19	4	yes
HireDate	smalldatetime	no	4			no
TerminationDate	smalldatetime	no	4			yes
ManagerEmpID	int	no	4	10	0	no
Department	varchar	no	25			no
PreviousEmployer	varchar	no	30			yes
DateOfBirth	datetime	no	8			yes
LastRaiseDate	datetime	no	8			no

As you can see, all of our 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 re-create your indexes as part of the build script to create your new version of the table—yuck!

But wait! There's more! While we 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 re-create 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, procs, triggers, etc.). It goes like this:

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

Actually, this is about as simple as SQL statements get. We could drop both of our tables at the same time if we wanted:

```
USE Accounting  
  
DROP TABLE Customers, Employees
```

And this deletes them both.

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.

Remember my comment at the beginning of this chapter about putting a `USE` statement at the top of your scripts? Well, here’s an example of why it’s so important — the *Northwind* database also has tables called *Customers* and *Employees*. You wouldn’t want those gone, now, would you? As it happens, some issues of drop order would prevent you from accidentally dropping those tables with the above command, but it wouldn’t prevent you from dropping the `[Order Details]` table, or, say, *Shipments*.

The syntax is very much the same for dropping the entire database. Now let’s 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 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 we 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

We'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. Let's take a look at the graphical tool in the Management Studio that allows us to build and relate tables. From this point on, we'll not only be dealing with code, but with the tool that can generate much of that code for us.

Creating a Database Using the Management Studio

If you run the SQL Server Management Studio and expand the Databases node, you should see something like Figure 5-1.

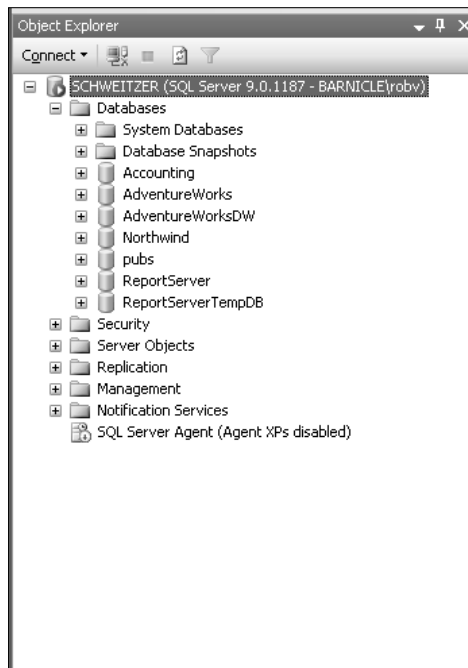


Figure 5-1

If you look closely at this screenshot, you'll see that my Accounting database is still showing even though we just dropped it in the previous example. You may or may not wind up seeing this, depending on whether you already had the Management Studio open when you dropped the database or you opened it after you dropped the database in QA.

Why the difference? Well, in earlier versions of SQL Server, the tools that are now the Management Studio refreshed information such as the available databases regularly. Now it updates only when it knows it has a reason to (for example, you deleted something by using the Management Studio Object Explorer instead of a Query window, or perhaps you explicitly chose to refresh). The reason for the change was performance. The old 6.5 Enterprise Manager used to be a slug performance-wise because it

was constantly making round trips to “poll” the server. The newer approach performs much better, but doesn’t necessarily have the most up-to-date information.

The bottom line on this is that, if you see something in the Management Studio that you don’t expect to, try pressing F5 (refresh), and it should update things for you.

Now try right-clicking on the Databases node, and choose the New Database . . . option.

This will pull up the Database Properties dialog box, and allow you to fill in the information on how you want your database created. We’ll use the same choices that we did when we created the `Accounting` database at the beginning of the chapter. First comes the basic name and size info, as shown in Figure 5-2.

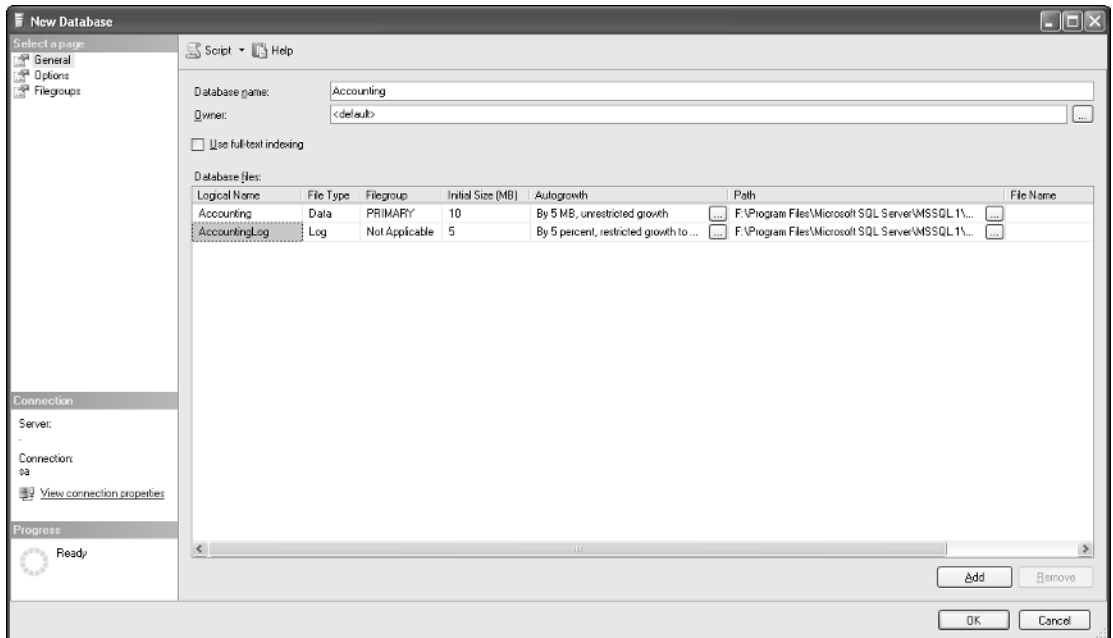


Figure 5-2

This entire tab on the dialog is new, so let’s take a look at things.

First, the name — this is pretty basic. We called it `Accounting` before, and, because we deleted the first one we created, there’s no reason not to call it that again.

Next comes our file name, size, and growth information.

I’ve expanded the dialog out manually to make sure you could see everything. You may see less than what’s pictured here as the default size of the dialog is not nearly enough to show it all — just grab a corner of the dialog and expand it to see the additional information

Next let’s move on to the Options tab, which contains a host of additional settings, as shown in Figure 5-3.

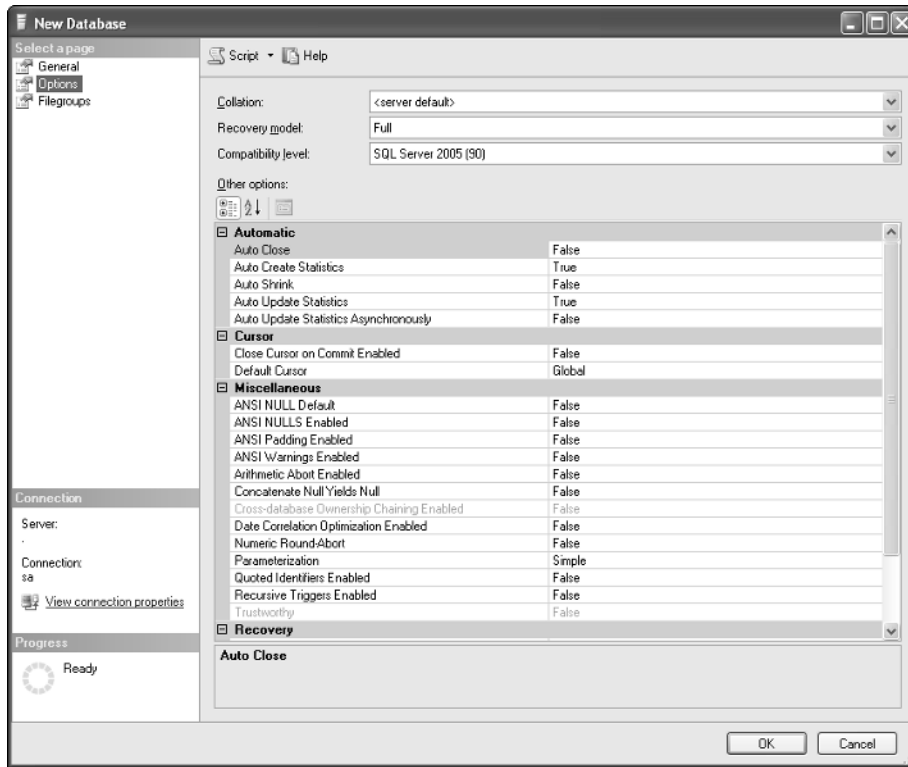


Figure 5-3

Perhaps the most interesting thing here though is the collation name. Beginning with SQL Server 2000, we gained the choice of having each database (and, indeed, individual columns if we wish) have its own collation. For the vast majority of installs, you'll want to stick with whatever the server default was set to when the server was installed (presumably, someone had already thought this out fairly well). However, you can change it for just the current database by setting it here.

“Why,” you may ask, “would I want a different collation?” Well, in the English-speaking world, a common need for specific collations would be that some applications are written expecting an “a” to be the same as an “A” — while others are expecting case sensitivity (“a” is not the same as “A”). In the old days, we would have to have separate servers set up in order to handle this. Another, non-English example would be dialect differences that are found within many countries of the world — even where they speak the same general language.

Next comes the compatibility level. This will control whether certain SQL Server 2005 syntax and keywords are supported or not. As you might expect from the name of this setting, the goal is to allow you to rollback to keywords and functional behavior that more closely matches older versions if your particular application happens to need that.

The remaining properties will vary from install to install, but work as I described them earlier in the chapter.

OK, given that the other settings are pretty much standard fare to what we've seen earlier in the chapter, let's go ahead and try it out. Click OK and, after a brief pause to actually create the database, you'll see it added to the tree.

Now expand the tree to show the various items underneath the Accounting node, and select the Database Diagrams node. Right-click it, and you'll get a dialog indicating that the database is missing some objects it needs to support database diagramming, as shown in Figure 5-4. Click Yes.

Note that you should only see this the first time a diagram is being created for that database. SQL Server keeps track of diagrams inside special tables that it only creates in your database if you are going to actually create a diagram that will use them.

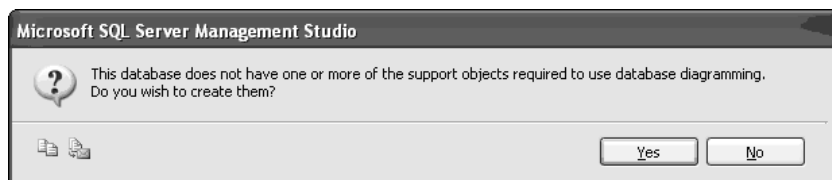


Figure 5-4

With that, you'll get an Add Table dialog, as shown in Figure 5-5. This lets us decide what tables we want to include in our diagram — we can create multiple diagrams if we wish, potentially each covering some subsection — or *submodel* — of the overall database schema. In this case, we have only one table showing (recall that we dropped the Customers and Orders tables a while back — leaving us just an empty database in terms of what tables we created for ourselves).



Figure 5-5

For now, just click Cancel — you should get an empty diagram screen. The nice part is that you can add a table by either right-clicking and choosing the appropriate option, or by clicking on the New table icon in the toolbar. When you choose new table, SQL Server will ask you for the name you want to give your new table. You will then get a mildly helpful dialog box that lets you fill in your table one piece at a time — complete with labels for what you need to fill out, as shown in Figure 5-6.

Customers *			
	Column Name	Data Type	Allow Nulls
☐	CustomerNo	int	☐
☐	CustomerName	varbinary(30)	☐
☐	Address1	varchar(50)	☐
☐	Address2	varchar(50)	☐
☐	City	varchar(50)	☐
☐	State	char(2)	☐
☐	Zip	varchar(50)	☐
☐	Contact	varchar(50)	☐
☐	Phone	char(15)	☐
☐	FedIDNo	varchar(9)	☐
☒	DateInSystem	smalldatetime	☐
☐			☐

Figure 5-6

I’ve gone ahead and filled in the columns as they were in our original Customers table, but we also need to define our first column as being an identity column. Unfortunately, we don’t appear to have any way of doing that with the default grid here. To change what items we can define for our table, we need to right-click in the editing dialog, and select Table View ⇄ Modify Custom.

We then get a list of items from which we can choose, shown in Figure 5-7. For now, we’ll just select the extra item we need — Identity and it’s associated elements Seed and Increment.

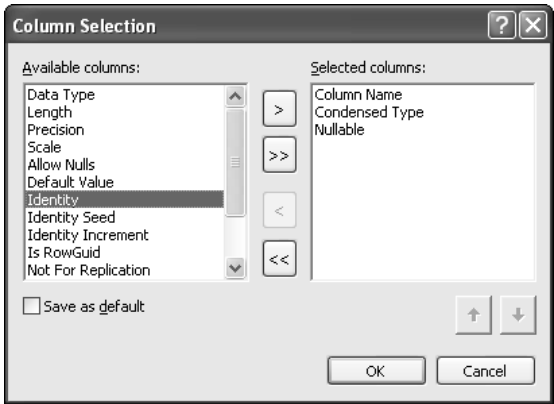


Figure 5-7

Now go back to our editing dialog and select Table View ⇄ Custom to view the identity column (see Figure 5-8), and we’re ready to fill in our table definition.

OK, so SQL Server can be a bit temperamental on this. If you do not check the box to make this the default, then SQL Server will change what your “custom” view looks like, but it will not make the custom view the active one — the result is that you won’t see the changes you made as you exit the dialog.

So, again, make sure that **after** changing the view, you right-click and select Table View ⇄ Custom again. It should then look like Figure 5-8.

Column Name	Condensed Type	Nullable	Identity	Identity Seed	Identity Increment
CustomerNo	int	No	☒	1	1
CustomerName	varbinary(30)	No	☐		
Address1	varchar(50)	No	☐		
Address2	varchar(50)	No	☐		
City	varchar(50)	No	☐		
State	char(2)	No	☐		
Zip	varchar(50)	No	☐		
Contact	varchar(50)	No	☐		
Phone	char(15)	No	☐		
FedIDNo	varchar(9)	No	☐		
DateInSystem	smalldatetime	No	☐		

Figure 5-8

Once you have the table filled out, you can save the changes, and that will create your table for you.

This is really a point of personal preference, but I prefer to set the view down to just column names at this point. You can do this by clicking on the Show icon on the toolbar or, as I prefer, by right-clicking the table and choosing Table View ⇄ Column Names. I find that this saves a lot of screen real estate and makes more room for me to work on additional tables.

Now try to add in the `Employees` table as we had it defined earlier in the chapter. The steps should be pretty much as they were for the `Customers` table, with just one little hitch — we have a computed column. To deal with the computed column, just select `Modify Custom` again (from the right-click menu), and add the “formula” column. Then, simply add the proper formula (in this case, `Salary-PriorSalary`). When you have all the columns entered, save your new table (accepting the confirmation dialog) and your diagram should have two tables in it (see Figure 5-9).

CustomerNo
CustomerName
Address1
Address2
City
State
Zip
Contact
Phone
FedIDNo
DateInSystem

EmployeeID
FirstName
MiddleInitial
LastName
Title
SSN
Salary
PriorSalary
LastRaise
HireDate
TerminationDate
ManagerEmpID
Department

Figure 5-9

It's very important to understand that the diagramming tool that is included with SQL Server is not designed to be everything to everyone.

Presumably, since you are reading this part of this book, you are just starting out on your database journey — this tool will probably be adequate for you for a while. Eventually, you may want to take a look at some more advanced (and far more expensive) tools to help you with your database design.

Backing into the Code: The Basics of Creating Scripts with the Management Studio

One last quick introduction before we exit this chapter — we want to see the basics of having the Management Studio write our scripts for us. For now, we are going to do this as something of a quick and dirty introduction. Later on, after we've learned about the many objects that the scripting tool references, we will take a more advanced look.

To generate scripts, we go into the Management Studio and right-click on the database for which we want to generate scripts. (In this case, we're going to generate scripts on our Accounting database.) On the pop-up menu, choose Script Database As → CREATE To → New Query Editor Window, shown in Figure 5-10.

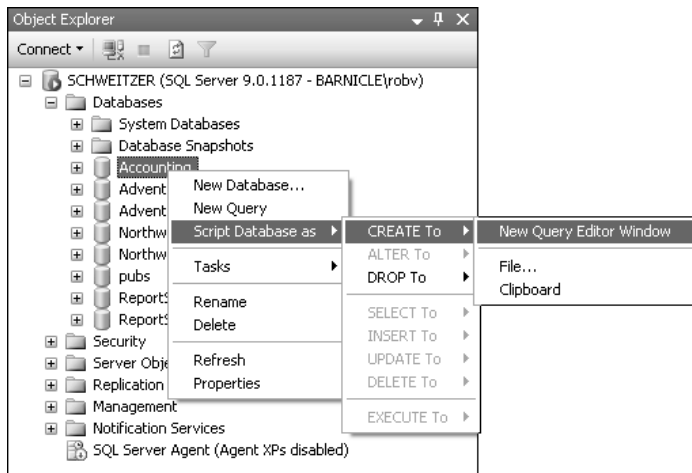


Figure 5-10

Whoa! SQL Server generates a heck of a lot more code than we saw when we created our database to begin with. Don't panic, however — all it is doing is being very explicit in scripting major database settings rather than relying on defaults as we did when we scripted it ourselves.

Note that we are not limited to scripting the database — if you want to script other objects in the database, just navigate and right click on them much the way that you right-clicked on the Accounting database and, boom!, you've got yourself a SQL Script.

As you can see, scripting couldn't be much easier. Once you get a complex database put together, it still isn't quite as easy as it seems in this particular demonstration, but it is a lot easier than writing it all out by hand. The reality is that it really is pretty simple once you learn what the scripting options are, and we'll learn much more about those later in the book.

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 we can use the Management Studio for in order to simplify our lives, and make design and scripting simpler.

At this point, you're ready to start getting into some hardcore details about how to lay out your tables, and a discussion on the concepts of normalization and more general database design. I am, however, actually going to make you wait another chapter before we get there, so that we can talk about constraints and keys somewhat before hitting the design issues.

Exercises

1. Using the Management Studio's script generator, generate SQL for both the Customers and the Employees tables.
2. Without using the Management Studio, script a database called "MyDB" with a starting database size of 17MB and a starting log size of 5MB — set both the log and the database to grow in 5MB increments.
3. Create a Table called Foo with a single variable length character field called "Col1" — limit the size of Col1 to 50 characters.

6

Constraints

You've heard me talk about them, but now it's time to look at them seriously — it's time to deal with 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 back in Chapters 1 and 2, where ensuring data integrity is not the responsibility of the programs that use your database, but rather the responsibility of the database itself. If you think about it, this is really cool. Data is inserted, updated, and deleted from the database by many sources. Even in stand-alone 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, you can actually run into situations where literally hundreds of different access paths 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* 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 (datatypes and triggers are among the other worker bees of data integrity). Well let's take a look.

Chapter 6

In this chapter, we'll be looking at the three different 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, including:

- ❑ 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 have been added with SQL Server 2005. We'll look at cascading and other ANSI referential integrity actions in detail when we look at FOREIGN KEY constraints.

We'll also take a very cursory look at triggers and stored procedures (there will be much more on these later) as a method of implementing data integrity rules.

Types of Constraints

There are a number of different ways to implement constraints, but each of them falls into one of three categories—entity, domain, or referential integrity constraints, as illustrated in Figure 6-1.

Domain Constraints

Domain constraints deal with one or more columns. What we're talking about here is ensuring 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.

Order ID	Line Item	Part Number	Unit Price
1	1	0R2400	45.00
2	1	3I8764	75.36
3	1	2P9500	98.50
3	2	3X5436	12.00
4	1	1R5564	.98
4	2	8H0805	457.00
4	3	3H6665	65.00
5	1	7Y0024	26.75

Figure 6-1

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

We'll see this kind of constraint in 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 problems.

"What," you say, "a unique column? Doesn't that mean it's a domain constraint?" No, it doesn't. We're not saying that a column has to meet any particular format, or that the value has to be greater or less than anything. What we're saying is that for *this* row, the same value can't already exist in some other row.

We'll see this kind of constraint in 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 either the same table or, far more typically, a different table.

Let's say that we are taking orders for a product, and that we accept credit cards. In order to be paid by the credit card company, we need to have some form of merchant agreement with that company. We don't want our employees to take credit cards from companies from which we're not going to be paid back. That's where referential integrity comes in—it allows us to build what we would call a *domain* or *lookup table*. A domain table is a table whose sole purpose in life is to provide a limited list of acceptable values. In our case, we might build a table that looks something like this:

CreditCardID	CreditCard
1	VISA
2	MasterCard
3	Discover Card
4	American Express

We can then build one or more tables that *reference* the `CreditCardID` column of our 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. For each row that we insert into the referencing table, it will have to have a value that is in our domain list (it will have to have a corresponding row in the `CreditCard` table).

We'll see more of this as we learn about `FOREIGN KEY` constraints later in this chapter.

Constraint Naming

Before we get down to the nitty-gritty constraints, we'll digress for a moment and address the issue of naming constraints.

For each of the different types of constraints that we 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_Employees_145C0A3F`. This is a SQL Server-generated name for a primary key on the `Employees` table of the Accounting database, which we will create later in the chapter—the `PK` is for primary key (which is the major thing that makes it useful), the `Employees` is for the `Employees` 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_Employees`.

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, we know that it's a CHECK constraint, but we know nothing of what the nature of the CHECK is.

Since we can have multiple CHECK constraints on a table, you could wind up with all these 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.

Personally, I either use a combination of type of constraint together 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 we'll use in this book, how exactly you name things is really not all that important. What is important is that you:

- ☐ 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. For this chapter, we'll only take a look at the first three of these, as they provide constraints on the database.

An inversion key is basically just any index (we cover indexes in Chapter 9) that does not apply some form of constraint to the table (primary key, foreign key, unique). 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 — this will be one of the most important concepts you'll read about in this book, and will become absolutely critical as we move on to normalization in our design chapter.

PRIMARY KEY Constraints

Before we define what a primary key actually is, let's digress slightly into a brief discussion of relational databases. Relational databases are constructed on the idea of being able to "relate" data. Therefore, it

Chapter 6

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 allows you to accurately reference a record from another table in the database — so forming a relation between those two tables.

This is a wildly different concept from what we had with our 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, 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. While a GUID can certainly be used as a primary key, they incur some overhead, and are usually not called for when we're only dealing with 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 severely 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.

Situations where you can have multiple rows that are logically identical are 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 being part of that primary key, and that unique value serves as an identifier for each row in that table. How do we create a primary key? Actually, there are two ways. You can create the primary key either in your `CREATE TABLE` command or with an `ALTER TABLE` command.

Creating the Primary Key at Table Creation

Let's review one of our `CREATE TABLE` statements from the last chapter:

```
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,
Contact       varchar(25)     NOT NULL,
Phone         char(15)        NOT NULL,
FedIDNo       varchar(9)      NOT NULL,
DateInSystem  smalldatetime   NOT NULL
)

```

This `CREATE` statement should seem old hat by now, but it's missing a very important piece — our `PRIMARY KEY` constraint. We want to identify `CustomerNo` as our primary key. Why `CustomerNo`? Well, we'll look into what makes a good primary key in the next chapter, but for now, just think about it a bit — do we 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 re-inventing the wheel here.

To alter our `CREATE TABLE` statement to include a `PRIMARY KEY` constraint, we just add in the constraint information right after the column(s) that we want to be part of our primary key. In this case, we would use:

```

CREATE TABLE Customers
(
    CustomerNo    int    IDENTITY    NOT NULL
                PRIMARY KEY,
    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,
    Contact       varchar(25)        NOT NULL,
    Phone         char(15)           NOT NULL,
    FedIDNo       varchar(9)         NOT NULL,
    DateInSystem  smalldatetime      NOT NULL
)

```

Note that, if you want to try out this code, you may need to first `DROP` the existing table by issuing a `DROP TABLE Customers` command. Notice that we altered one line (all we did was remove the comma) and added some code on a second line for that column. In a word, it was easy! Again, we just added one simple keyword (OK, so it's two words, but they operate as one) and we now have ourselves a primary key.

Creating a Primary Key on an Existing Table

Now, what if we already have a table and we want to set the primary key? That's also easy — we'll do that for our `Employees` table:

```

USE Accounting

ALTER TABLE Employees
    ADD CONSTRAINT PK_EmployeeID
    PRIMARY KEY (EmployeeID)

```


Our ALTER command tells SQL Server:

- ❑ That we are adding something to the table (we could also be dropping something from the table if we so chose)
- ❑ What it is that we're adding (a constraint)
- ❑ What we want to name the constraint (to allow us to address the constraint directly later)
- ❑ The type of constraint (PRIMARY KEY)
- ❑ The column(s) that the constraint applies to

FOREIGN KEY Constraints

Foreign keys are both 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 either 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 let's do it by example.

When I say that a value must be "set to NULL," I'm referring to how the actual INSERT statement looks. As we'll learn in a moment, the data may actually look slightly different once it gets in the table depending on what options you've set in your FOREIGN KEY declaration.

Let's create another table in our Accounting database called Orders. One thing you'll notice in this CREATE script is that we're going to use both a primary key and a foreign key. A primary key, as we will see as we continue through the design, is a critical part of a table. Our foreign key is added to the script in almost exactly the same way as our primary key was, except that we must say what we are referencing. The syntax goes on the column or columns that we are placing our 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}]
```

Try It Out Creating a Table with a Foreign Key

For the moment, we're going to ignore the ON clause. That leaves us, for our 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 either a `PRIMARY KEY` or a `UNIQUE` constraint defined on it (we'll discuss `UNIQUE` constraints later in the chapter).

It's also worth noting that primary and foreign keys can exist on the same column. You can see an example of this in the Northwind database with the `Order Details` table. The primary key is composed of both the `OrderID` and `ProductID` columns — both of these are also foreign keys, and reference the `Orders` and `Products` tables respectively. We'll actually create a table later in the chapter that has a column that is both a primary key and a foreign key.

How It Works

Once 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 our new `Orders` table, and you'll get information back giving you the names, criteria, and status for all the constraints on the table. At this point, our `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. We will explore the meaning of this further in Chapter 9.

Our new foreign key has been referenced in the physical definition of our table, and is now an integral part of our table. As we discussed back in Chapter 1, the database is in charge of its own integrity — our foreign key enforces one constraint on our data and makes sure our database integrity remains intact.

Unlike primary keys, we are not limited to just one foreign key on a table. We 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

Just like with primary keys, or any constraint for that matter, we have situations where we want to add our foreign key to a table that already exists. This process is similar to creating a primary key.

Try It Out Adding a Foreign Key to an Existing Table

Let's add another foreign key to our `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, we need to be able to uniquely identify a target record in the referenced table. As I've

Chapter 6

already mentioned, you can do this by referencing either a primary key or a column with a `UNIQUE` constraint. In this case, we'll make use of the existing primary key that we 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` again against the `Orders` table and you'll see that our new constraint has been added.

How It Works

Our 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 now also not allow a row to be inserted into the `Orders` table where the referenced `Employee` in charge of that order is not 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 in order to maintain the integrity of the database.

Note that while we've added two foreign keys, there is still a line down at the bottom of our `sp_helpconstraint` results (or under the Messages tab if you have Results in Grid selected) that says No foreign keys reference this table—this is telling us that, while we 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 now referenced by our 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 is actually used with regularity.

Before we actually create this self-referencing constraint that references a required (non-nullable) field that's based on an identity column, it's rather critical that we get at least one row in the table prior to the foreign key being added. 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 here is to go ahead and create the foreign key, but then disable it when adding the first row. We'll learn about disabling constraints a little later in this chapter.

OK — because this is a table that's referencing a column based on an identity column, we need to get a primer row into the table before we add our 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 we have a primer row in, we can add in our foreign key. In an `ALTER` situation, this works just the same as any other foreign key definition. We 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. We already have our `Employees` table set up at this point, but if we were creating it from scratch, here would be 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're going to get an error. Why? Well, when we established the reference in our `Orders` table to the `Employees` table, the two tables became "schema-bound"—that is, the `Employees` table now knows that it has what is called a dependency on it. SQL Server will not 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 will allow you to delete the `Employees` table (or the `Customers` table for that matter).

In addition, doing the self-referencing foreign key in the constraint doesn't allow us to get our 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 bi-directional; 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 we do something to the parent (by doing so, they avoid orphans). The default behavior is for SQL Server to "restrict" the parent row from being deleted if any child rows exist. Sometimes, however, we would rather automatically delete any dependent records than prevent the deletion of the referenced record. The same notion applies to updates to records where we would like the dependent record to automatically reference to 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 either NULL or whatever the default value is 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 we implement cascading actions in SQL Server? All we need is a modification to the syntax we use when declaring our foreign key—we just add the `ON` clause that we skipped at the beginning of this section.

Let's check this out by adding a new table to our Accounting database. We'll make this a table to store the individual line items in an order, and we'll 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 we have a whole lot going on, so let's take it apart piece by piece.

Before we get too far into looking at the foreign key aspects of this — notice something about how the primary key was done here. Instead of placing the declaration immediately after the key, I decided to declare it as a separate constraint item. This helps facilitate both the multi-column primary key (which therefore could not be declared as a column constraint) and the clarity of the overall `CREATE TABLE` statement. Likewise, I could have declared the foreign key either 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 our foreign key is also part of our primary key — this is not at all uncommon in child tables, and is actually almost always the case for associate tables (more on this next chapter). Just remember that each constraint stands alone — you add, change, or delete each of them independently.

Next, look at our foreign key declaration:

```
FOREIGN KEY    (OrderID)
REFERENCES     Orders (OrderID)
```

We've declared our `OrderID` as being dependent on a "foreign" column. In this case, it's for a column (also called `OrderID`) in a separate table (`Orders`), but, as we saw earlier in the chapter, it could just as easily have been in the same table if that's what our need was.

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 never-ending loop.

Now, to get to the heart of our cascading issue, however, we need to look at our `ON` clauses:

```
ON UPDATE     NO ACTION
ON DELETE     CASCADE
```

We've 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. For situations where the parent record (in the `Orders` table) is updated, we've said that we would not like that update to be cascaded to our child table (`OrderDetails`). For illustration purposes, however, I've chosen a `CASCADE` for deletes.

Note that `NO ACTION` is the default, and so specifying this in our code is optional. The fact that this keyword was not supported until SQL Server 2000 has caused the "typical" way of coding this to be to leave out the `NO ACTION` keywords. If you don't need backward support, I would encourage you to include the `NO ACTION` explicitly to make your intent clear.

Chapter 6

Let’s try an insert into our 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, we haven’t inserted anything into our Orders table yet, so how can we 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 exists in situations 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 along 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, save one or two dependency levels.

OK, so, in order to get our row into our OrderDetails table, we must also have a record already in the Orders table. Unfortunately, getting a row into the Orders table requires that we have one in the Customers table (remember that foreign key we built on Orders?). So, let’s 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()
    )
```

Now we have a customer, so let’s select the record back out just to be sure:

Customer No	1
Customer Name	Billy Bob’s Shoes
Address 1	123 Main Street

Address 2	
City	Vancouver
State	WA
Zip	98685
Contact	Billy Bob
Phone	(360) 555-1234
FedIDNo	931234567
DateInSystem	2000-07-10T17:00

So—we have a `CustomerID` of 1 (your number may be different depending on what experimentation you’ve done). We’ll take that number and use it in our next `INSERT` (into `Orders` finally). Let’s insert an order for `CustomerID` 1:

```
INSERT INTO Orders
  (CustomerNo, OrderDate, EmployeeID)
VALUES
  (1, GETDATE(), 1)
```

This time, things should work fine.

It’s worth noting that the reason that we don’t still get one more error here is that we already inserted that primer row in the `Employees` table; otherwise, we would have needed 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, we’re ready for our insert into the `OrderDetails` table. Just to help with a `CASCADE` example we’re going to be doing in the moment, we’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)
```

So, let’s verify things by running a `SELECT`:

```
SELECT OrderID, PartNo FROM OrderDetails
```

This gets us back our expected two rows:

Chapter 6

OrderID	PartNo
-----	-----
1	0R2400
1	4X4525

(2 row(s) affected)

Now that we have our data in all the way, let’s look at the effect a CASCADE has on the data. We’ll 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
```

This yields us 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)

(1 row(s) affected)

OrderID	CustomerNo	OrderDate	EmployeeID
-----	-----	-----	-----

(0 row(s) affected)

OrderID	PartNo	Description	UnitPrice	Qty
-----	-----	-----	-----	-----

(0 row(s) affected)

Notice that, even though we issued a `DELETE` against the `Orders` table only, the `DELETE` also `CASCADED` to our matching records in the `OrderDetails` table. Records in both tables were deleted. If we had defined our 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 we had a `ShipmentDetails` table that referenced rows in `OrderDetails` with a `CASCADE` action, then those too would have been deleted just by our one `DELETE` in the `Orders` table.

This is actually one of the danger areas of cascading actions — it's very, very easy to not realize 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, then the child row will be set to either `NULL` or whatever the default value for that column is (whichever you chose — `SET NULL` or `SET DEFAULT`). It's just that simple.

Other Things to Think About with Foreign Keys

There are some other things to think about before we're done with foreign keys. We will be coming back to this subject over and over again throughout the book, but for now, I just want to get in a couple of finer points:

- ☐ What makes values in foreign keys required versus optional
- ☐ How foreign keys are bi-directional

What Makes Values in Foreign Keys Required vs. Optional

By the nature of a foreign key itself, you have two possible choices on what to fill into a column or columns that have a foreign key defined for them:

- ☐ Fill the column in with a value that matches the corresponding column in the referenced table.
- ☐ Do not 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 in the preceding list) by simply defining the referencing column as `NOT NULL`. Since 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.

How Foreign Keys Are Bi-Directional

We touched on this some when we discussed CASCADE actions, but when defining foreign keys, I can't stress enough that they effectively place restrictions on *both* tables. Up to this point, we've been talking about things in terms of the referencing table; however, once the foreign key is defined, the referenced table must also live by a rule:

By default, you cannot delete a record or update the referenced column in a referenced table if that record is referenced from the dependent table. If you want to be able to delete or update such a record, then you need to set up a CASCADE action for the delete and/or update.

Let's illustrate this "You can't delete or update a referenced record" idea.

We just defined a couple of foreign keys for the `Orders` table. One of those references the `EmployeeID` columns of the `Employees` table. Let's say, for instance, that we have an employee with an `EmployeeID` of 10 who takes many orders for us for a year or two, and then decides to quit and move on to another job. Our tendency would probably be to delete the record in the `Employees` table for that employee, but that would create a rather large problem — we would get what are called *orphaned* records in the `Orders` table. Our `Orders` table would have a large number of records that still have an `EmployeeID` of 10. If we are allowed to delete `EmployeeID` 10 from the `Employees` table, then we will no longer be able to tell which employee entered in all those orders — the value for the `EmployeeID` column of the `Orders` table will become worthless!

Now let's take this example one step further. Let's say now, that the employee did not quit. Instead, for some unknown reason, we wanted to change that employee's ID number. If we made the change (via an `UPDATE` statement) to the `Employees` table, but did not make the corresponding update to the `Orders` table, then we would again have orphaned records — we would have records with a value of 10 in the `EmployeeID` column of the `Orders` table with no matching employee.

Now, let's take it one more step further! Imagine that someone comes along and inserts a new record with an `EmployeeID` of 10 — we now have a number of records in our `Orders` table that will be related to an employee who didn't take those orders. We would have bad data (yuck!).

Instead of allowing orphaned records, SQL Server, by default, restricts us from deleting or updating records from the referenced table (in this case, the `Employees` table) unless any dependent records have already been deleted from or updated in the referencing (in this case, `Orders`) table.

This is actually not a bad segue into a brief further discussion of when a `CASCADE` action makes sense and when it doesn't. Data-integrity-wise, we probably wouldn't want to allow the deletion of an employee if there are dependent rows in the `Orders` table. Not being able to trace back to the employee would degrade the value of our data. On the other hand, it may be perfectly valid (for some very strange reason) to change an employee's ID. We could `CASCADE` that update to the `Orders` table with little ill effects. Another moral to the story here is not to think that you need the same `CASCADE` decision for both `UPDATE` and `DELETE` — think about each separately (and carefully).

As you can see, although the foreign key is defined on one table, it actually placed restrictions on both tables (if the foreign key is self-referenced, then both sets of restrictions are on the one table).

UNIQUE Constraints

These are relatively easy. `UNIQUE` constraints are essentially the younger sibling 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 are not considered to be *the* unique identifier of a record in that table (even though you could effectively use it that way) and that you *can* have more than one `UNIQUE` constraint (remember that you can only have one primary key per table).

Once you establish a `UNIQUE` constraint, every value in the named columns must be unique. If you go to update or insert a row with a value that already exists in a column with a unique constraint, SQL Server will raise an error and reject the record.

Unlike a primary key, a `UNIQUE` constraint does not automatically prevent you from having a `NULL` value. Whether `NULL`s are allowed or not depends on how you set the `NULL` option for that column in the table. Keep in mind, however, that, if you do allow `NULL`s, you will be able to insert only 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).

Since there is nothing novel about this (we've pretty much already seen it with primary keys), let's get right to the code. Let's create yet another table in our `Accounting` database — this time, it will be our `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. We will go ahead and create a `UNIQUE` constraint on our `Employees` table:

```
ALTER TABLE Employees
    ADD CONSTRAINT AK_EmployeeSSN
    UNIQUE (SSN)
```

A quick run of `sp_helpconstraint` verifies that our 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. Examples of the criteria for a `CHECK` constraint include:

Goal	SQL
Limit <code>Month</code> 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 <code>Shippers</code>	<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 really 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 as compared to the alternatives (rules and triggers).

Still building on our Accounting database, let's add a modification to our `Customers` table to check for a valid date in our `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 we 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).

We'll see the other type of default when we 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 either define it as a literal value (say, setting a default salary to zero or "UNKNOWN" for a string column) or as one of several system values such as `GETDATE()`.

The main things to understand about a `DEFAULT` constraint are that:

- ❑ Defaults are only used in `INSERT` statements — they are ignored for `UPDATE` and `DELETE` statements.
- ❑ If any value is supplied in the `INSERT`, then the default is not used.
- ❑ If no value is supplied, the default will always be used.

Defaults are only made use of in `INSERT` statements. I cannot express enough how much this is a confusion point for many SQL Server beginners. Think about it this way — when you are first inserting the record, SQL Server doesn't have any kind of value for your column except what you supplied (if anything) or the default. If neither of these is supplied, then SQL Server will either insert a `NULL` (essentially amounting to “I don't know”), or, if your column definition says `NOT NULL`, then SQL Server will reject the record. After that first insert, however, SQL Server already has some value for that column. If you are updating that column, then it has your new value. If the column in question isn't part of an `UPDATE` statement, then SQL Server just leaves what is already in the column.

If a value was provided for the column, then there is no reason to use the default — the supplied value is used.

If no value is supplied, then the default will always be used. Now this seems simple enough until you think about the circumstance where a `NULL` value is what you actually wanted to go into that column for a record. If you don't supply a value on a column that has a default defined, then the default will be used. What do you do if you really wanted it to be `NULL`? Say so — insert `NULL` as part of your `INSERT` statement.

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 we'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 we created earlier in the chapter. This time, we'll 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 have run your `CREATE` script, you can again make use of `sp_helpconstraint` to show you what you have done. You can then test out 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 since we didn't supply a value ourselves:

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. We make use of our `ALTER` statement and `ADD` the constraint as before, but we add a `FOR` operator to tell SQL Server what 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`, we are able to 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 will let you do this, and you wouldn't see the issues with it until run time.

Disabling Constraints

Sometimes we 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 us some way of deleting constraints, but SQL Server also allows us to just 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 the circumstances in which you create the constraint at the same time as you create the table. Quite often, however, data rules are established after the fact. Let's 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, let's look at an example.

The `Customers` table we created in the `Accounting` database has a field called `Phone`. The `Phone` field was created with a datatype of `char` because we expected all of the phone numbers to be of the same length. We also set it with a length of 15 in order to ensure that we have enough room for all the formatting characters. However, we have not done anything to make sure that the records inserted into the database do indeed match the formatting criteria that we expect. To test this out, we'll insert a record in a format that is not what we'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 let's add a constraint to control the formatting of the Phone field:

```
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 we run this, we 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 will not create the constraint unless the existing data meets the constraint criteria. To get around this long enough to install the constraint, either we need to correct the existing data or we must make use of the `WITH NOCHECK` option in our `ALTER` statement. To do this, we 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 we run our 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 we modify our `INSERT` statement to adhere to our 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 ())
```

Try running a `SELECT` on the `Customers` table at this point. You'll see data that both does and does not adhere to our `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

All right — so you understand why we need to be able to add new constraints that do not check old data, but why would we want to temporarily disable an existing constraint? Why would we want to let data that we know is bad be added to the database? Actually, the most common reason is basically the same reason for which we make use of the `WITH NOCHECK` option — old data.

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 up 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 back using a `WITH NOCHECK`. But what a pain! Fortunately, we don't need to do that. Instead, we 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 we just added in the last section:

```
ALTER TABLE Customers
NOCHECK
CONSTRAINT CN_CustomerPhoneNo
```

Now we can run that `INSERT` statement again — the one we proved 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, we are able to `INSERT` non-conforming data to the table.

By now, the question may have entered your mind asking how do you know whether you have the constraint turned on or not. It would be pretty tedious if you had to create a bogus record to try to insert in order to test whether your constraint is active or not. Like most (but not all) of these kinds of dilemmas, SQL Server provided a procedure to indicate the status of a constraint, and it's a procedure we've already seen, `sp_helpconstraint`. To execute it against our `Customers` table is easy:

```
EXEC sp_helpconstraint Customers
```

The results are a little too verbose to fit into the pages of this book, but the second result set this procedure generates includes a column called `status_enabled`. Whatever this column says the status is can be believed — in this case, it should currently be `Disabled`.

When we are ready for the constraint to be active again, we simply turn it back 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.

```

Our other option, of course, is to run `sp_helpconstraint` again, and check out the `status_enabled` column. If it shows as `Enabled`, then our 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 stand-by, 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 are not ANSI-compliant (which creates portability issues), and they do not perform as well as constraints do. Microsoft has listed Rules and Defaults as there only for backward compatibility since version 7.0—that's three versions and 6 years—not an encouraging thing if you're asking yourself whether this feature is going to continue to be supported in the future. I wouldn't go so far as to suggest that you start sifting through and replacing any old code that you may come across, but you should use constraints for any new code you generate.

The primary thing that sets rules and defaults apart from constraints is in their very nature; constraints are features of a table—they have no existence on their own—while rules and defaults are actual objects in and of themselves. 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.

The independent object nature of rules and defaults gives them the ability to be reused without being redefined. Indeed, rules and defaults are not limited to being bound to just tables; they can also be bound to datatypes—vastly improving your ability to make highly functional user-defined datatypes.

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 will not 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), whereas `LIKE ([0-9][0-9][0-9])` would (it applies only to whatever column the rule is bound to).

Let's define a rule so that you can see the differences first hand:

```
CREATE RULE SalaryRule
AS @Salary > 0
```

Notice that what we are comparing is shown as a variable—whatever the value is of the column being checked, that is the value that will be used in the place of `@Salary`. Thus, in this example, we're saying that any column our rule is bound to would have to 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
```

And it will show you your exact rule definition:

```
Text
-----
CREATE RULE SalaryRule
    AS @Salary > 0
```

Now we've got a rule, but it isn't doing anything. If we tried to insert a record in our `Employees` table, we could still insert any value right now without any restrictions beyond datatype.

In order to activate the rule, we need to make use of a special stored procedure called `sp_bindrule`. We want to bind our `SalaryRule` to the `Salary` column of our `Employees` table. The syntax looks like this:

```
sp_bindrule <'rule'>, <'object_name'>, [<'futureonly_flag'>]
```

The `rule` part is simple enough—that's the rule we want to bind. The `object_name` is also simple enough—it's the object (column or user-defined datatype) to which we 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 datatype. The default is for this to be off. However, if you set it to `True` or pass in a `1`, then the binding of the rule will only apply to new columns to which you bind the user-defined datatype. Any columns that already have the datatype in its old form will continue to use that form.

Since we're just binding this rule to a column, our syntax requires only the first two parameters:

```
sp_bindrule 'SalaryRule', 'Employees.Salary'
```

Take a close look at the `object_name` parameter—we have both `Employees` and `Salary` separated by a `"."`—why is that? Since 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 do not use the `tablename.column` naming structure, then SQL Server will assume that what you're naming must be a user-defined datatype—if it doesn't find one, you'll get back an error message that can be a bit confusing if you hadn't intended to bind the rule to a datatype:

```
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 our case, trying to insert or update an `Employees` record with a negative value violates the rule and generates an error.

If we want to remove our rule from use with this column, we make use of `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 is used against a user-defined datatype (rather than a specific column), then the unbinding will only apply to future uses of that datatype—existing columns using that datatype will still make use of the rule.

Dropping Rules

If you want to completely eliminate a rule from your database, you use the same `DROP` syntax that we've already become 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 in the way that they are attached to a table and the default's (the object, not the constraint) support for a user-defined datatype.

The concept of defaults vs. `DEFAULT` constraints is wildly difficult for a lot of people to grasp. After all, they have almost the same name. If we refer to “default”, then we are referring to either the object-based default (what we’re talking about in this section), or as shorthand to the actual default value (that will be supplied if we don’t provide an explicit value). If we refer to a “`DEFAULT` constraint,” then we are talking about the non-object-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, to define a default of zero for our `Salary`:

```
CREATE DEFAULT SalaryDefault  
AS 0
```

Again, a default is worthless without being bound to something. To bind it we make use of `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, we 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, you use the same `DROP` syntax that we've already become familiar with for tables and rules:

```
DROP DEFAULT <default name>
```

Determining Which Tables and Datatypes 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 datatypes are making use of 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 something called "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 will now allow 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 will not have any dependency listing if you use `sp_depends`.

Triggers for Data Integrity

We've got a whole chapter coming up on triggers, but 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. Since we have that chapter coming up, I'm not going to get into it very deeply here other than to say that triggers have a very large number of things they can do 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 relatively last resort.

Choosing What to Use

Wow. Here you are with all these choices, and now 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 rather confusing when making a decision on what to use. You've got some hints from me as we've been 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 we look at them all together for a bit.

Restriction	Pros	Cons
Constraints	Fast. Can reference other columns. Happens before the command occurs. ANSI-compliant.	Must be redefined for each table. Can't reference other tables. Can't be bound to datatypes.
Rules, Defaults	Independent objects. Reusable. Can be bound to datatypes. Happens before the command occurs.	Slightly slower. Can't reference across columns. Can't reference other tables. Really meant for backward compatibility only!!!
Triggers	Ultimate flexibility. Can reference other columns and other tables. Can even use .NET to reference information that is external to your SQL Server.	Happens after the command occurs. High overhead.

The main time to use rules and defaults is if you are implementing a rather robust logical model, and are making extensive use of user-defined datatypes. 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 only be used 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). We will cover these in some detail later in the book.

That leaves us with 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 over and over again if you have a common constraint logic.

Regardless of what kind of integrity mechanism you're putting in place (keys, triggers, constraints, 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 to it before you stick it in your database. 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 what constraints you have before you put the next one on, 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, we will look at the tie between some forms of constraints and indexes, along with taking a look at the advanced data integrity rules that can be implemented using triggers. We'll also begin looking at how the choices between these different mechanisms affect our design decisions.

7

Adding More to Our Queries

When I first started writing about SQL Server a number of years ago, I was faced with a question of when exactly to introduce more complex queries into the knowledge mix — this book faces that question all over again. At issue is something of a “chicken or egg” thing — talk about scripting, variables, and the like first, or get to some things that a relatively beginning user might make use of long before they do server side scripting. This time around, the notion of “more queries early” won out.

Some of the concepts in the chapter are going to challenge you to a new way of thinking. You already had a taste of this just dealing with joins, but you haven’t had to deal with the kind of depth that I want to challenge you with 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 (sub-procedures — logical steps) as opposed to as a whole (the “set,” or SQL way).

While SQL Server 2005 supports procedural language concepts now more than ever, 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 combine it into the whole again to the largest extent that makes sense.

This is really what’s at the heart of my challenge of a “new way of thinking” — conceptualizing the question as a whole rather than in steps. When we program in most languages, we usually work in a linear fashion. With SQL however, you need to think more in terms of set theory. You can liken this to math class, and the notion of $A \cup B$ or $A \cap B$. We need to think less in terms of steps to resolve the data, and more about how the data “fits” together.

In this chapter, we’re going to be using this concept of data fit 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. In addition, we’ll also be taking a look at query performance, and what you can do to get the most out of queries.

Among the topics we’ll be covering in this chapter are:

- ❑ Nested subqueries
- ❑ “Correlated” subqueries
- ❑ Derived Tables
- ❑ Making use of the `EXISTS` operator
- ❑ Optimizing query performance

We’ll see how using subqueries, we can make the seemingly impossible completely possible, and how an odd tweak here and there can make a big difference in your query performance.

What Is a Subquery?

A *subquery* is a normal T-SQL query that is nested inside another query. They are created using parentheses 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 a couple of 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 debate (perhaps 20 or 30 e-mails flying back and forth with examples, reasons, etc. over a few days) with a coworker over the joins vs. 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 we 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 we know what a subquery theoretically is, let’s 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 full 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:

```
SELECT DISTINCT o.OrderDate, od.ProductID
FROM Orders o
JOIN [Order Details] od
    ON o.OrderID = od.OrderID
WHERE OrderDate = '7/4/1996' --This is first OrderDate in the system
```

This yields us the correct results:

OrderDate	ProductID
1996-07-04 00:00:00.000	11
1996-07-04 00:00:00.000	42
1996-07-04 00:00:00.000	72

(3 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.

Chapter 7

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 — or can we? Actually, the answer is “Yes, we can.” By putting it all into just one statement:

```
SELECT DISTINCT o.OrderDate, od.ProductID
FROM Orders o
JOIN [Order Details] od
    ON o.OrderID = od.OrderID
WHERE o.OrderDate = (SELECT MIN(OrderDate) FROM Orders)
```

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 you 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 criteria for a query.

For this one, let's switch over to using the `Pubs` database as we did in Chapter 5. What we want is a list of all the stores that have discount records. The stores are, not surprisingly, in a table called `Stores`. The discounts are in a table called, appropriately enough, `discounts`.

We might write something like this:

```
USE PUBS

SELECT stor_id AS "Store ID", stor_name AS "Store Name"
FROM Stores
WHERE stor_id IN (SELECT stor_id FROM Discounts)
```

As it happens, this gets us back only one row — but what's interesting is that it is exactly the same row we saw doing an inner join in a query in Chapter 5:

```
Store ID      Store Name
-----
8042          Bookbeat

(1 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:

```
SELECT s.stor_id AS "Store ID", stor_name AS "Store Name"
FROM Stores s
JOIN Discounts d
    ON s.stor_id = d.stor_id
```

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's done.

*SQL Server is actually pretty smart about this kind of thing. In the lion's share of situations, SQL Server will actually resolve the nested subquery solution to the same query plan it would use on the join. So, with that in mind, the truth is that most of the time, there really isn't that much difference. The problem, of course, is that I just said **most** of the time. When the query plans vary, the join is usually the better choice, and thus the recommendation to use that syntax by default.*

Using a Nested `SELECT` to Find Orphaned Records

This type of nested `SELECT` is nearly identical to our 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.

Before we do the nested `SELECT` syntax, let's review one of our examples of an outer join from Chapter 5. In this query, we were trying to identify all the stores in the `pubs` database that didn't have matching discount records:

```
USE Pubs

SELECT s.Stor_Name AS "Store Name"
FROM Discounts d
RIGHT OUTER JOIN Stores s
            ON d.Stor_ID = s.Stor_ID
WHERE d.Stor_ID IS NULL
```

This got us five rows back:

```
Store Name
-----
Eric the Read Books
Barnum's
News & Brews
Doc-U-Mat: Quality Laundry and Books
Fricative Bookshop

(5 row(s) affected)
```

This is the way that, typically speaking, things should be done. I can't say, however, that it's the way that things are usually done. The join usually takes a bit more thought, so we usually wind up with the nested `SELECT` instead.

See if you can write this nested `SELECT` on your own — but I'll warn you: this one has something of a gotcha in it. Once you're done, come back and take a look below.

It should wind up looking like this:

```
SELECT stor_id AS "Store ID", stor_name AS "Store Name"
FROM Stores
WHERE stor_id NOT IN
    (SELECT stor_id FROM Discounts WHERE stor_id IS NOT NULL)
```


This yields us exactly the same five records. I'm guessing, however, that you probably didn't use the `NOT NULL` comparison in the inner query the first time you tried it.

Whether you need to include the `NOT NULL` qualification or not depends on whether your table accepts `NULL`s and what exactly you want for results. In our case, if we leave the comparison off, we will, in error, wind up thinking that there aren't any stores that don't have discounts (when there really are). The reason has to do with how `NULL`s compare—you need to be extremely careful when dealing with the possibility of `NULL` values in your `IN` list.

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 processed only 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:

- ❑ The outer query obtains a record, and passes it into the inner query.
- ❑ The inner query executes based on the passed in value(s).
- ❑ 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 let's look at it in an example.

We'll go back to the `Northwind` database and 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 `Orders` table, so we know that our 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 only looking 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.

A temporary table is pretty much just what it sounds like—a table that is created for temporary use and will go away after our processing is complete—exactly how long it will stay around is variable and is outside the scope of this chapter. We will, however, visit temporary tables a bit more as we continue through the book.

The temporary table solution might look something like this:

```
USE Northwind

-- Get a list of customers and the date of their first order
SELECT CustomerID, MIN((OrderDate)) AS OrderDate
INTO #MinOrderDates
FROM Orders
GROUP BY CustomerID
ORDER BY CustomerID

-- Do something additional with that information
SELECT o.CustomerID, o.OrderID, o.OrderDate
FROM Orders o
JOIN #MinOrderDates t
    ON o.CustomerID = t.CustomerID
    AND o.OrderDate = t.OrderDate
ORDER BY o.CustomerID

DROP TABLE #MinOrderDates
```

We get back 89 rows:

```
(89 row(s) affected)
```

CustomerID	OrderID	OrderDate
-----	-----	-----
ALFKI	10643	1997-08-25 00:00:00.000
ANATR	10308	1996-09-18 00:00:00.000
ANTON	10365	1996-11-27 00:00:00.000
AROUT	10355	1996-11-15 00:00:00.000

```
BERGS                10278                1996-08-12 00:00:00.000
...
...
...
WHITC                10269                1996-07-31 00:00:00.000
WILMK                10615                1997-07-30 00:00:00.000
WOLZA                10374                1996-12-05 00:00:00.000
```

(89 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 Northwind 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 we explore our 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.

OK, 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.OrderID, o1.OrderDate
FROM Orders o1
WHERE o1.OrderDate = (SELECT Min(o2.OrderDate)
                     FROM Orders o2
                     WHERE o2.CustomerID = o1.CustomerID)

ORDER BY CustomerID
```

With this, we get back the same 89 rows:

```
CustomerID      OrderID      OrderDate
-----
ALFKI           10643        1997-08-25 00:00:00.000
ANATR           10308        1996-09-18 00:00:00.000
ANTON           10365        1996-11-27 00:00:00.000
AROUT           10355        1996-11-15 00:00:00.000
BERGS           10278        1996-08-12 00:00:00.000
...
...
...
WHITC           10269        1996-07-31 00:00:00.000
WILMK           10615        1997-07-30 00:00:00.000
WOLZA           10374        1996-12-05 00:00:00.000
```

(89 row(s) affected)

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—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 concerning 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 we 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 89 `row(s)` affected only once. That’s because it affected 89 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 only references the inner query 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 often 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

Chapter 7

addition, we no longer need to build any kind of condition in—we’re asking for all customers (no restrictions), and we just want to know when their first order date was.

The query actually winds up being a bit simpler than the last one, and it looks like this:

```
SELECT cu.CompanyName,
       (SELECT Min(OrderDate)
        FROM Orders o
        WHERE o.CustomerID = cu.CustomerID)
       AS "Order Date"
FROM Customers cu
```

This gets us data that looks something like this:

CompanyName	Order Date
Alfreds Futterkiste	1997-08-25 00:00:00.000
Ana Trujillo Emparedados y helados	1996-09-18 00:00:00.000
Antonio Moreno Taquería	1996-11-27 00:00:00.000
Around the Horn	1996-11-15 00:00:00.000
Berglunds snabbköp	1996-08-12 00:00:00.000
Blauer See Delikatessen	1997-04-09 00:00:00.000
...	
...	
...	
White Clover Markets	1996-07-31 00:00:00.000
Wilman Kala	1997-07-30 00:00:00.000
Wolski Zajazd	1996-12-05 00:00:00.000

(91 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 `Order Date` column. Why do you suppose that is? The cause is, of course, because there is no record in the `Orders` 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 us at this point is `ISNULL()`. `ISNULL()` accepts a variable (which we’ll talk about in Chapter 11) 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
ISNULL(NULL, 5)	5
ISNULL(5, 15)	5
ISNULL(MyColumnName, 0) where MyColumnName IS NULL	0
ISNULL(MyColumnName, 0) where MyColumnName = 3	3
ISNULL(MyColumnName, 0) where MyColumnName = 'Fred Farmer'	Fred Farmer

Now let’s see this at work in our query:

```
SELECT cu.CompanyName,
       ISNULL(CAST ((SELECT MIN(o.OrderDate)
                     FROM Orders o
                     WHERE o.CustomerID = cu.CustomerID)AS varchar), ' NEVER ORDERED')
       AS "Order Date"
FROM Customers cu
```

Now, in our two lines that we had problems with, we go from:

```
...
FISSA Fabrica Inter. Salchichas S.A.      NULL
...
Paris spécialités                          NULL
...
```

to something substantially more useful:

```
...
FISSA Fabrica Inter. Salchichas S.A.      NEVER ORDERED
...
Paris spécialités                          NEVER ORDERED
...
```

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. Because the first row starts off returning a valid date, the column Order Date 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 further later in the chapter.

So, at this point, we’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 into 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

Chapter 7

we’ve discussed up to this point. An example would be where, for each row in a given table, you may have multiple results in the subquery, but you’re looking for an action more complex than our `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 we turn to a somewhat lesser known construct in SQL — a *derived table*. A derived table is made up of the columns and rows of a result set from a query. (Heck, they have columns, rows, datatypes, and so on 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, Chocolate. No problem! Your query might look something like this:

```
SELECT c.CompanyName
FROM Customers AS c
JOIN Orders AS o
    ON c.CustomerID = o.CustomerID
JOIN [Order Details] AS od
    ON o.OrderID = od.OrderID
JOIN Products AS p
    ON od.ProductID = p.ProductID
WHERE p.ProductName = 'Chocolate'
```

OK, so that was easy. Now I’m going to throw you a twist — let’s now say I want to know all the customers that ordered not only Chocolate, but also Vegie-spread. 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.ProductName = 'Chocolate' AND p.ProductName = 'Vegie-spread'
```

But that’s not going to work at all — each row is for a single product, so how can it have both Chocolate and Vegie-spread 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 Chocolate with the results of a query to find buyers of Vegie-spread. How do we join results, however? Well, as you might expect given the title of this section, through the use of derived tables.

To create our derived table, we need two things:

- ❑ To enclose our query that generates the result set in parentheses
- ❑ To alias the results of the query

So, the syntax looks something like this:

```
SELECT <select list>
FROM (<query that returns a regular resultset>) 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 Chocolate and Vegie-spread. So our query should look something like this:

```

SELECT DISTINCT c.CompanyName
FROM Customers AS c
JOIN
    (SELECT CustomerID
     FROM Orders o
     JOIN [Order Details] od
         ON o.OrderID = od.OrderID
     JOIN Products p
         ON od.ProductID = p.ProductID
     WHERE p.ProductName = 'Chocolate') AS spen
    ON c.CustomerID = spen.CustomerID
JOIN
    (SELECT CustomerID
     FROM Orders o
     JOIN [Order Details] od
         ON o.OrderID = od.OrderID
     JOIN Products p
         ON od.ProductID = p.ProductID
     WHERE ProductName = 'Vegie-spread') AS spap
    ON c.CustomerID = spap.CustomerID

```

As it happens, we get only one customer:

```

CompanyName
-----
Ernst Handel

(1 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—for example, Ernst Handel has ordered Vegie-spread twice, so I would have gotten one record for each. I only asked which customers had ordered both, not how many had they ordered.

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 an operator much like the IN keyword is, but it also looks at things just a bit differently.

Chapter 7

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. What we're going to query here is a list of customers that have placed at least one order (we don't care how many):

```
SELECT CustomerID, CompanyName
FROM Customers cu
WHERE EXISTS
    (SELECT OrderID
     FROM Orders o
     WHERE o.CustomerID = cu.CustomerID)
```

This gets us what amounts to the same 89 records that we've been dealing with throughout this chapter:

CustomerID	CompanyName
-----	-----
ALFKI	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y helados
ANTON	Antonio Moreno Taquería
AROUT	Around the Horn
BERGS	Berglunds snabbköp
BLAUS	Blauer See Delikatessen
...	
...	
...	
WHITC	White Clover Markets
WILMK	Wilman Kala
WOLZA	Wolski Zajazd

(89 row(s) affected)

We could have easily done this same thing with a join:

```
SELECT DISTINCT cu.CustomerID, cu.CompanyName
FROM Customers cu
JOIN Orders o
    ON cu.CustomerID = o.CustomerID
```

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.

Let's take a brief look at things the other way around — that is, what if our query wanted the customers who had not ordered anything? Under the join method that we looked at back in Chapter 5, we would have had to make some significant changes in the way we went about getting our answers. First, we

would have to use an outer join. Then we would perform a comparison to see whether any of the `Order` records were `NULL`.

It looked like this:

```
USE Northwind

SELECT c.CustomerID, CompanyName
FROM Customers c
LEFT OUTER JOIN Orders o
    ON c.CustomerID = o.CustomerID
WHERE o.CustomerID IS NULL
```

And it returned two rows.

To do the same change over when we're using `EXISTS` we add only one word — `NOT`:

```
SELECT CustomerID, CompanyName
FROM Customers cu
WHERE NOT EXISTS
    (SELECT OrderID
     FROM Orders o
     WHERE o.CustomerID = cu.CustomerID)
```

And we get back those exact same two rows:

CustomerID	CompanyName
FISSA	FISSA Fabrica Inter. Salchichas S.A.
PARIS	Paris spécialités

(2 row(s) affected)

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

One common use of `EXISTS` is to check for the existence of a table before running a create statement. You may want to drop an existing table, or you just may want to change to an `ALTER` statement or some other statement that adjusts the existing table if there is one. One of the most common ways you'll see this done will look something like this:

```
IF EXISTS (SELECT * FROM sysobjects WHERE id = object_id(N'[dbo].[Shippers]')
AND OBJECTPROPERTY(id, N'IsUserTable') = 1)
DROP TABLE [dbo].[Shippers]
GO

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

Chapter 7

```
[ShipperID] [int] IDENTITY (1, 1) NOT NULL ,
[CompanyName] [nvarchar] (40) NOT NULL ,
[Phone] [nvarchar] (24) NULL
)
GO
```

Since the `EXISTS` returns nothing but `TRUE` or `FALSE`, that means it works as an excellent conditional expression. The preceding example will run the `DROP TABLE` code only if the table exists; otherwise, it skips over that part and moves right into the `CREATE` statement. This avoids one of two errors showing up when you run the script. First, that it can't run the `CREATE` statement (which would probably create other problems if you were running this in a script where other tables were depending on this being done first) because the object already exists. Second, that it couldn't `DROP` the table (this pretty much just creates a message that might be confusing to a customer who installs your product) because it didn't exist. You're covered for both.

As for an instance of this, let's write our own `CREATE` script for something that's often skipped in the automation effort — the database. But creation of the database is often left as part of some cryptic directions that say something like "create a database called 'xxxx'". The fun part is when the people who are actually installing it (who often don't know what they're doing) start including the quotes, or create the database too small, or a host of other possible and very simple errors to make. This is the point where I hope you have a good tech support department.

Instead, we'll just build a little script to create the database object that could go with `Northwind`. For safety's sake, we'll call it `NorthwindCreate`. We'll also keep the statement to a minimum because we're interested in the `EXISTS` rather than the `CREATE` command:

```
USE master

GO

IF NOT EXISTS (SELECT 'True' FROM sys.databases WHERE name = 'NorthwindCreate')
BEGIN
    CREATE DATABASE NorthwindCreate
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 `NorthwindCreate` (unless by sheer coincidence that you created something called that before we got to this point), so you'll get a response back that looks like this:

```
Command(s) completed successfully.
```

This was high unhelpful in terms of telling you what exactly was done, but at least you know it thinks it did what you asked.

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 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. For example, the query we used as a derived table example can also be written with a couple of `EXISTS` operators (one for each product), but the derived table happens to run more than twice as fast. That's definitely the exception not the rule—`EXISTS` will normally smoke a derived table for performance—just remember that rules are sometimes made to be broken.

Mixing Datatypes: `CAST` and `CONVERT`

You'll see both `CAST` and `CONVERT` used frequently. Indeed, we've touched briefly on both of these already in this chapter. Considering how often we'll use these two functions, this seems like a good time to look a little closer at what they can do for you.

Both `CAST` and `CONVERT` perform datatype 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.

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 for 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 datatype 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:

```
SELECT 'The Customer has an Order numbered ' + OrderID
FROM Orders
WHERE CustomerID = 'ALFKI'
```

Chapter 7

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(OrderID AS varchar)
FROM Orders
WHERE CustomerID = 'ALFKI'
```

And you get a much different result:

```
-----
The Customer has an Order numbered 10643
The Customer has an Order numbered 10692
The Customer has an Order numbered 10702
The Customer has an Order numbered 10835
The Customer has an Order numbered 10952
The Customer has an Order numbered 11011

(6 row(s) affected)
```

The conversions can actually get a little less intuitive also. Take for example, that you wanted to convert a timestamp column into a regular number. A timestamp is just a binary number, so the conversion isn't any really big deal:

```
CREATE TABLE ConvertTest
(
    ColID    int    IDENTITY,
    ColTS    timestamp
)

INSERT INTO ConvertTest
    DEFAULT VALUES

SELECT ColTS AS "Uncovered", CAST(ColTS AS int) AS "Converted" FROM ConvertTest
```

Yields us something like (your exact numbers will vary):

```
(1 row(s) affected)

Uncovered          Converted
-----
0x000000000000000C9    201

(1 row(s) affected)
```

We can also convert dates:

```
SELECT OrderDate, CAST(OrderDate AS varchar) AS "Converted"
FROM Orders
WHERE OrderID = 11050
```

This yields us something similar to (your exact format may change depending on system date configuration):

OrderDate	Converted
-----	-----
1998-04-27 00:00:00.000	Apr 27 1998 12:00AM

(1 row(s) affected)

Notice that 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 Orders
WHERE OrderID = 11050
```

This yields us:

OrderDate	Converted
-----	-----
1998-04-27 00:00:00.000	1998/04/27

(1 row(s) affected)

Which is quite a bit different from what CAST did. Indeed, we could have converted to any one of 34 two-digit or four-digit year formats.

```
SELECT OrderDate, CONVERT(varchar(12), OrderDate, 5) AS "Converted"
FROM Orders
WHERE OrderID = 11050
```

This gives us:

OrderDate	Converted
-----	-----
1998-04-27 00:00:00.000	27-04-98

(1 row(s) affected)

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; and 5 the Italian standard, with a two-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 some changes have been made to things to deal with the infamous Y2K issue. One of the changes is that you can set a `split` point that SQL Server will use to determine whether a two-digit year should 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 in the database server configuration.

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 and 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 we’re 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 gauge 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. ?

This 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 using 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.	Pre-query. 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.
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.

Situation	Favors
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 having to join the entire table.
The match, after considering all criteria, is going to return 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.
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 is 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 to them, 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 that, 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 Chapters 3 and 4 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 bit. You need something more, and, hopefully the options covered in this chapter have given you a little extra arsenal to deal with those tough situations.

Exercises

1. Write a query that returns the start date of all Northwind employees in MM/DD/YY format.
2. Write separate queries using a JOIN, a subquery, and then an EXISTS to list all Northwind customers who have not placed an order.
3. Show the most recent five orders that were purchased from a customer who has spent more than \$25,000 with Northwind.

8

Being Normal: Normalization and Other Basic Design Issues

I can imagine you as being somewhat perplexed about the how and why of some of the tables we've constructed thus far. With the exception of a chapter or two, this book has tended to have 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. As such, the table examples will typically have a database layout that is, for the most part, *normalized* to what is called the third normal form.

So what is “normal form”? We'll be taking a very solid look at that in this chapter, but, for the moment, let's just say that it means your data has been broken out into a logical, non-repetitive 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 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.

This is probably going to be one of the toughest chapters in the book to grasp because of a paradox in what to learn first. Some of the concepts used in this chapter refer to things we'll be covering later—such as triggers and stored procedures. On the other hand, it is difficult to relate those topics without understanding their role in database design.

I strongly recommend reading this chapter through, and then coming back to it again after you've read several of the subsequent chapters.

Tables

This is going to seem beyond basic, but let's make a brief review of what exactly a table is. We're obviously not talking about the kind that sits in your kitchen, but, rather, the central object of any database.

A table is a collection of instances of data that have the same general *attributes*. These instances of data are organized into *rows* and *columns* of data. A table should represent a “real-world” collection of data (often referred to as an *entity*), and will have *relationships* with information in other tables. A drawing of the various entities (tables) and relationships (how they work together) is usually referred to as an *Entity-Relationship diagram* — or *ER Diagram*. Sometimes the term “ER Diagram” will even be shortened further down to *ERD*.

By connecting two or more tables through their various relationships, you are able to temporarily create other tables as needed from the combination of the data in both tables (we've already seen this to some degree in Chapters 4 and 5). A collection of related entities are then grouped together into a database.

Keeping Your Data “Normal”

Normalization is something of the cornerstone model of modern OLTP database design. Normalization first originated along with the concept of relational databases. Both came from the work of E. F. Codd (IBM) in 1969. Codd put forth the notion that a database “consists of a series of unordered tables that can be manipulated using non-procedural operations that return tables.”

Several things are key about this:

- ❑ Order must be unimportant.
- ❑ The tables would be able to “relate” to each other in a non-procedural way (indeed, Codd called tables, “relations”).
- ❑ That, by relating these base tables, you would be able to create a virtual table to meet a new need.

Normalization was a natural offshoot of the design of a database of “relations.”

The concept of normalization has to be one of most over-referenced and 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. Now, excuse me while I put that soapbox away, and we'll get on to talking about the normal forms and what they purportedly do for us.

Let's start off by saying that there are six normal forms. For those of you who have dealt with databases and normalization some before, that number may come as a surprise. You are very likely to hear that a fully normalized database is one that is normalized to the third normal form — doesn't it then follow that there must be only three normal forms? Perhaps it will make those same people who thought there were only three normal forms feel better that in this book we're only going to be looking to any extent at the three forms you've heard about, as they are the only three that are put to any regular use in the real world. I will, however, take a brief (very brief) skim over the other three forms just for posterity.

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. 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.

Let's take a look at the various normal forms and what each does for us.

Before the Beginning

You actually need to begin by getting a few things in place even before you try to get your data into first normal form. You have to have a thing or two in place before you can even consider the table to be a true entity in the relational database sense of the word:

- ☐ 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.

The place to start, then, is by identifying the right entities to have. Some of these will be fairly obvious, others will not. Many of them will be exposed and refined as you go through the normalization process. At the very least, go through and identify all the obvious entities.

If you're familiar with object-oriented programming, then you can liken the most logical top-level entities to objects in an object model.

Let's think about a hyper simple model — our sales model again. To begin with, we're not going to worry about the different variations possible, or even what columns we're going to have — instead, we're just going to worry about identifying the basic entities of our system.

First, think about the most basic process. What we want to do is create an entity for each atomic unit that we want to be able to maintain data on in the process. Our process then, looks like this: a customer calls or comes in and talks to an employee who takes an order.

Chapter 8

A first pass on this might have one entity: Orders.

As you become more experienced at normalization, your first pass at something like this is probably going to yield you quite a few more entities right from the beginning. For now though, we'll just take this one and see how the normalization process shows us the others that we need.

Assuming you've got your concepts down of what you want your entities to be, the next place to go is to figure out your beginning columns and, from there, a primary key. Remember that a primary key provides a unique identifier for each row.

We can peruse our list of columns and come up with *key candidates*. Your list of key candidates should include any column that can potentially be used to uniquely identify each row in your entity. There is, otherwise, no hard and fast rule on what column has to be the primary key (this is one of many reasons you'll see such wide variation in how people design databases that are meant to contain the same basic information). In some cases, you will not be able to find even one candidate key, and you will need to make one up (remember `Identity` and `rowguid()` columns?).

We've already created an Orders table in the last chapter, but for example purposes let's take a look at a very common implementation of an Orders table in the old flat file design:

Orders
OrderNo
CustomerNo
CustomerName
CustomerAddress
CustomerCity
CustomerState
CustomerZip
OrderDate
ItemsOrdered
Total

Since this is an Orders table, and logically, an order number is meant to be one to an order, I'm going go with `OrderNo` as my primary key.

OK, so now we have a basic entity. Nothing about this entity cares about the ordering of columns (tables are, by convention, usually organized as having the primary key as the first column(s), but, technically speaking, it doesn't have to be that way). Nothing in the basic makeup of this table cares about the ordering of the rows. The table, at least superficially, describes just one entity. In short, we're ready to begin our normalization process (actually, we sort of already have).

The First Normal Form

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 we already have), then moving any repeating data groups into new tables, creating new keys for those tables, and so on. In addition, we break out any columns that combine data into separate rows for each piece of data.

In the more traditional flat file designs, repeating data was commonplace — as was multiple pieces of information in a column — this was rather problematic in a number of ways:

- ❑ At that time, disk storage was extremely expensive. Storing data multiple times means wasted space. Data storage has become substantially less expensive, so this isn't as big an issue as it once was.
- ❑ Repetitive data means more data to be moved, and larger I/O counts. This means that performance is hindered as large blocks of data must be moved through the data bus and or network. This, even with today's much faster technology, can have a substantial negative impact on performance.
- ❑ The data between rows of what should have been repeating data often did not agree, creating something of a data paradox and a general lack of data integrity.
- ❑ If you wanted to query information out of a column that has combined data, then you had to first come up with a way to parse the data in that column (this was extremely slow).

Now, there are a lot of columns in our table, and I probably could have easily tossed in a few more. Still, the nice thing about it is that I could query everything out of one place when I wanted to know about orders.

Just to explore what this means, however, let's take a look at what some data in this table might look like. Note that I'm going to cut out a few columns here just to help things fit on a page, but I think you'll still be able to see the point:

Order No	Order Date	Customer No	Customer Name	CustomerF Address	Items Ordered
100	1/1/99	54545	ACME Co	1234 1st St.	1A4536, Flange, 7lbs, \$75; 4-OR2400, Injector, .5lbs, \$108; 4-OR2403, Injector, .5lbs, \$116; 1-4I5436, Head, 63lbs, \$750
101	1/1/99	12000	Sneed Corp.	555 Main Ave.	1-3X9567, Pump, 5lbs, \$62.50
102	1/1/99	66651	ZZZ & Co.	4242 SW 2nd	7-8G9200; Fan, 3lbs, \$84; 1-8G5437, Fan, 3lbs, \$15; 1-3H6250, Control, 5lbs, \$32
103	1/2/99	54545	ACME Co	1234 1st St.	40-8G9200, Fan, 3lbs, \$480; 1-2P5523, Housing, 1lbs, \$165; 1-3X9567, Pump, 5lbs, \$42

Chapter 8

We have a number of issues to deal with in this table if we're going to normalize it. While we have a functional primary key (yes, these existed long before relational systems), we have problems with both of the main areas of the first normal form.

- ❑ I have repeating groups of data (customer information). I need to break that out into a different table.
- ❑ The ItemsOrdered column does not contain data that is atomic in nature.

We can start by moving several columns out of the table:

OrderNo (PK)	OrderDate	CustomerNo	ItemsOrdered
100	1/1/1999	54545	1A4536, Flange, 7lbs, \$75; 4-OR2400, Injector, .5lbs, \$108; 4-OR2403, Injector, .5lbs, \$116; 1-4I5436, Head, 63lbs, \$750
101	1/1/1999	12000	1-3X9567, Pump, 5lbs, \$62.50
102	1/1/1999	66651	7-8G9200, Fan, 3lbs, \$84; 1-8G5437, Fan, 3lbs, \$15; 1-3H6250, Control, 5lbs, \$32
103	1/2/1999	54545	40-8G9200, Fan, 3lbs, \$480; 1-2P5523, Housing, 1lbs, \$165; 1-3X9567, Pump, 5lbs, \$42

And putting them into their own table:

CustomerNo (PK)	CustomerName	CustomerAddress
54545	ACME Co	1234 1st St.
12000	Sneed Corp.	555 Main Ave.
66651	ZZZ & Co.	4242 SW 2nd

There are several things to notice about both the old and new tables:

- ❑ We must have a primary key for our new table to ensure that each row is unique. For our Customers table, there are two candidate keys — CustomerNo and CustomerName. CustomerNo was actually created just to serve this purpose and seems the logical choice — after all, it's entirely conceivable that you could have more than one customer with the same name. (For example, there have to be hundreds of AA Auto Glass companies in the U.S.)
- ❑ Although we've moved the data out of the Orders table, we still need to maintain a reference to the data in the new Customers table. This is why you still see the CustomerNo (the primary key) column in the Orders table. Later on, when we build our references, we'll create a foreign key constraint to force all orders to have valid customer numbers.

- ❑ We were able to eliminate an instance of the information for ACME Co. That's part of the purpose of moving data that appears in repetitive groups — to just store it once. This both saves us space and prevents conflicting values.
- ❑ We only moved repeating *groups* of data. We still see the same order date several times, but it doesn't really fit into a group — it's just a relatively random piece of data that has no relevance outside of this table.

So, we've dealt with our repeating data; next, we're ready to move onto the second violation of first normal form — atomicity. If you take a look at the ItemsOrdered column, you'll see that there are actually several different pieces of data there:

- ❑ Anywhere from one to many individual part numbers
- ❑ Quantity weight information on each of those parts

Part number, weight, and price are each atomic pieces of data if kept to themselves, but combined into one lump grouping you no longer have atomicity.

Believe it or not, things were sometimes really done this way. At first glance, it seemed the easy thing to do — paper invoices often had just one big block area for writing up what the customer wanted, and computer based systems were often just as close to a clone of paper as someone could make it.

We'll go ahead and break things up — and, while we're at it, we'll add in a new piece of information in the form of a unit price, as shown in Figure 8-1. The problem is that, once we break up this information, our primary key no longer uniquely identifies our rows — our rows are still unique, but the primary key is now inadequate.

Order No (PK)	Order Date	Customer No	Part No	Description	Qty	Unit Price	Total Price	Wt.
100	1/1/1999	54545	1A4536	Flange	5	15	75	6
100	1/1/1999	54545	0R2400	Injector	4	27	108	.5
100	1/1/1999	54545	0R2403	Injector	4	29	116	.5
100	1/1/1999	54545	4I5436	Head	1	750	750	3
101	1/1/1999	12000	3X9567	Pump	1	62.50	62.50	5
102	1/1/1999	66651	8G9200	Fan	7	12	84	3
102	1/1/1999	66651	8G5437	Fan	1	15	15	3
102	1/1/1999	66651	3H6250	Control	1	32	32	5
103	1/2/1999	54545	8G9200	Fan	40	12	480	3
103	1/2/1999	54545	2P5523	Housing	1	165	165	1
103	1/2/1999	54545	3X9567	Pump	1	42	42	5

Figure 8-1

For now, we'll address this by adding a line item number to our table, as shown in Figure 8-2, so we can, again, uniquely identify our rows.

Order No (PK)	Line Item (PK)	Order Date	Customer No	Part No	Description	Qty	Unit Price	Total Price	Wt.
100	1	1/1/1999	54545	1A4536	Flange	5	15	75	6
100	2	1/1/1999	54545	0R2400	Injector	4	27	108	.5
100	3	1/1/1999	54545	0R2403	Injector	4	29	116	.5
100	4	1/1/1999	54545	4I5436	Head	1	750	750	3
101	1	1/1/1999	12000	3X9567	Pump	1	62.50	62.50	5
102	1	1/1/1999	66651	8G9200	Fan	7	12	84	3
102	2	1/1/1999	66651	8G5437	Fan	1	15	15	3
102	3	1/1/1999	66651	3H6250	Control	1	32	32	5
103	1	1/2/1999	54545	8G9200	Fan	40	12	480	3
103	2	1/2/1999	54545	2P5523	Housing	1	165	165	1
103	3	1/2/1999	54545	3X9567	Pump	1	42	42	5

Figure 8-2

Rather than create another column as we did here, we also could have taken the approach of making PartNo part of our primary key. The fallout from this would have been that we could not have had the same part number appear twice in the same order. We'll briefly discuss keys based on more than one column—or composite keys—in our next chapter.

At this point, we meet our criteria for first normal form. We have no repeating groups of data, and all columns are atomic. We do have issues with data having to be repeated within a column (because it's the same for all rows for that primary key), but we'll deal with that shortly.

The Second Normal Form

The next phase in normalization is to go to the second normal form (2NF). Second normal form 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.

Our example has a problem—actually, it has a couple of them—in this area. Let's look at the first normal form version of our Orders table again (Figure 8-2—is every column dependent on the whole key? Are there any that need only part of the key?

The answers are no and yes respectively. There are two columns that only depend on the OrderNo column—not the LineItem column. The columns in question are OrderDate and CustomerNo; both are the same for the entire order regardless of how many line items there are. Dealing with these requires that we introduce yet another table. At this point, we run across the concept of a *header* vs. a *detail* table for the first time.

Being Normal: Normalization and Other Basic Design Issues

Sometimes what is, in practice, one entity still needs to be broken out into two tables and, thus, two entities. The header is something of the parent table of the two tables in the relationship. It contains information that only needs to be stored once while the detail table stores the information that may exist in multiple instances. The header usually keeps the name of the original table, and the detail table usually has a name that starts with the header table name and adds on something to indicate that it is a detail table (for example, `OrderDetails`). For every one header record, you usually have at least one detail record and may have many, many more. This is one example of a kind of relationship (a one-to-many relationship) that we will look at in the next major section.

So let's take care of this by splitting our table again. We'll actually start with the detail table since it's keeping the bulk of the columns. From this point forward, we'll call this table `OrderDetails`:

OrderNo (PK)	LineItem (PK)	PartNo	Description	Qty	Unit Price	Total Price	Wt
100	1	1A4536	Flange	5	15	75	6
100	2	OR2400	Injector	4	27	108	.5
100	3	OR2403	Injector	4	29	116	.5
100	4	4I5436	Head	1	750	750	3
101	1	3X9567	Pump	1	62.50	62.50	5
102	1	8G9200	Fan	7	12	84	3
102	2	8G5437	Fan	1	15	15	3
102	3	3H6250	Control	1	32	32	5
103	1	8G9200	Fan	40	12	480	3
103	2	2P5523	Housing	1	165	165	1
103	3	3X9567	Pump	1	42	42	5

Then we move on to what, although you could consider it to be the new table of the two, will serve as the header table and thus keep the `Orders` name:

OrderNo (PK)	OrderDate	CustomerNo
100	1/1/1999	54545
101	1/1/1999	12000
102	1/1/1999	66651
103	1/2/1999	54545

So, now we have second normal form. All of our columns depend on the entire key. I'm sure you won't be surprised to hear that we still have a problem or two though — we'll deal with them next.

The Third Normal Form

This is the relative end of the line. There are technically levels of normalization beyond this, but none that get much attention outside of academic circles. We'll look at those extremely briefly next, but first we need to finish the business at hand.

I mentioned at the end of our discussion of second normal form that we still had problems — we still haven't reached third normal form (3NF). Third normal form deals with the issue of having all the columns in our 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.

We already know that we're in second normal form, so let's look at the other two rules.

First, do we have any columns that have dependencies other than the primary key? Yes! Actually, there are a couple of columns that are dependent on the PartNo as much as, if not more than, the primary key of this table. Weight and Description are both entirely dependent on the PartNo column — we again need to split into another table.

Your first tendency here might be to also lump UnitPrice into this category, and you would be partially right. The Products table that we will create here can and should have a UnitPrice column in it — but it will be of a slightly different nature. Indeed, perhaps it would be better named ListPrice, as it is the cost we have set in general for that product. The difference for the UnitPrice in the OrderDetails table is twofold. First, we may offer discounts that would change the price at time of sale. This means that the price in the OrderDetails record may be different than the planned price that we will keep in the Products table. Second, the price we plan to charge will change over time with factors such as inflation, but changes in future prices will not change what we have charged on our actual orders of the past. In other words, price is one of those odd circumstances where there are really two flavors of it — one dependent on the PartNo, and one dependent on the primary key for the OrderDetails table (in other words OrderID and LineItem).

First, we need to create a new table (we'll call it Products) to hold our part information. This new table will hold the information that we had in OrderDetails that was more dependent on PartNo than on OrderID or LineItem:

PartNo (PK)	Description	Wt
1A4536	Flange	6
OR2400	Injector	.5
OR2403	Injector	.5
4I5436	Head	3

PartNo (PK)	Description	Wt
3X9567	Pump	5
8G9200	Fan	3
8G5437	Fan	3
3H6250	Control	5
8G9200	Fan	3
2P5523	Housing	1
3X9567	Pump	5

We can then chop all but the foreign key out of the `OrderDetails` table:

OrderNo (PK)	LineItem (PK)	PartNo	Qty	Unit Price	Total Price
100	1	1A4536	5	15	75
100	2	OR2400	4	27	108
100	3	OR2403	4	29	116
100	4	4I5436	1	750	750
101	1	3X9567	1	62.50	62.50
102	1	8G9200	7	12	84
102	2	8G5437	1	15	15
102	3	3H6250	1	32	32
103	1	8G9200	40	12	480
103	2	2P5523	1	165	165
103	3	3X9567	1	42	42

That takes care of problem number 1 (cross-column dependency), but doesn't deal with derived data. We have a column called `TotalPrice` that contains data that can actually be derived from multiplying `Qty` by `UnitPrice`. This is a no-no in normalization.

Derived data is one of the places that you'll see me "de-normalize" data most often. Why? Speed! A query that reads `WHERE TotalPrice > $100` runs faster than one that reads `WHERE Qty * UnitPrice > 50`—particularly if we are able to index our computed `TotalPrice`.

On the other side of this, however, I do sometimes take more of a hybrid approach by utilizing a computed column and letting SQL Server keep a sum of the other two columns for us (you may recall us using this idea for our `PreviousSalary` example in the `Employees` table of the Accounting database in Chapter 5). If this is a very important column from a performance perspective (you're running lots of columns that filter based on the values in this column), then you may want to add an index to your new computed column. The significance of this is that the index "materializes" the computed data. What does that mean? Well, it means that even SQL Server doesn't have to calculate the computed column on-the-fly—instead, it calculates it once when the row is stored in the index, and, thereafter, uses the precalculated column. It can be very fast indeed, and we'll examine it further in Chapter 9.

So, to reach third normal form, we just need to drop off the `TotalPrice` column and compute it when needed.

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 multi-valued 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. As such, they are largely ignored in the database world, and we 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 we 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
- ❑ One-to-many
- ❑ Many-to-many

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.

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.

To illustrate a one-to-one relationship, let’s look at a slight variation of a piece of our earlier example. Imagine that you have customers—just as we did in our earlier example. This time, however, we’re going to imagine that we are a subsidiary of a much larger company. Our parent company wants to be able to track all of its customers, and to be able to tell the collective total of each customer’s purchases—regardless of which subsidiary(s) the customer made purchases with.

Even if all the subsidiaries run out of one server at the main headquarters, there’s a very good chance that the various subsidiaries would be running with their own databases. One way to track all customer information, which would facilitate combining it later, would be to create a master customer database owned by the parent company. The subsidiaries would then maintain their own customer table, but do so with a one-to-one relationship to the parent company’s customer table. Any customer record created in the parent company would imply that you needed to have one in the subsidiaries also. Any creation of a customer record in a subsidiary would require that one was also created in the parent company’s copy.

A second example—one that used to apply frequently to SQL Server prior to version 7.0—is the situation where you have too much information to fit in one row. Remember that the maximum row size for SQL Server is 8060 bytes of non-BLOB data. That’s a lot harder to fill than version 6.5’s 1962 bytes, but

you can still have situations where you need to store a very large number of columns or even fewer very wide columns. One way to get around this problem was to actually create two different tables and split our rows between the tables. We could then impose a one-to-one relationship. The combination of the matching rows in the two tables then meets our larger row size requirement.

SQL Server has no inherent method of enforcing a true one-to-one relationship. You can say that table A requires a matching record in table B, but when you then add that table B must have a matching record in table A, you create a paradox—which table gets the record first? If you need to enforce this kind of relationship in SQL Server, the best you can do is force all inserts to be done via a stored procedure. The stored procedure can have the logic to insert into both tables or neither table. Neither foreign key constraints nor triggers can handle this circular relationship.

Zero or One-to-One

SQL Server can handle the instance of zero or one-to-one relationships. This is essentially the same as a one-to-one, with the difference that one side of the relationship has the option of either having a record or not.

Going back to our parent company vs. subsidiary example, you might prefer to create a relationship where the parent company needs to have a matching record for each subsidiary's records, but the subsidiary doesn't need the information from the parent. You could, for example, have subsidiaries that have very different customers (such as a railroad and a construction company). The parent company wants to know about *all* the customers regardless of what business they came from, but your construction company probably doesn't care about your railroad customers. In such a case, you would have *zero or one* construction customers to *one* parent company customer record.

Zero or one-to-one relationships can be enforced in SQL Server through:

- ❑ A combination of a unique or primary key with a foreign key constraint. A foreign key constraint can enforce that *at least* one record must exist in the "one" (or parent company in our example) table, but it can't ensure that *only* one exists (there could be more than one). Using a primary key or unique constraint would ensure that one was indeed the limit.
- ❑ Triggers. Note that triggers would be required in both tables.

The reason SQL Server can handle a zero or one-to-one, but not a one-to-one relationship is due to the "which goes first" problem. In a true one-to-one relationship, you can't insert into either table because the record in the other table isn't there yet—it's a paradox. However, with a zero or one-to-one, you can insert into the required table first (the "one"), and the optional table (the zero or one), if desired, second. This same problem will hold true for the "one-to-one or many" and the "one to zero, one, or many" relationships also.

One-to-One or Many

This is one form of your run-of-the-mill, average, every-day foreign key kind of relationship. Usually, this is found in some form of header/detail relationship. A great example of this would be our Orders situation, as shown in Figure 8-3. *OrderDetails* (the one or many side of the relationship) doesn't make much sense without an *Orders* header to belong to (does it do you much good to have an order for a part if you don't know who the order is for?). Likewise, it doesn't make much sense to have an order if there wasn't anything actually ordered (for example, "Gee, look, ACME company ordered absolutely nothing yesterday.").

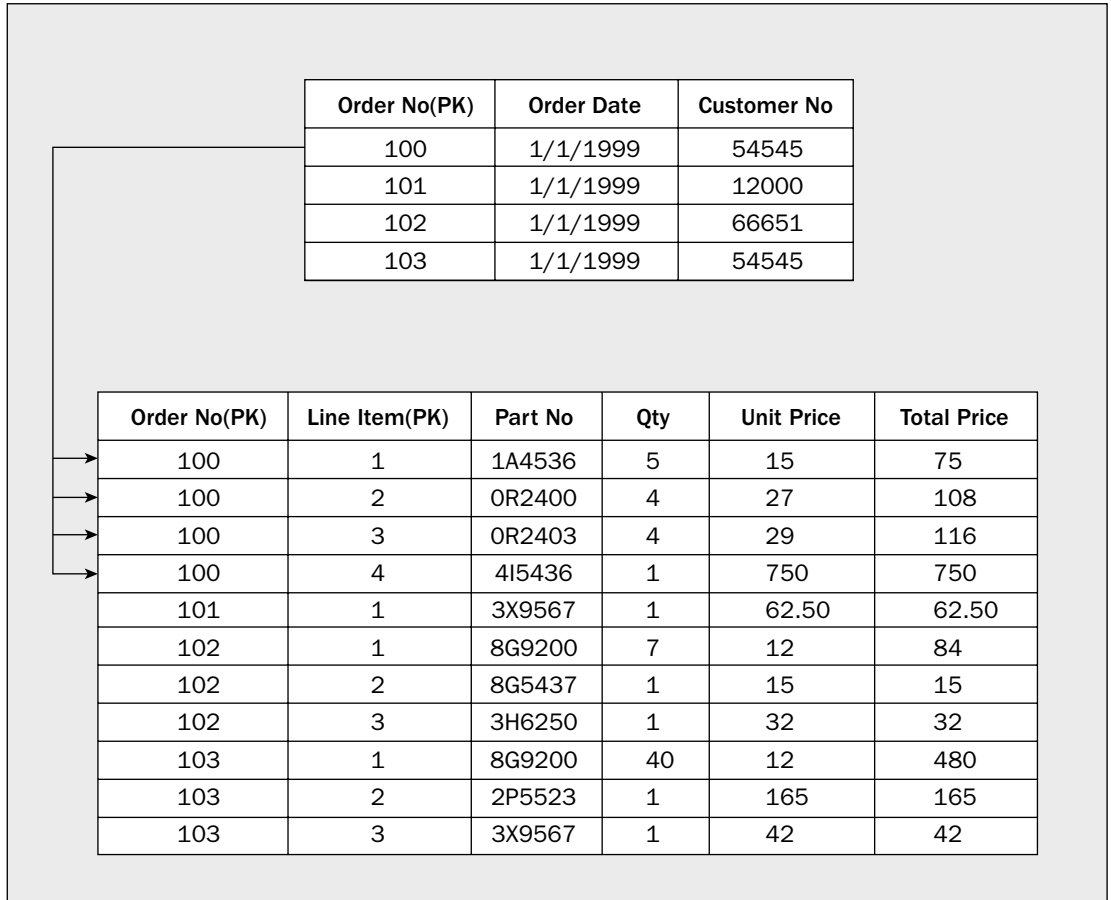


Figure 8-3

This one, however, gives us the same basic problem that we had with one-to-one relationships. It's still that chicken or egg thing — which came first? Again, in SQL Server, the only way to fully implement this is by restricting all data to be inserted or deleted via stored procedures.

One-to-Zero, One, or Many

This is the other, and perhaps even more common, form of the run-of-the-mill, average, everyday, foreign key relationship. The only real difference in implementation here is that the referencing field (the one in the table that has the foreign key constraint) is allowed to be null; that is, the fact that you have a record in the “one” table, doesn’t necessarily mean that you have any instances of matching records in the referencing table.

An example of this can be found in the Northwind database in the relationship between Suppliers and Orders. The Orders table tracks which shipper was used to ship the order—but what if the order was picked up by the customer? If there is a shipper, then we want to limit it to our approved list of shippers, but it’s still quite possible that there won’t be any shipper at all, as illustrated in Figure 8-4.

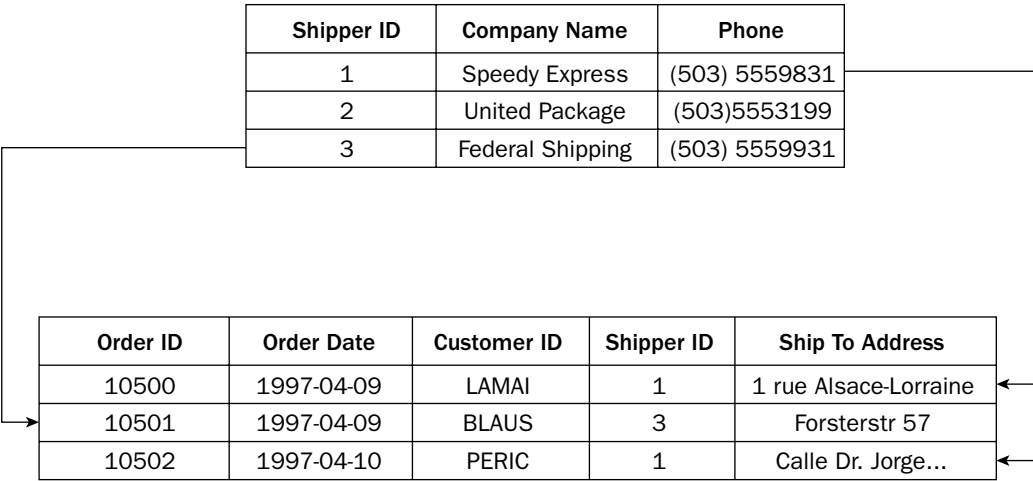


Figure 8-4

A virtually identical example can be found in the AdventureWorks database in the relationship between Purchasing.PurchaseOrderHeader and Purchasing.ShipMethod, with the only real difference being that this is a list of companies shipping *to* us rather than *from* us.

This kind of relationship usually sets up what is called a *domain relationship*. A *domain* is a limited list of values that the dependent table must choose from—nothing outside the domain list is considered a valid option. The table that holds the rows that make up the domain list is commonly referred to as a *domain* or *lookup table*. Nearly all databases you create are going to have at least one, and probably many, domain tables in them. Our Shippers table is a domain table—the purpose of having it isn’t just to store the information on the name and phone number of the shipper, but also to limit the list of possible shippers in the Orders table.

In SQL Server, we can enforce this kind of relationship through two methods:

- ❑ **FOREIGN KEY constraint:** You simply declare a `FOREIGN KEY` constraint on the table that serves as the “many” side of the relationship, and reference the table and column that is to be the “one” side of the relationship (you’ll be guaranteed of only one in the referenced table since you must have a `PRIMARY KEY` or `UNIQUE` constraint on the column(s) referenced by a foreign key).
- ❑ **Triggers:** Actually, for all the early versions of SQL Server, this was the only option for true referential integrity. You actually need to add two triggers — one for each side of the relationship. Add a trigger to the table that is the “many” side of the relationship and check that any row inserted or changed in that table has a match in the table it depends on (the “one” side of the relationship). Then, you add a delete and update triggers to the other table — this trigger checks records that are being deleted (or changed) from the referenced table to make sure that it isn’t going to *orphan* (make it so it doesn’t have a reference).

We’ve previously discussed the performance ramifications of the choices between the two in our chapter on constraints. Using a `FOREIGN KEY` constraint is generally faster — particularly when there is a violation. That being said, triggers may still be the better option in situations where you’re going to have a trigger executing anyway (or some other special constraint need).

Many-to-Many

In this type of relationship, both sides of the relationship may have several records — not just one — that match. An example of this would be the relationship of products to orders. A given order may have many different products in the order. Likewise, any given product may be ordered many times. We still may, however, want to relate the tables in question — for example, to ensure that an order is for a product that we know about (it’s in our Products table).

SQL Server has no way of physically establishing a direct many-to-many relationship, so we cheat by having an intermediate table to organize the relationship. Some tables create our many-to-many relationships almost by accident as a normal part of the normalization process — others are created entirely from scratch for the sole purpose of establishing this kind of relationship. This latter “middleman” kind of table is often called either a *linking table*, an *associate table*, or sometimes a *merge table*.

First, let’s look at a many-to-many relationship that is created in the normal course of normalization. An example of this can be found in the Northwind database’s `OrderDetails` table, which creates a many-to-many relationship between our `Orders` and `Products` tables, shown in Figure 8-5.

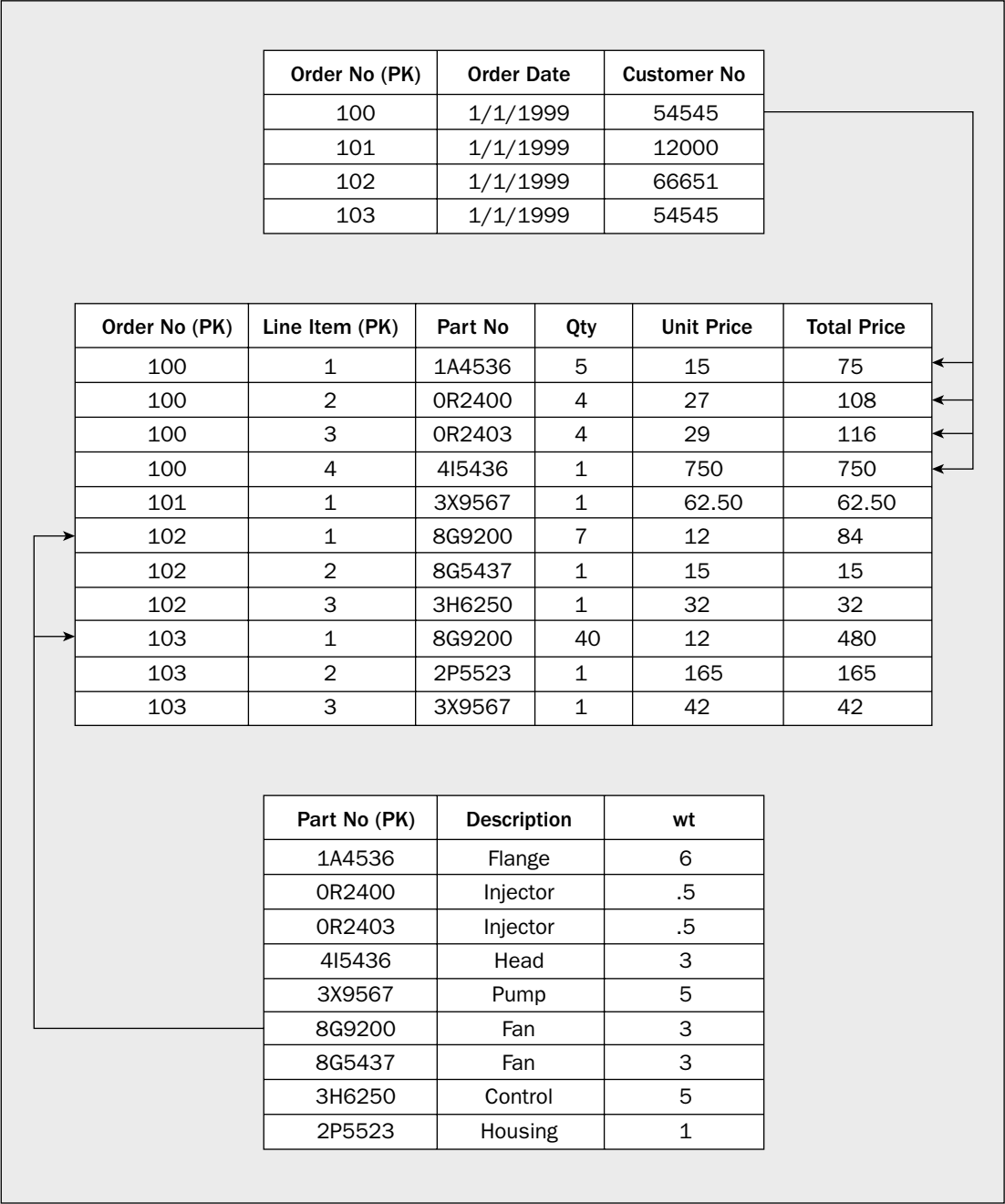


Figure 8-5

Being Normal: Normalization and Other Basic Design Issues

By using the join syntax that we learned back in Chapter 4, we can relate one product to the many orders that it's been part of, or we can go the other way and relate an order to all the products on that order.

Let's move on now to our second example—one where we create an associate table from scratch just so we can have a many-to-many relationship. We'll take the example of a user and a group of rights that a user can have on the system.

We might start with a Permissions table that looks something like this:

PermissionID	Description
1	Read
2	Insert
3	Update
4	Delete

Then we add a Users table:

UserID	Full Name	Password	Active
JohnD	John Doe	Jfz9..nm3	1
SamS	Sam Spade	klk93)md	1

Now comes the problem—how do we define what users have which permissions? Our first inclination might be to just add a column called Permissions to our Users table:

UserID	Full Name	Password	Permissions	Active
JohnD	John Doe	Jfz9..nm3	1	1
SamS	Sam Spade	klk93)md	3	1

This seems fine for only a split second, and then a question begs to be answered—what about when our users have permission to do more than one thing?

In the older, flat file days, you might have just combined all the permissions into the one cell, like:

UserID	Full Name	Password	Permissions	Active
JohnD	John Doe	Jfz9..nm3	1,2,3	1
SamS	Sam Spade	klk93)md	1,2,3,43	1

Chapter 8

This violates our first normal form, which said that the values in any column must be atomic. In addition, this would be very slow because you would have to procedurally parse out each individual value within the cell.

What we really have between these two tables, Users and Permissions, is a many-to-many relationship—we just need a way to establish that relationship within the database. We do this by adding an associate table, as shown in Figure 8-6. Again, this is a table that, in most cases, doesn’t add any new data to our database other than establishing the association between rows in two other tables.

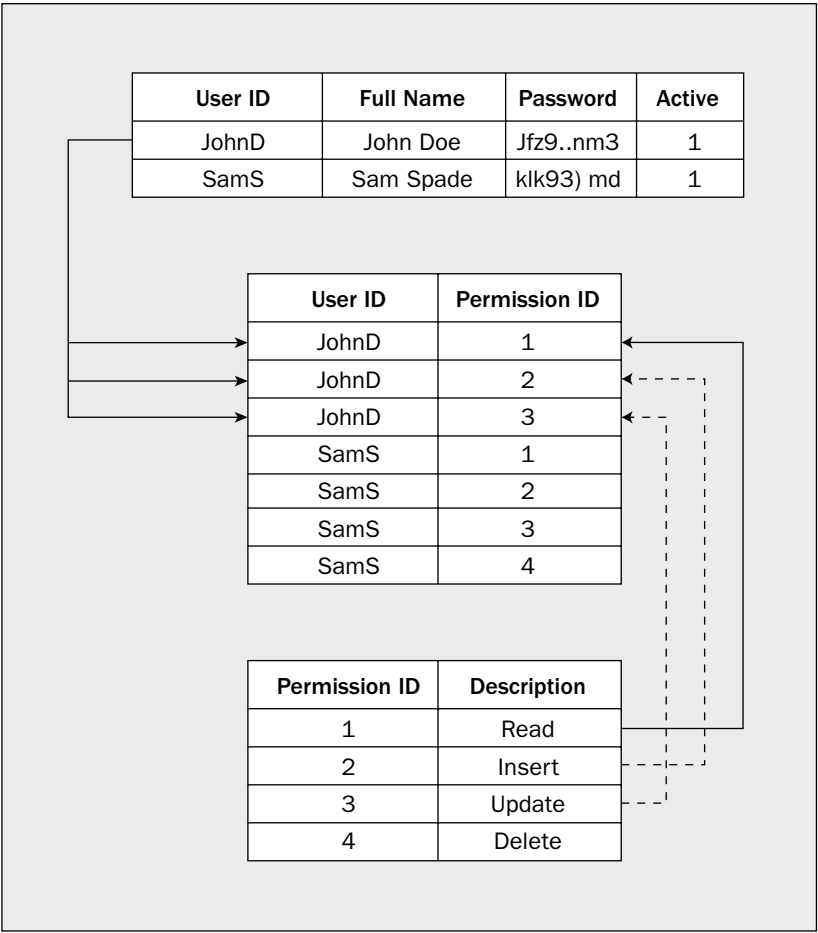


Figure 8-6

With the addition of our new table (we’ll call it UserPermissions), we can now mix and match our permissions to our users.

Note that, for either example, the implementation of referential integrity is the same—each of the base tables (the tables that hold the underlying data and have the many-to-many relationship) has a one-to-many relationship with the associate table. This can be done via either a trigger or a FOREIGN KEY constraint.

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.

Fortunately, SQL Server includes a *very* basic diagramming tool that you can use as a starting point for building rudimentary ERDs.

Before the first time I wrote this topic, I debated for a long while about how I wanted to handle this. On one hand, serious ER diagramming is usually done with an application that is specifically designed to be an ER diagramming tool (we talk about a few of these in Appendix C). These tools almost always support at least one of a couple of industry standard diagramming methods. Even some of the more mass-market diagramming tools—such as Visio—support a couple of ERD methodologies. SQL Server has an ERD tool built in, and therein lies the problem. The tools that are included with SQL Server 2005 are a variation on a toolset and diagramming methodology that Microsoft has used in a number of tools for many years now. The problem is that they do not complete with any ERD standard that I’ve seen anywhere else. As I’ve done every time I’ve written on this topic, I’ve decided to stick with what I know you have—the built-in tools. I do, however, encourage you to examine the commercially available ERD tools out there to see the rich things that they offer to simplify your database design efforts.

You can open up SQL Server’s build 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 we are going to see you’ll find familiar—some of the dialogs are the same as we saw in Chapter 5 when we 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. Indeed, if you’re familiar with the relationship editor in Access, then much of the SQL Server tools will seem very familiar.

Try It Out Diagramming

Let’s start by creating our first diagram. You can create your new diagram by right-clicking the Diagrams node underneath the Northwind database and choosing the New Database Diagram option.

As we saw back in Chapter 5, 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.

SQL Server starts us out with the same Add Table dialog (see Figure 8-7) we saw back in Chapter 5—the only thing different is the tables listed.

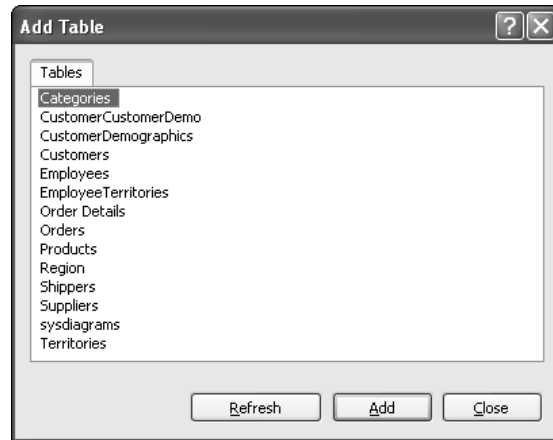


Figure 8-7

Select all of the tables (remember to hold down the control key to select more than one table) except for the sysdiagrams table (you may remember this is actually a system table that exists only to support diagramming) as show in Figure 8-8.

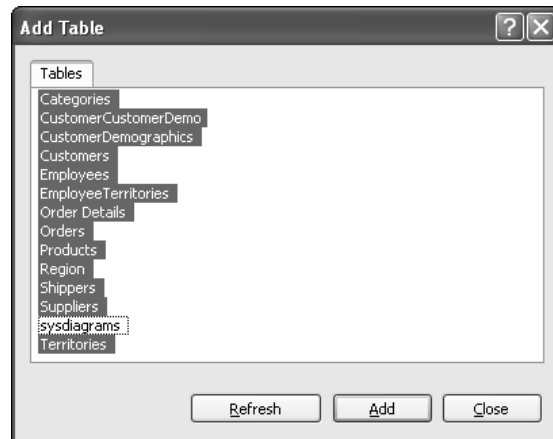


Figure 8-8

And then click Add and, after a brief pause while SQL Server draws all the tables you selected, the Close button. SQL Server has added our tables to the diagram, but, depending on your screen resolution, they are probably very difficult to see due to the zoom on the diagram. To pull more of the tables into view, change the zoom setting in the toolbar. Finding the right balance between being able to see many tables at once versus making them so small you can't read them is a bit of a hassle, but you should be able to adjust to something that meets your particular taste — for now, I've set mine at 70 percent so I can squeeze in all the tables at once (usually not that realistic on a database of any real size table count wise) as shown in Figure 8-9.

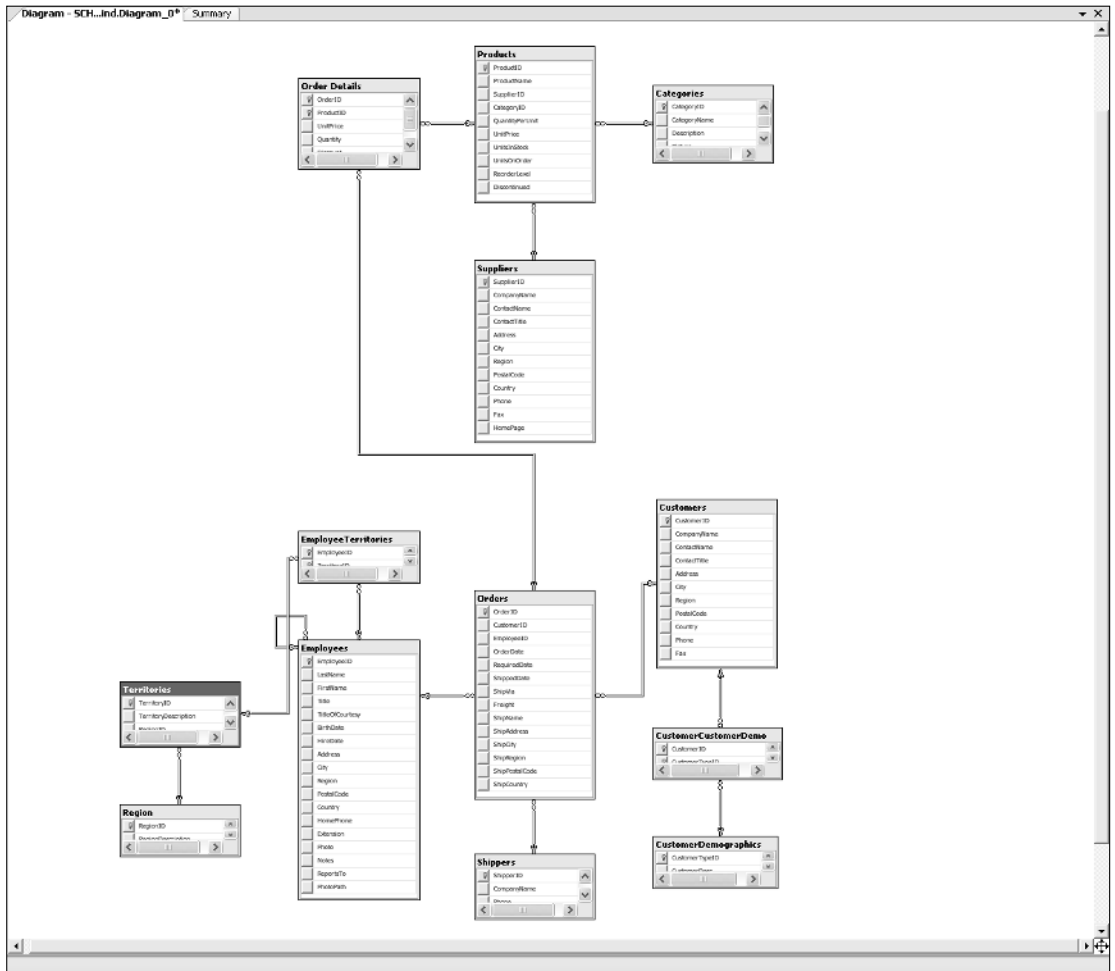


Figure 8-9

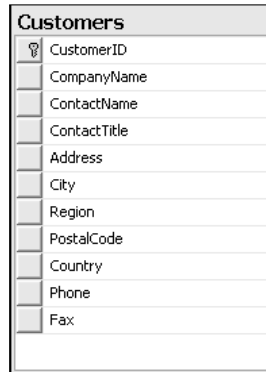
How It Works

You'll notice right away that there is a lot more than what we saw with our first look at the diagram tools back in Chapter 5. SQL Server enumerates through each table we have said we 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, let's use this diagram as a launching point for explaining how the diagramming tool works and building a few tables here and there.

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 in Figure 8-10.



CustomerID	
CompanyName	
ContactName	
ContactTitle	
Address	
City	
Region	
PostalCode	
Country	
Phone	
Fax	

Figure 8-10

Just like in Chapter 5, this is just the default view for the table—you can select from several others that allow you to edit the very makeup of the table. To check out your options for views of a table, just 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 as we did in Chapter 5; this or “standard” is what you would use when you want to edit the table from right within the diagram (very nice!).

Adding and Deleting Tables

You can add a new diagram to the table 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. 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, datatypes, 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.

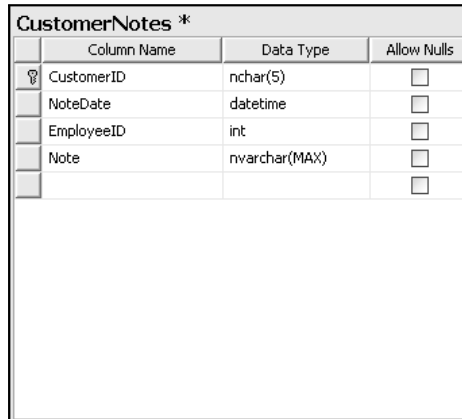
First, don’t forget to add a Primary Key to your table. SQL Server does not automatically do this, nor does it even prompt you (as Access does). This is a somewhat less than intuitive process. To add a Primary Key, you must select the columns that you want to have in the key. Then right-click and choose “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.

Try It Out Adding Tables From Within The Diagram

Let's go ahead and add a table to our database just to show how it works.

Start by clicking on the New Table button in the diagramming window's toolbar. When prompted for a name, choose a name of CustomerNotes (see Figure 8-11). You should then get a new window table up using the Standard view:



	Column Name	Data Type	Allow Nulls
	CustomerID	nchar(5)	☐
	NoteDate	datetime	☐
	EmployeeID	int	☐
	Note	nvarchar(MAX)	☐
			☐

Figure 8-11

Notice that I've added several columns to my table along with a Primary Key (remember, select the columns you want to be the primary key, and then right-click and choose Set Primary Key). Before you click to save this, let's try something out — open up the Management Studio, and try and run a query against your new table:

```
SELECT * FROM CustomerNotes
```

Back comes an error message:

```
Msg 208, Level 16, State 1, Line 1  
Invalid object name 'CustomerNotes'.
```

That's because our table exists only as an edited item on the diagram — it won't be added until we actually save our changes.

If you look at the CustomerNotes table in the diagram window at this point, you should see a * to the right of the name — that's there to tell you that there are unsaved changes in that table.

Now, switch back to the Management Studio. There are two save options:

- ❑ **Save:** This saves the changes to both the diagram and to the database (this is the little disk icon on the toolbar).
- ❑ **Save Change Script:** This saves the changes to a script so it can be run at a later time (This is found in the Table Designer menu or as a “Save To Text” if you use the disk save button from the previous bullet).

Go ahead and just choose Save, and you’ll be prompted for confirmation (after all, you’re about to alter your database — there’s no “undo” for this):

Confirm the changes, and try running that query again against your CustomerNotes table. You should not receive an error this time because the table has now been created (you won’t get any rows back, but the query should still run).

How It Works

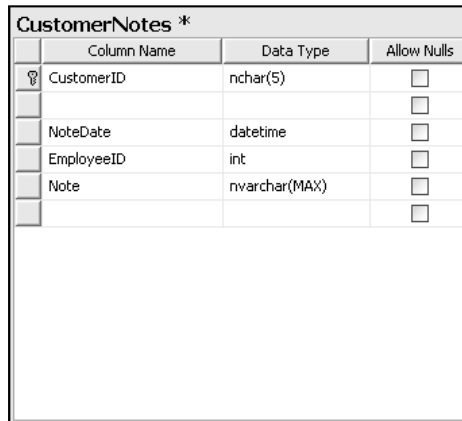
When we create a diagram, SQL Server creates script behind the scenes that looks basically just as our scripts did back in Chapter 6 when we were scripting our own changes. However, these scripts are not actually generated and run until we choose to save the diagram.

OK, we’ve got our CustomerNotes table into the database, but now we notice a problem — the way our Primary Key is declared, we can only have one note per customer. More than likely, we are going to keep taking more and more notes on the customer over time. That means that we need to change our Primary Key, and leaves us with a couple of options depending on our requirements:

- ❑ **Make the date part of our Primary Key:** This is problematic from two standpoints. First, we’re tracking what employee took the note — what if two different employees wanted to add notes at the same time? We could, of course, potentially address this by also adding EmployeeID to the Primary Key. Second, what’s to say that even the same employee wouldn’t want to enter to completely separate notes on the same day (OK, so, since this is a datetime field, they could do it as long as they didn’t get two rows inserted at the same millisecond — but just play along with me here)? Oops, now even our EmployeeID being in the key doesn’t help us.
- ❑ Add another column to help with the key structure. We could either do this by adding a counter column for each note per customer. As yet another alternative, we could just add an identity column to ensure uniqueness — it means that our Primary Key doesn’t really relate to anything, but that isn’t always a big deal (though it does mean that we have one more index that has to be maintained) and it does allow us to have a relatively unlimited number of notes per customer.

I’m going to take the approach of adding a column I’ll call “Sequence” to the table. By convention (it’s not a requirement and not everyone does it this way), Primary Keys are normally the first columns in your table. If we were going to be doing this by script ourselves, we’d probably just issue an ALTER TABLE statement and ADD the column — this would stick our new column down at the end of our column list. If we wanted to fix that, we’d have to copy all the data out to a holding table, drop any relationships to or from the old table, drop the old table, CREATE a new table that has the columns and column order we want, then re-establish the relationships and copy the data back in (a long and tedious process). With the diagramming tools, however, SQL Server takes care of all that for us.

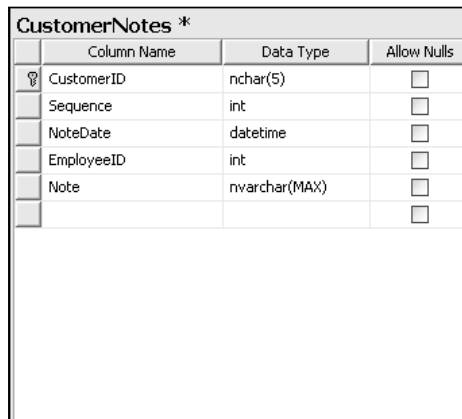
To insert a new row in the middle of everything, I just right-click on the row that is to immediately follow the row I want to insert. The tool is nice enough to bump everything down for me to create space just like Figure 8-12.



	Column Name	Data Type	Allow Nulls
PK	CustomerID	nchar(5)	☐
	NoteDate	datetime	☐
	EmployeeID	int	☐
	Note	nvarchar(MAX)	☐
			☐

Figure 8-12

I can then add in my new column, and reset the Primary Key as shown in Figure 8-13 (select both rows, right-click and choose Set Primary Key).



	Column Name	Data Type	Allow Nulls
PK	CustomerID	nchar(5)	☐
	Sequence	int	☐
	NoteDate	datetime	☐
	EmployeeID	int	☐
	Note	nvarchar(MAX)	☐
			☐

Figure 8-13

Now just save, and you have a table with the desired column order. Just to verify this, try using `sp_help`:

```
EXEC sp_help CustomerNotes
```

Chapter 8

You'll see that we have the column order we expect:

```
....  
CustomerID  
Sequence  
NoteDate  
EmployeeID  
Note  
....
```

Making things like column order changes happens to be one area where the daVinci tools positively excel. I've used a couple of other ERD tools, and they all offered the promise of synchronizing a change in column order between the database and the diagram — the success has been pretty hit and miss. (In other words, be very careful about doing it around live data.) The tools are getting better, but this is an area where the daVinci tools show some of the genius of their namesake.

Also, under the heading of one more thing — use the scripting option rather than the live connection to the database to make changes like this if you're operating against live data. That way you can fully test the script against test databases before risking your real data. Be sure to also fully back up your database before making this kind of change.

Editing Table Properties and Objects That Belong to the Table

Beyond the basic attributes that we've looked at thus far, we can also edit many other facets of our table. How to get at these to edit or add to them happens in two different ways:

Properties: These are edited in a window that pops up and docks, by default, on the right-hand side of the Management Studio inside the diagramming window. To bring up the properties window, click the "Properties Window" icon on the toolbar in the Management Studio.

Objects that belong to the table, such as Indexes, Constraints, and Relationships: These are edited in their own dialog which you can access by right-clicking on the table in the diagram and choosing the item that you want to set.

These are important facets of our diagram-based editing, so let's look at some of the major players.

Properties Window

Figure 8-14 shows the Properties Window for our CustomerNotes table:

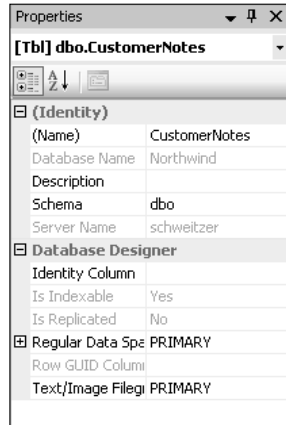


Figure 8-14

You can use this properties window to set several key table properties — most notably what schema the table belongs to as well as whether the table has an Identity column.

Relationships

Much like it sounds, this dialog allows us to edit the nature of the relationships between tables. As you can see from Figure 8-15, the relationships for the CustomerNotes table doesn't yet have anything in it.

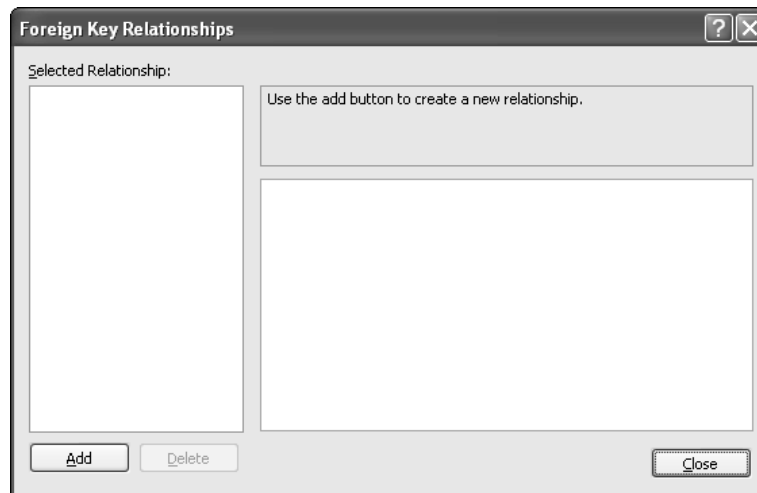


Figure 8-15

Chapter 8

For now, just realize that we can edit just about anything to do with relationships here. We could, for example, create a relationship to another table just by clicking Add and filling out the various boxes. Again, we'll look into this farther in a page or two.

Indexes/Keys

A lot of this one may be something of a mystery to you at this point—we haven't gotten to our chapter on indexing yet, so some of the terms may seem a bit strange. Still, let's take a look at what we get in Figure 8-16.

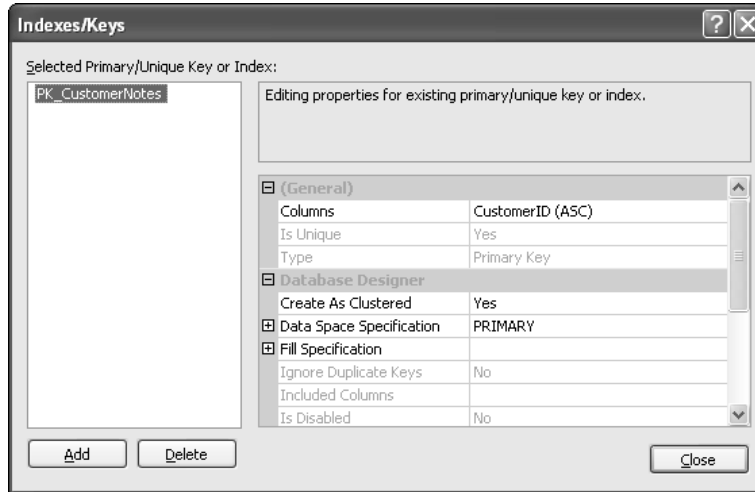


Figure 8-16

From here, you can, as I'm sure you can imagine, create, edit, and delete indexes. You can also establish what filegroup you want the index to be stored on (in most instances, you'll just want to leave this alone). We'll look further into indexes in our next chapter.

Check Constraints

Notice that we're only doing *Check* constraints on this tab as shown in Figure 8-17. (Keys and Defaults have been dealt with in the other tabs.)

Again, this one is pretty much grayed out. Why? Well, there aren't any constraints of any kind other than a Primary Key defined for our CustomerNotes table, and that Primary Key is dealt with on the Index/Keys tab. This one is CHECK Constraints only—if you want to see this tab in full action, then you need to click Add and add a constraint.

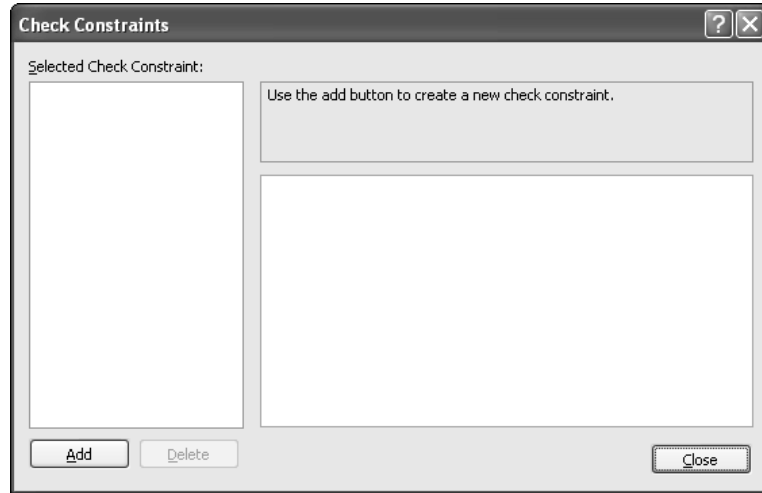


Figure 8-17

Relationships

Well, we’ve seen what the diagramming tool offers us relative to tables, so, as promised, next up on our list to review is the relationship line (and the underlying details of that relationship).

Looking at a relationship line, the side with the key is the side that is the “one” side of a relationship. The side that has the infinity symbol represents your “Many” side. The tools have no relationship line available to specifically represent relationships where zero is possible (it still uses the same line). In addition, the only relationships that actually show in the diagram are ones that are declared using Foreign Key constraints. Any relationship that is enforced via triggers — regardless of the type of relationship — will not cause a relationship line to appear.

Looking at our Northwind diagram again, and try right-clicking either the Customers or Orders table and selecting Relationships. This brings up a more populated version of the Relationship dialog we looked at in the last section — the Relationships dialog for the Orders table is shown in Figure 8-18.

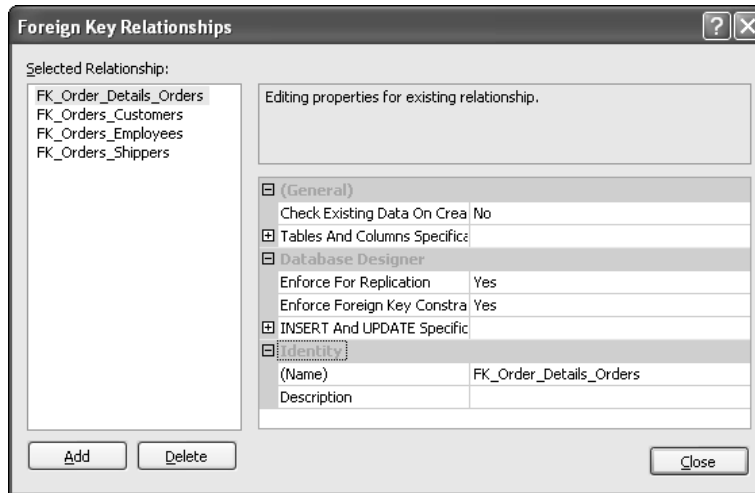


Figure 8-18

From here, we can edit the nature of our relationship, including such things as cascading actions, whether the foreign key is enabled or disabled (for example, if we want to deliberately add data in that violates the relationship), and even the name of the relationship.

Database designers seem to vary widely in their opinion regarding names for relationships. Some don't care what they are named, but I prefer to use a verb phrase to describe my relationships—for example, in our Customers/Orders relationship, I would probably name it CustomerHasOrders or something of that ilk. It's nothing critical—most of the time you won't even use it—but I find that it can be really helpful when I'm looking at a long object list or a particularly complex ER Diagram where the lines may run across the page past several unrelated entities.

Adding Relationships in the Diagramming Tool

Just drag and drop—it's that easy. The only trick is making sure that you start and end your drag in the places you meant to. If in doubt, select the column(s) you're interested in before starting your drag.

Try It Out

Let's add a relationship between our new CustomerNotes table (we created it in the last section) and the Customers table—after all, if it's a customer note we probably want to make sure that we are taking notes on a valid customer. To do this, click and hold in the gray area to the left of the CustomerID column in the Customers table, then drag your mouse until it is pointing at the CustomerID column in the CustomerNotes table. A dialog box should pop up to confirm the column mapping between the related tables (see Figure 8-19).

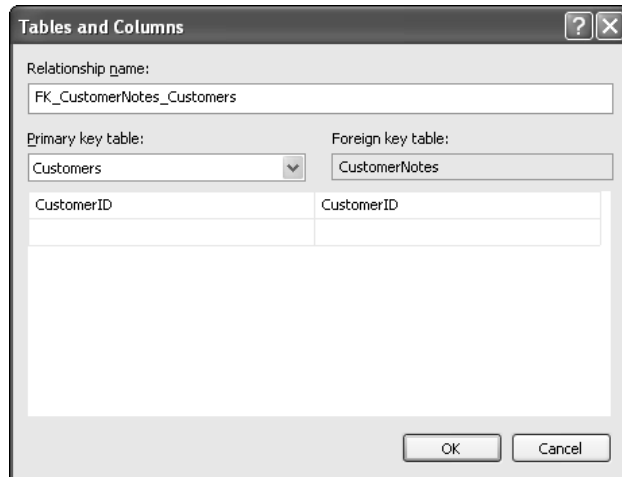


Figure 8-19

If you did your drag and drop right, the names of the columns on both sides of the relationship should come up right to start with, but if they came up with something other than you expected, don't worry too much about — just click the combo box for the table you want to change columns for and select the new column. Changing the name of the relationship from the default of FK_CustomerNotes_Customers to CustomerHasNotes. As soon as you click OK, you will be taken to the more standard relationship dialog so you can set any other settings you may want to adjust before you save the new relationship to the table. Go ahead and change the DELETE and UPDATE CASCADE actions to CASCADE — ensuring that if the related customer record ever gets updated or deleted, the notes for that customer will also get updated or deleted as necessary. You can see what this looks like in Figure 8-20.

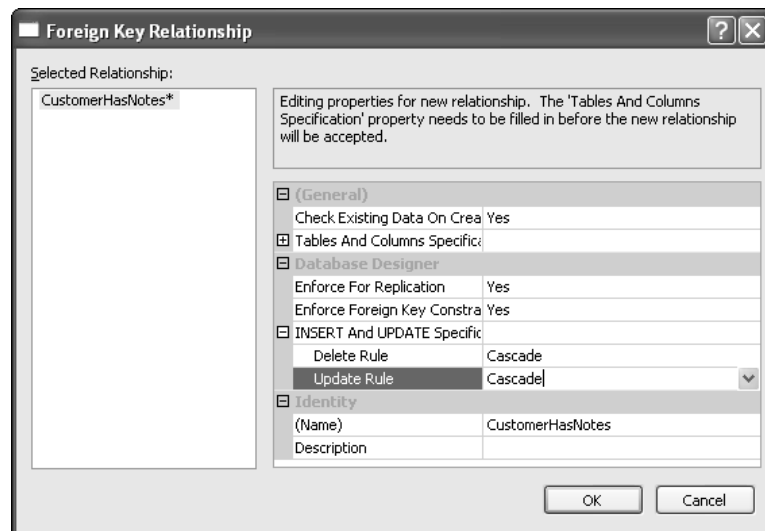


Figure 8-20

With this done, you can click OK and see the new relationship line in your diagram.

How It Works

Much like when we added the table in the previous section, SQL Server is constructing SQL behind the scenes to make the changes you need. So far, it has added the relationship to the diagram only — if you hover over that relationship, you even see a tooltip with the relationship’s name and nature, as shown in Figure 8-21.

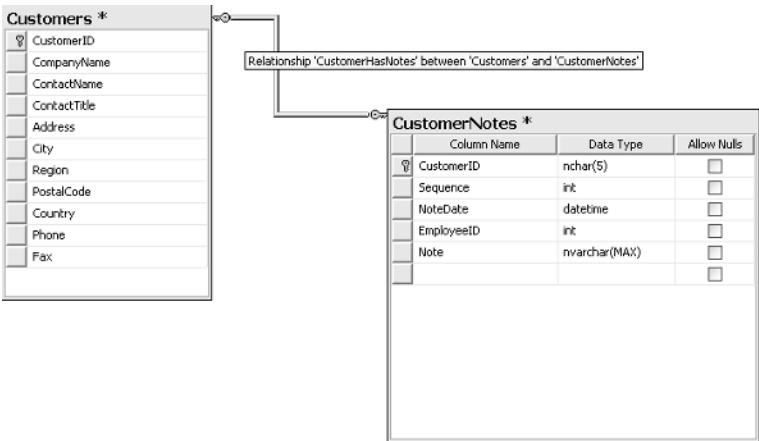


Figure 8-21

Notice the asterisk on both tables!!! The changes we have made have only been added to the change list that you’ve made to the diagram — they will not be added to the physical database until you choose to Save the diagram!

There is an instance where the way the line is displayed will change — when we “disable” the Foreign Key. We saw how to disable constraints in our last chapter, and we can do it in the Relationship dialog by changing the “Enforce Foreign Key Constraint” drop-down to be set to no. When you do that, you’re line will change to let you know that the constraint has been disabled. It will now look something like Figure 8-22.



Figure 8-22

If you see one of these, you’re first question should be “Why is that disabled?” Maybe it was intentional, but you’ll want to be sure.

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 vs. the risk. (For example, what if your "summary" data gets out of synch 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.

Beyond Normalization

In this section, we're going to look into a basic set of "beyond normalization" rules of the road in design. Very few of these are hard and fast kind of rules — they are just things to think about. The most important thing to understand here is that, while normalization is a big thing in database design, it is not the only thing.

Keep It Simple

I run into people on a regular basis that have some really slick ways to do things differently than it's ever been done before. Some of the time, I wind up seeing some ideas that are incredibly cool and incredibly useful. Other times I see ideas that are incredibly cool, but not very useful. As often as not though, I see ideas that are neither — they may be new, but that doesn't make them good.

Before I step too hard on your creative juices here, let me clarify what I'm trying to get across — don't accept the "because we've always done it that way" approach to things, but also recognize that the tried and true probably continues to be tried for a reason — it usually works.

SQL Server 2005 brings all new ways to overdo it in terms of making things too complex. Complex data rules and even complex datatypes are available now through powerful and flexible new additions to the product (code driven functions and datatypes). Try to avoid instilling more complexity in your database than you really need to. A minimalist approach usually (but not always) yields something that is not only easier to edit, but also runs a lot faster.

Choosing Datatypes

In keeping with the minimalist idea, choose what you need, but only what you need.

For example, if you're trying to store months (as the number, 1-12) — those can be done in a single byte by using a `tinyint`. Why then, do I regularly come across databases where a field that's only going to store a month is declared as an `int` (which is 4 bytes)? Don't use an `nchar` or `nvarchar` if you're never going to do anything that requires Unicode — these datatypes take up two bytes for every one as compared to their non-Unicode cousins.

There is a tendency to think about this as being a space issue. When I bring this up in person, I sometimes hear the argument, "Ah, disk space is cheap these days!" Well, beyond the notion that a name-brand SCSI hard drive still costs more than I care to throw away on laziness, there's also a network bandwidth issue. If you're passing an extra 100 bytes down the wire for every row, and you pass a 100 record result, then that's about 10K worth of extra data you just clogged your network with. Still not convinced? Now, say that you have just 100 users performing 50 transactions per hour — that's over 50MB of wasted network bandwidth per hour.

The bottom line is, most things that happen with your database will happen repetitively — that means that small mistakes snowball and can become rather large.

Err on the Side of Storing Things

There was an old movie called *The Man Who Knew Too Much* — Hitchcock I believe — that man wasn't keeping data.

Every time that you're building a database, you're going to come across the question of, "Are we going to need that information later?" Here's my two-bit advice on that — if in doubt, keep it. You see, most of the time you can't get back the data that has already come and gone.

I guarantee that at least once (and probably many, many more times than that), there will be a time where a customer (remember, customers are basically anyone who needs something from you — there is such a thing as an internal customer, not just the ones in Accounts Receivable) will come to you and say something like, “Can you give me a report on how much we paid each non-incorporated company last year?”

OK, so are you storing information on whether your vendor is a corporation or not? You had better be if you are subject to U.S. tax law (1099 reporting). So you turn around and say that you can handle that, and the customer replies, “Great! Can you print that out along with their address as of the end of the year?”

Ooops — I’m betting that you don’t have past addresses, or at the very least, aren’t storing the date that the address changed. In short, you never know what a user of your system is going to ask for — try and make sure you have it. Just keep in mind that you don’t want to be moving unnecessary amounts of data up and down your network wire (see my comments on choosing a datatype). If you’re storing the data just for posterity, then make sure you don’t put it in any of your application’s `SELECT` statements if it isn’t needed (actually, this should be your policy regardless of why you’re storing the data).

If you think that there may be legal ramifications either way (both in keeping it and in getting rid of it), consult your attorney. Sometimes you’re legally obligated to keep data a certain amount of time; other times it’s best to get rid of information as soon as legally possible.

Drawing Up a Quick Example

Let’s walk quickly through a process of designing the invoicing database that we’ve already started with during our section on normalization. For the most part, we’re going to just be applying the diagramming tools to what we’ve already designed, but we’ll also toss in a few new issues to show how they affect our design.

Creating the Database

Unlike a lot of the third-party diagramming tools out there, the SQL Server diagramming tools will not create the database for you — you have to already have it created in order to get as far as having the diagram available to work with.

We’re not going to be playing with any data to speak of, so just create a small database called *Invoice*. I’ll go ahead and use the dialog in the Management Studio for the sake of this example.

After right-clicking on the *Databases* node of my server and selecting *New Database*, I enter information in for a database called *Invoice* that is set up as 3MB in size.

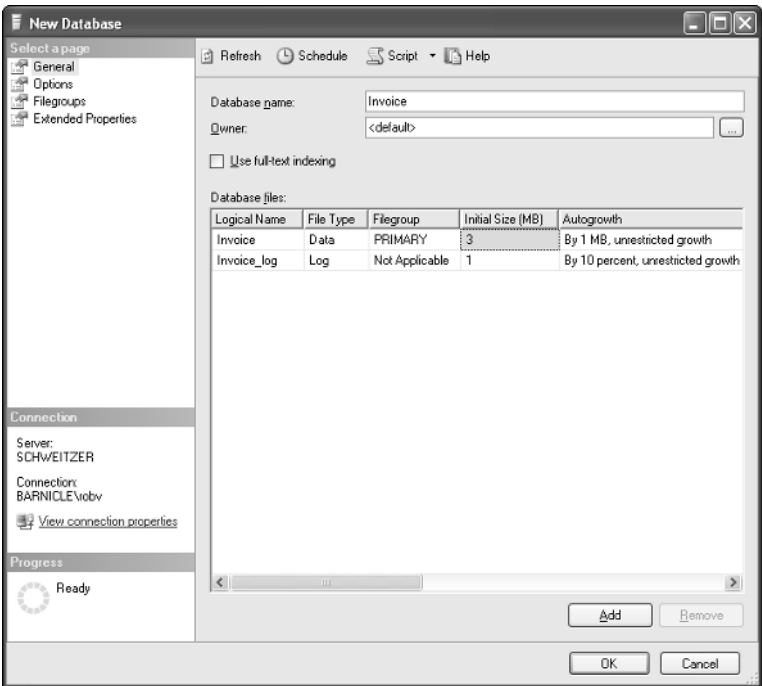


Figure 8-23

Since we’ve already had a chapter on creating databases (and for the sake of brevity), I’m just going to accept the defaults on all the other options, as shown in Figure 8-23.

Adding the Diagram and Our Initial Tables

As we did when creating our Northwind diagram, expand the node for our database (it should have been added underneath the Databases node) and accept the dialog asking if you want to add the objects needed to support diagramming. Then right-click on the Diagrams node and select *New Database Diagram*. The *Add Table* dialog pops up, but since there are no user tables in our database, we’ll just want to click *Cancel* so we wind up with a clean sheet.

Now we’re ready to start adding new tables. You can either click the *New table* icon on the toolbar, or right-click anywhere in the diagram and select *New Table*. Let’s start off by adding in the *Orders* table, as shown in Figure 8-24.

Orders *					
	Column Name	Condensed Type	Nullable	Default Value	Identity
	OrderID	int	No		☒
	OrderDate	datetime	No	GETDATE()	☐
	CustomerNo	int	No		☐
					☐

Figure 8-24

Note that I've changed from the default view — which doesn't have `Default Value` and `Identity` as part of it — over to the “custom” view. I also had to choose to `Modify Custom` and select the `Default Value` and `Identity` columns to be added to my custom view.

Let's stop long enough to look at a couple of the decisions that we made here. While we had addressed the issue of normalization, we hadn't addressed any of the other basics yet. First up of those was the question of datatypes.

Because `OrderID` is the primary key for the table, we need to be sure that we allow enough room for our values to be unique as we insert more and more data. If this was a table we weren't going to be making very many inserts into, we might choose a smaller datatype, but since it is our `Orders` table (and we hope to be entering lots of orders), we'll push the size up a bit. In addition, numeric order numbers seem suitable (make sure you ask your customers about issues like this) and facilitate the use of an automatic numbering mechanism in the form of an identity column. If you need more than 2 billion or so order numbers (in which case, I may want some stock in your company), you can take a look at the larger `BigInt` datatype. (Suffice to say that I'm certain you won't have too many orders for that datatype to hold — just keep in mind the extra space used, although that's often trivial in the larger schema of how much space the database as a whole is using.)

With `OrderDate`, the first thing to come to mind was a `smalldatetime` field. After all, we don't need the kind of precision that a `datetime` field offers, and we also don't need to go back all that far in history (maybe a few years at most). Why, then, did we go for `datetime` rather than `smalldatetime`? To show mercy on Visual Basic programmers! Visual Basic prior to .NET throws fits when you start playing around with `smalldatetime` fields. You can get around the problems, but it's a pain. We used a `datetime` column for nothing more than making the client coding easy.

Our customer has told us (and we've seen in the earlier sample data), that `CustomerNo` is five digits, all numeric. This is one of those areas where you start saying to your customer, “Are you sure you're never going to be using alpha characters in there?” Assuming the answer is yes, we can go with an integer since it is:

- ❑ Faster on lookups.
- ❑ Smaller in size — 4 bytes will cover a 5-digit number easily, but it takes 5 bytes minimum (6 if you're using variable-length fields) to handle 5 characters.

Note that we're kind of cheating on this one — realistically, the customer number for this table is really being defined by the relationship we're going to be building with the `Customers` table. Since that's the last table we'll see in this example, we're going ahead and filling in the blanks for this field now.

After datatypes, we also had to decide on the size of the column — this was a no-brainer for this particular table since all the datatypes have fixed sizes.

Next on the hit list is whether the rows can be null or not. In this case, we're sure that we want all this information and that it should be available at the time we enter the order, so we won't allow nulls.

I’ve touched on this before, but you just about have to drag me kicking and screaming in order to get me to allow nulls in my databases. There are situations where you just can’t avoid it — “undefined” values are legitimate. I’ll still often fill text fields with actual text saying “Value Unknown” or something like that.

The reason I do this is because nullable fields promote errors in much the same way that undeclared variables do. Whenever you run across null values in the table you wind up asking yourself, “Gee, did I mean for that to be there, or did I forget to write a value into the table for that row” — that is, do I have a bug in my program?

The next issue we faced was default values. We couldn’t have a default for `OrderID` because we’re making it an identity column (the two are mutually exclusive). For `OrderDate`, however, a default made some level of sense. If an `OrderDate` isn’t provided, then we’re going to assume that the order date is today. Last, but not least, is the `CustomerNo` — which customer would we default to? Nope — can’t do that here.

Next up was the issue of an identity column. `OrderID` is an ideal candidate for an identity column — the value has no meaning other than keeping the rows unique. Using a counter such as an identity field gives us a nice, presentable, and orderly way to maintain that unique value. We don’t have any reason to change the identity seed and increment, so we won’t. We’ll leave it starting at one and incrementing by one.

Now we’re ready to move on to our next table — the `OrderDetails` table as defined in Figure 8-25.

OrderDetails *			
	Column Name	Data Type	Allow Nulls
	OrderID	int	☐
	LineItem	int	☐
	PartNo	char(6)	☐
	Qty	int	☐
	UnitPrice	money	☐
			☐

Figure 8-25

For this table, the `OrderID` column is going to have a foreign key to it, so our datatype is decided for us — it must be of the same type and size as the field it’s referencing, so it’s going to be an `int`.

The `LineItem` is going to start over again with each row, so we probably could have gotten as little as a `tinyint` here. We’re going to go with an `int` on this one just for safety’s sake. (I’ve had people exceed limits that have been set on this sort of thing before.)

`PartNo` is, for this table, actually going to be defined by the fact that it needs to match up with the `PartNo` in the `Products` table. It’s going to be using a `char(6)` in that table (we’ll come to it shortly), so that’s what we’ll make it here.

`Qty` is guesswork. The question is, what’s the largest order you can take as far as quantity for one line-item oes? Since we don’t know what we’re selling, we can’t really make a guess on a maximum quantity (for example, if we were selling barrels of oil, it might be bought literally millions of barrels at a time). We’re also using an `int` here, but we would have needed a datatype that accepted decimals if we were selling things like gallons of fuel or things by weight.

`UnitPrice` is easy: As this field is going to hold a monetary value, its datatype must be `money`.

Moving along, we're again (no surprise here) considering all data fields to be required. No, we're not allowing nulls anywhere.

No defaults seem to make sense for this table, so we're skipping that part also.

Identity? The temptation might be to mark `OrderID` as an identity column again. Don't do that! Remember that `OrderID` is a value that we're going to match to a column in another table. That table will already have a value (as it happens, set by identity, but it didn't necessarily have to be that way), so setting our column to identity would cause a collision. We would be told that we can't do our insert because we're trying to set an identity value. All the other columns either get their data from another table or require user input of the data. `IsRowGuid` does not apply again.

That takes us to our `Products` and `Customers` tables, as shown in figures 8-26 and 8-27 respectively.

Products *			
	Column Name	Data Type	Allow Nulls
🔑	PartNo	char(6)	☐
	Description	varchar(15)	☐
	Weight	tinyint	☐
			☐

Figure 8-26

Customers *			
	Column Name	Data Type	Allow Nulls
🔑	CustomerNo	int	☐
	CustomerName	varchar(50)	☐
	CustomerAddress	varchar(50)	☐
			☐

Figure 8-27

Let's hit the highlights on the choices here and move on.

`PartNo` has been defined by the data that we saw when we were looking at normalization. It's a numeric, followed by an alpha, followed by four numerics. That's six characters, and it seems to be fixed. We would want to hold the customer to the cross about the notion that the size of the part number can't get any larger but, assuming that's OK, we'll go with a `char(6)` here. That's because a `char` takes up slightly less overhead than a `varchar`, and we know that the length is going to always remain the same (this means that there's no benefit from the variable size).

`Description` is one of those guessing games. Sometimes a field like this is going to be driven by your user interface requirements (don't make it wider than can be displayed on the screen), other times you're just going to be truly guessing at what is "enough" space. We're using a variable-length `char` over a regular `char` for two reasons:

- ❑ To save a little space
- ❑ So we don't have to deal with trailing spaces (look at the `char` vs. `varchar` datatypes back in Chapter 2 if you have questions on this)

We haven't used an `nchar` or `nvarchar` because this is a simple invoicing system for a U.S. business, and we're not concerned about localization issues.

`Weight` is similar to `Description` in that it is going to be somewhat of a guess. We've chosen a `tinyint` here because our products will not be over 255 pounds. Note that we are also preventing ourselves from keeping decimal places in my weight (integers only).

We described the `CustomerNo` field back when we were doing the `Orders` table.

`CustomerName` and `CustomerAddress` are pretty much the same situation as `Description` — the question is, how much is enough? But we need to be sure that we don't give too much

As before, all fields are required (there will be no nulls in either table) and no defaults are called for. Identity columns also do not seem to fit the bill here as both the customer number and part number have special formats that do not lend themselves to the automatic numbering system that an identity provides.

Adding the Relationships

OK, to make the diagram less complicated, I've gone through all four of my tables and changed the view on them down to just `Column Names`. You can do this, too, by simply right-clicking on the table and selecting the `Column Names` menu choice.

You should get a diagram that looks close to Figure 8-28.



Figure 8-28

You may not have the exact same positions for your table, but the contents should be the same. We're now ready to start adding relationships, but we probably ought to stop and think about what kind of relationships we need.

All the relationships that we'll draw with the relationship lines in our SQL Server diagram tool are going to be one-to-zero, one, or many relationships. SQL Server doesn't really know how to do any other kind of relationship implicitly. As we discussed earlier in the chapter, you can add things such as unique constraints and triggers to augment what SQL Server will do naturally with relations, but, assuming you don't do any of that, you're going to wind up with a one-to-zero, one, or many relationship.

The bright side is that this is by far the most common kind of relationship out there. In short, don't sweat it that SQL Server doesn't cover every base here. The standard foreign key constraint (which is essentially what our reference line represents) fits the bill for most things that you need to do, and the rest can usually be simulated via some other means.

We're going to start with the central table in our system — the Orders table. First, we'll look at any relationships that it may need. In this case, we have one — it needs to reference the Customers table. This is going to be a one-to-many relationship with Customers as the parent (the one) and Orders as the child (the many) table.

To build the relationship (and a foreign key constraint to serve as the foundation for that relationship), we're going to simply click and hold in the leftmost column of the Customers table (in the gray area) right where the CustomerNo column is. We'll then drag to the same position (the gray area) next to the CustomerNo column in the Orders table and let go of the mouse button. SQL Server promptly pops up with the first of two dialogs to confirm the configuration of this relationship. The first, shown in Figure 8-29, confirms which columns actually relate.

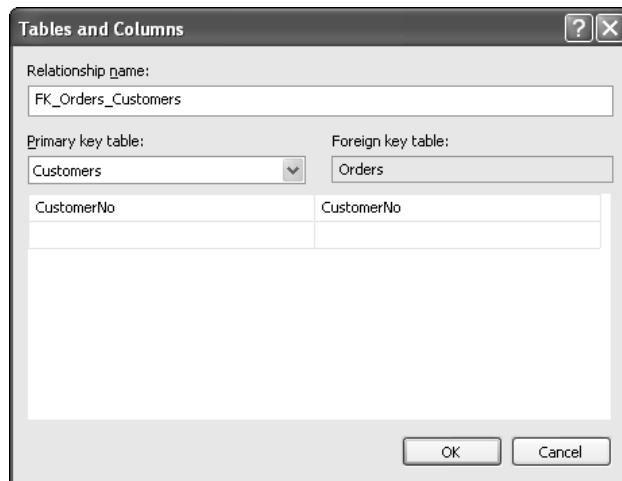


Figure 8-29

As I pointed out earlier in the chapter, don't sweat it if the names that come up don't match with what you intended — just use the combo boxes to change them back so both sides have CustomerNo in them. Note also that the names don't have to be the same — keeping them the same just helps ease confusion in situations where they really are the same.

Click OK, for this dialog, and then also click OK to accept the defaults of the Foreign Key Relationship dialog. As soon as we click OK on the second dialog, we have our first relationship in our new database, as in Figure 8-30.

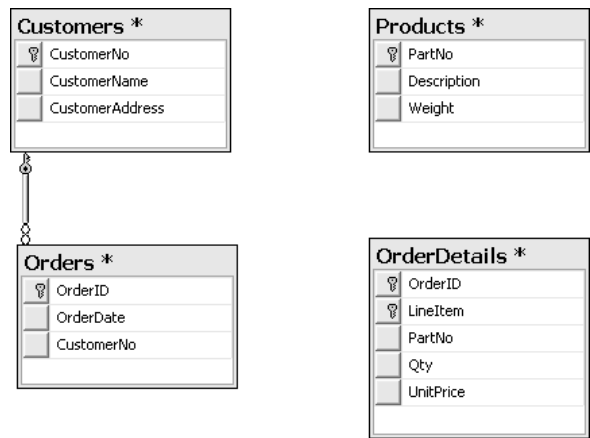


Figure 8-30

Now we'll just do the same thing for our other two relationships. We need to establish a one-to-many relationship from Orders to OrderDetails (there will be one order header for one or more order details) based on OrderID. Also, we need a similar relationship going from Products to OrderDetails (there will be one Products record for many OrderDetails records) based on ProductID as shown in Figure 8-31.

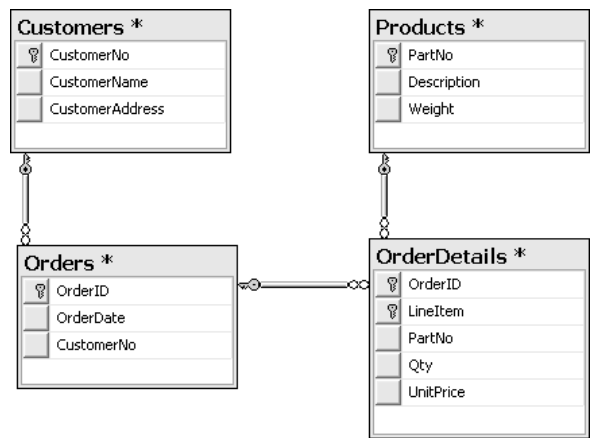


Figure 8-31

Adding Some Constraints

As we were going through the building of our tables and relationships, I mentioned a requirement that we still haven't addressed. This requirement needs a constraint to enforce it: the part number is formatted as 9A9999 where "9" indicates a numeric digit 0–9 and "A" indicates an alpha (non-numeric) character.

Let's add that requirement now by right-clicking on the `Products` table and selecting Check Constraints to bring up the dialog shown in Figure 8-32.

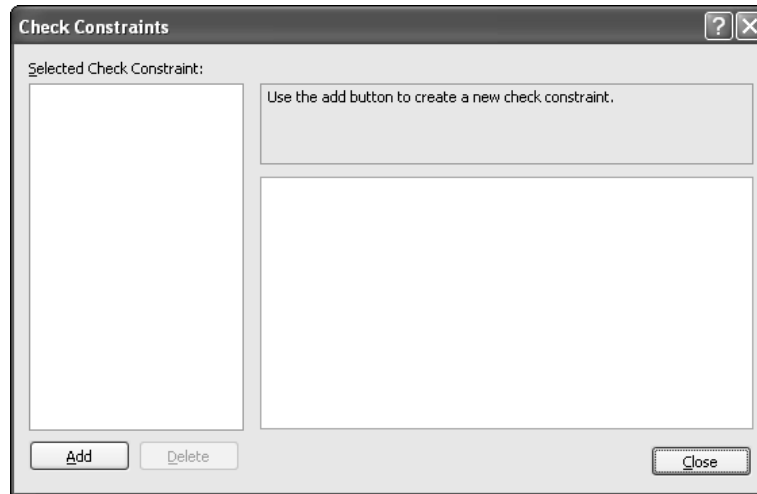


Figure 8-32

It is at this point that we are ready to click Add and define our constraint. In order to restrict part numbers entered to the format we've established, we're going to need to make use of the `LIKE` operator:

```
(PartNo LIKE ' [0-9] [A-Z] [0-9] [0-9] [0-9] [0-9] ')
```

This will essentially evaluate each character that the user is trying to enter in the `PartNo` column of our table. The first character will have to be 0 through 9, the second A through Z (an alpha), and the next four will again have to be numeric digits (the 0 through 9 thing again). We just enter this into the text box labeled `Expression`. In addition, we're going to change the default name for our constraint from `CK_Products` to `CK_PartNo`, as shown in Figure 8-33.

That didn't take us too long — and we now have our first database that we designed!!!

This was, of course, a relatively simple model — but we've now done the things that make up perhaps 90 percent or more of the actual data architecture.

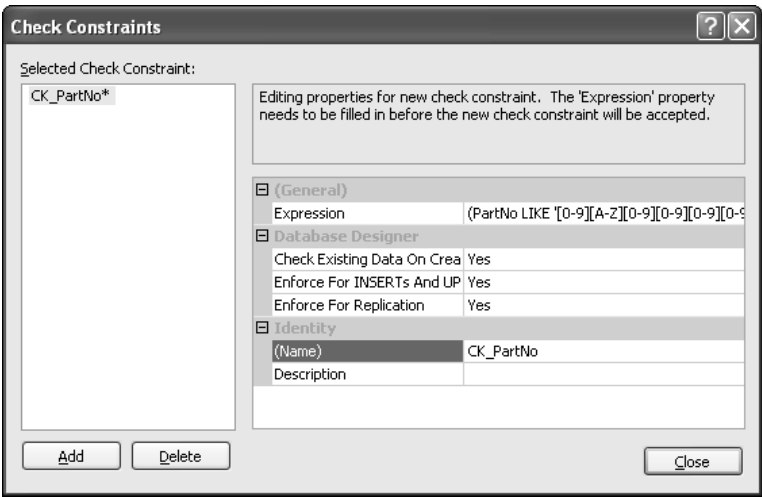


Figure 8-33

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, we have, however, gotten you off to a solid start. We’ve seen that data is considered normalized when we 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.” We’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, we’ve looked at some non-normalization related concepts in our database design, plus how to make use of the daVinci tools to design our database.

In our next chapter, we will be taking a very close look at how SQL Server stores information and how to make the best use of indexes.

Exercises

1. Normalize the following data into 3rd normal form:

Patient	SSN	Physician	Hospital	Treatment	AdmitDate	ReleaseDate
Sam Spade	555-55-5555	Albert Schweitzer	Mayo Clinic	Lobotomy	10/01/2005	11/07/2005
Sally Nally	333-33-3333	Albert Schweitzer	NULL	Cortizone Injection	10/10/2005	10/10/2005
Peter Piper	222-22-2222	Mo Betta	Mustard Clinic	Pickle Extraction	11/07/2005	11/07/2005
Nicki Doohickey	123-45-6789	Sheeze Sheila	Mustard Clinic	Cortizone Injection	11/07/2005	11/07/2005

9

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, you 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.

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. Let's take a look at them one by one.

The Database

OK — 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.

A lock is something of both a hold and a place marker that is used by the system. As you do development using SQL Server — or any other database for that matter — you will find that understanding and managing locks is absolutely critical to your system.

We will be looking into locking extensively in Chapter 14, but we will see the lockability of objects within SQL Server discussed in passing as we look at storage.

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.

There are a number of different *page types*. For purposes of this book, the types we care about are:

- ❑ **Data:** *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 or *varchar(max)*.
- ❑ **Index:** *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.

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 can be up to 8KB.

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 8060 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.

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 our 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 — 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 you 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.

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 9-1.

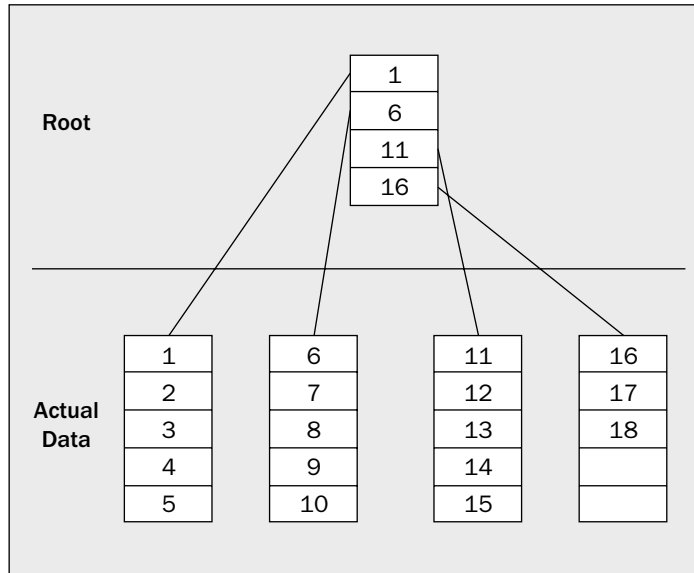


Figure 9-1

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 9-2, 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.

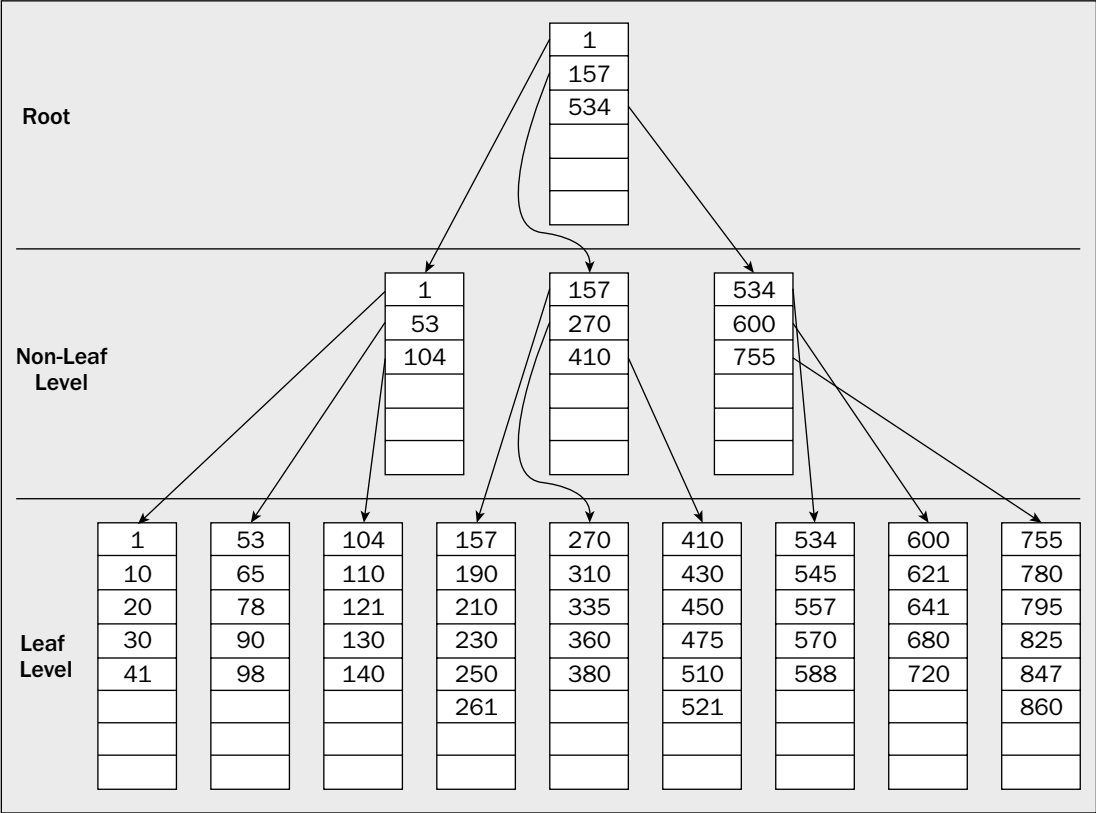


Figure 9-2

Page Splits — A First 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 9-3.

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.

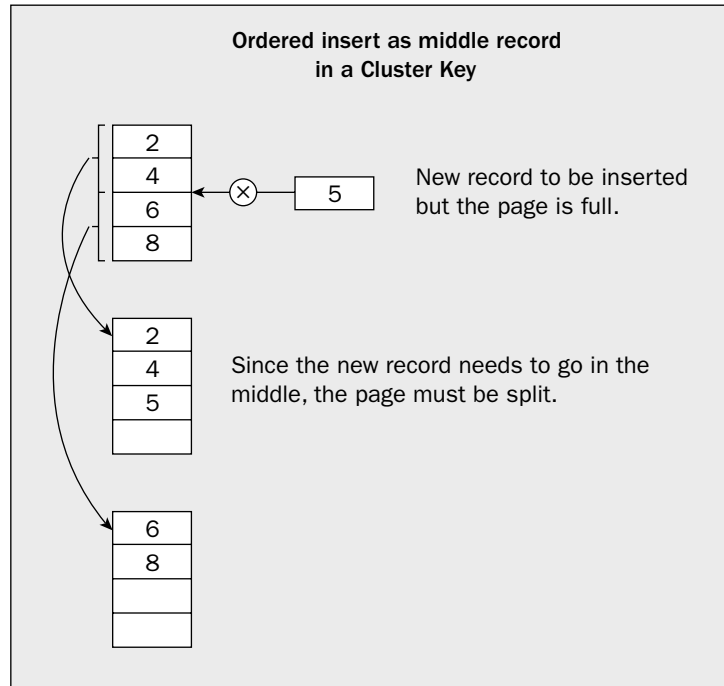


Figure 9-3

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 we'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 re-written).

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 index (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 makeup). 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 we've studied thus far (Chapter 7 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, we don't have to limit ourselves to Boolean situations (does the piece of data I was after exist—yes or no?). We can apply this same notion to both the beginning and end of a range—we are able to gather ranges of data with essentially the same benefits that using an index gives to finding data. What's more, we can do a very fast lookup (called a `SEEK`) of our data rather than hunting through the entire table.

Don't get the impression from my comparing what indexes do for us 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 we mentioned when we 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).

Clustered Tables

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.

Heaps

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 only necessary if there is no cluster key available (no clustered index).

Clustered Indexes

A *clustered index* is unique for any given table — you can only have 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 we look at our 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 re-sorted 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 9-4.

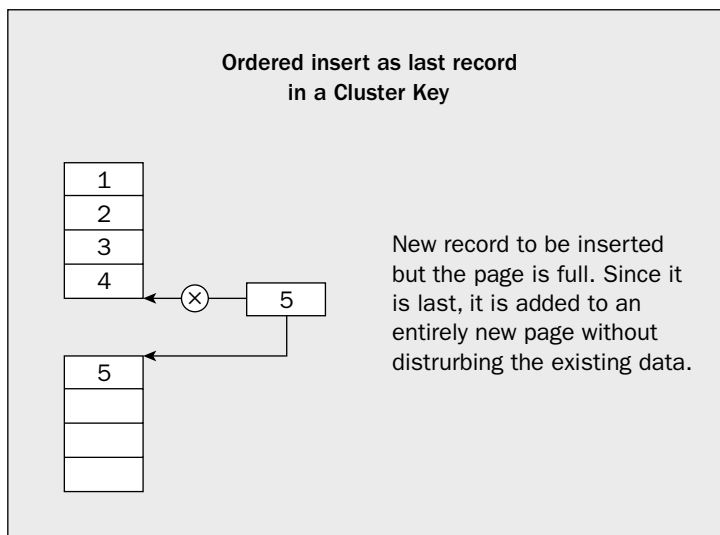


Figure 9-4

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. Let's take a look at a visualization of what a B-Tree looks like for a clustered index (Figure 9-5).

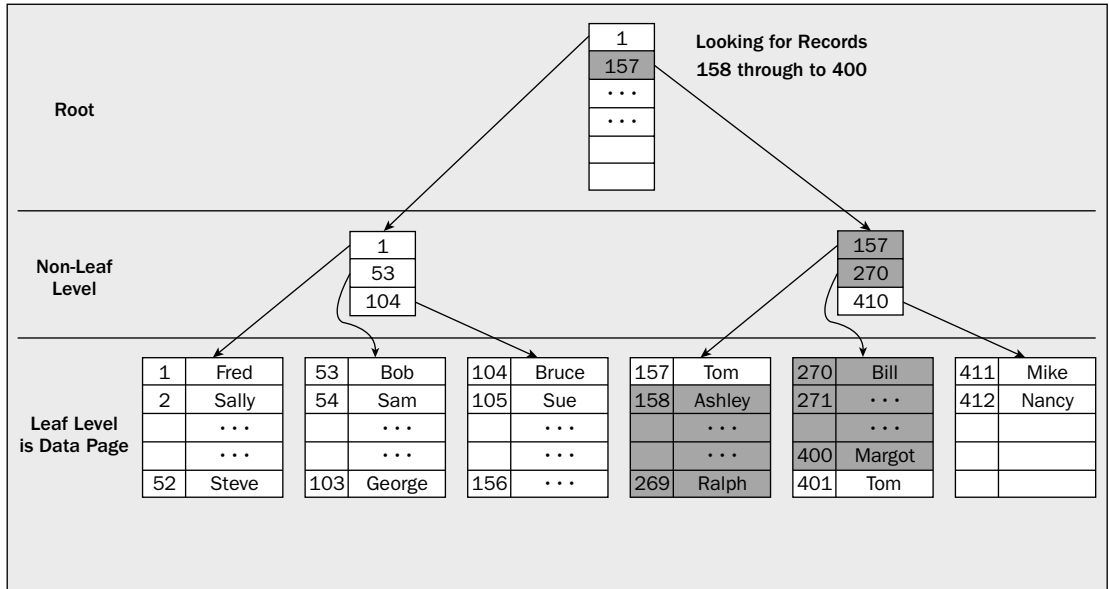


Figure 9-5

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 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 the node one level 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.

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 we 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 we 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), we only have one more step than with a clustered index — because the RID has the full information on the location of the row, we can go directly to the data.

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 vs. the clustered index, the odds are extremely high that any page that was already read once will still be in the memory cache, and, as such, will be retrieved extremely quickly. Still, it does add some additional logical operations to retrieve the data.

Figure 9-6 shows the same search we did with the clustered index, only with a non-clustered index on a heap this time.

Through most of the index navigation, things work exactly as they did before. We start out at the same root node, and we traverse the tree dealing with more and more focused pages until we get to the leaf level of our index. This is where we run into the difference. With a clustered index, we could have stopped right here, but, with a non-clustered index, we have more work to do. If the non-clustered index is on a heap, then we have just one more level to go. We take the Row ID from the leaf level page, and navigate to it — it is not until that point that we are at our actual data.

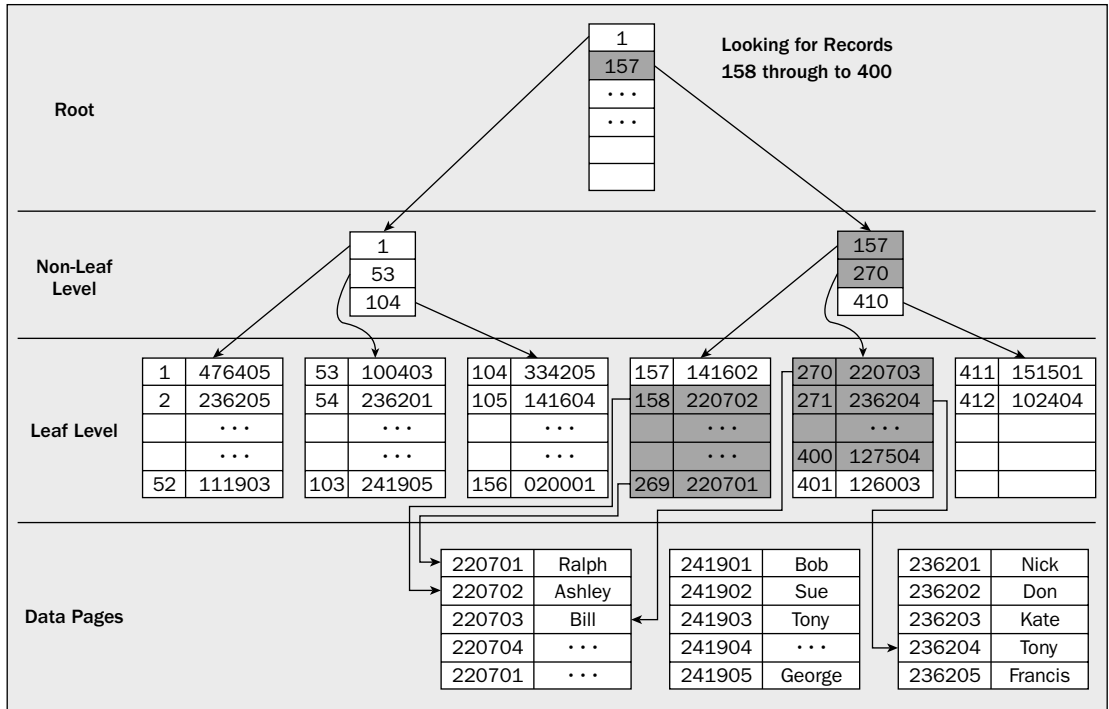


Figure 9-6

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 we get to the leaf level.

At the leaf level, we have a rather sharp difference from what we've seen with the other two index structures—we have yet another index to look over. With clustered indexes, when we got to the leaf level, we found the actual data. With non-clustered indexes on a heap, we didn't find the actual data, but did find an identifier that let us go right to the data (we were just one step away). With non-clustered indexes on a clustered table, we find the cluster-key. That is, we find enough information to go and make use of the clustered index.

We end up with something that looks like Figure 9-7.

What we end up with is two entirely different kinds of lookups.

In the example from our diagram, we start off with a ranged search—we do one single lookup in our index and are able to look through the non-clustered index to find a continuous range of data that meets our criterion (`LIKE 'T%'`). This kind of lookup, where we 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 9-8.

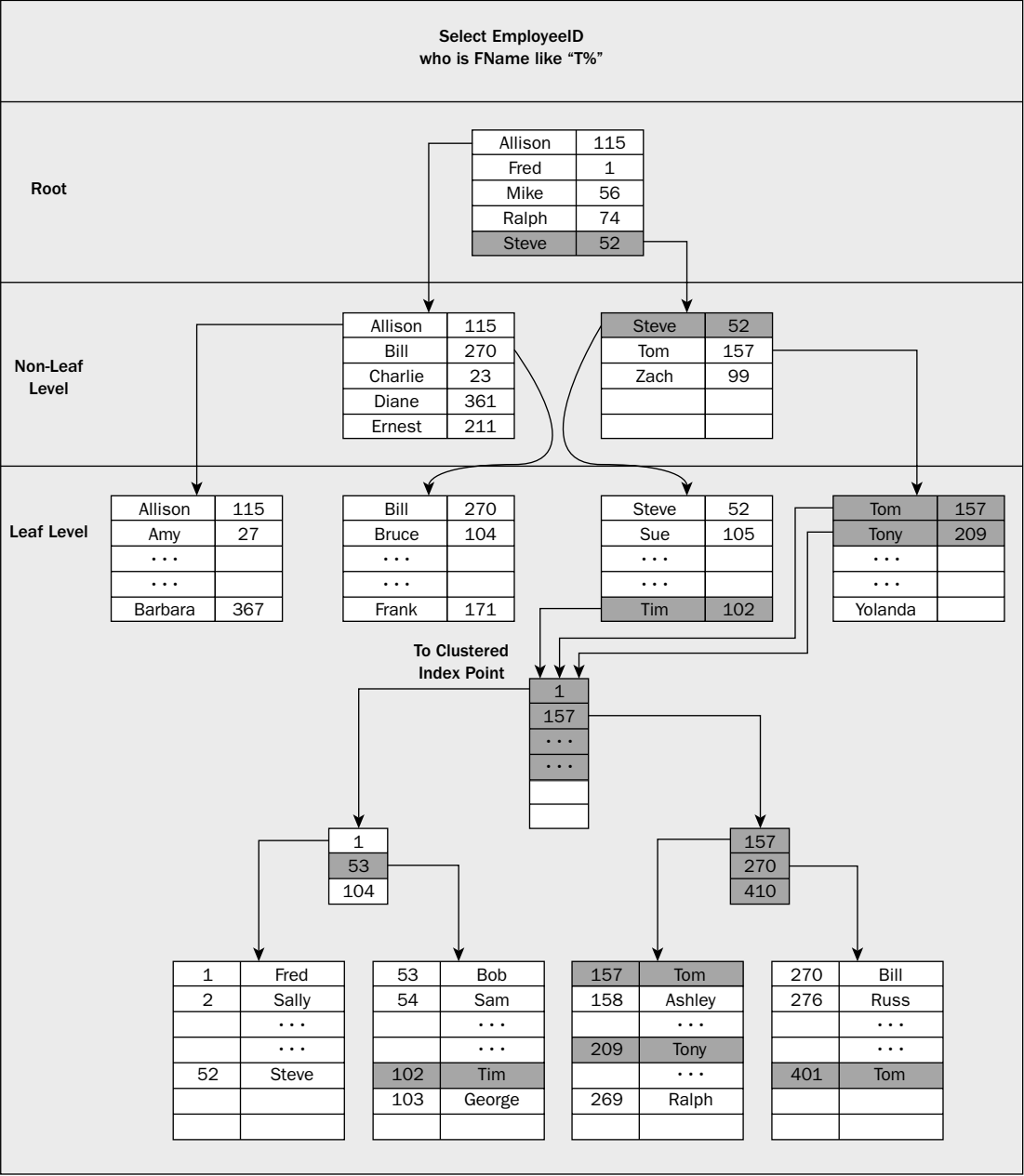


Figure 9-7

Needless to say, this multiple lookup situation introduces more overhead than if we had just been able to use the clustered index from the beginning. The first index search — the one through our 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 our drawing (say, something that would return 5 or 6 rows); it would only take something to the order of 8-10 logical reads to get the information from the non-clustered index. However, that only gets me 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 vs. 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. We'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. Let's 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 let's 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 we 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 we've seen plenty of already (and will see even more of). The primary hitch in things is that we have a few intervening parameters that we haven't seen elsewhere.

Just as we'll see with views in our next chapter, we do have to add an extra clause onto our `CREATE` statement to deal with the fact that an index isn't really a stand-alone kind of object. It has to go together with a table or view, and we need to state the table that our 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 let's 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 vs. 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

Chapter 9

name (an ascending order). In previous versions, SQL Server would have to do two operations—one for the first column and one for the second. By allowing us to control the physical sort order of our data, we gain flexibility in the way we 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*.

When you INCLUDE columns as opposed to placing them in the ON list, SQL Server only adds them 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 only applies 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 we'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 your 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 FILLFACTOR option that follows. Setting PAD_INDEX = ON would be meaningless without a FILLFACTOR (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 20).

We use a `FILLFACTOR` when we 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 8000 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 only be 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 only makes sense 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 vs. 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 how much we've introduced so far on locking, let's stick with a pretty simple explanation.

Through much of the book thus far we 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 we're looking at here are setting directives regarding whether this index will allow those styles of locks or not. This falls under the headed 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 we'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.

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 accomplish, but few have done with any real success. Kudos to the SQL Server team for pulling this one off. Enough gushing though — let's get down to the business of what XML indexes are all about.

This is another of those “chicken or egg?” things, in that we 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 we saw in the previous look at the CREATE statement with the exception of `IGNORE_DUP_KEY` and `ONLINE`. So, for a bit of hyper fast background:

Unlike the relational data that we'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 and 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 only be created 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 together 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 (aka, an *alternate key*)

We’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.

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 our 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 the 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 have 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
)
```

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 for 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:

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, we 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 and move around any of the old data. Now we'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 we'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 in this example, since an insert into a non-clustered index on a heap is usually faster than one on a cluster key.

Even as 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 only considered for use 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 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 Wizard

It would be my hope that you'll learn enough about indexes not to need the *Index Tuning Wizard*, 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 19), and looking over that information for what indexes will work best on your system.

The Index Tuning Wizard 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. Like most any tuning tool, 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, 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 be meaning 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 trouble-shooting 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.

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 ten rows it requires, SQL Server may have to load ten 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 vs. Likelihood of Page Splits

SQL Server gives us a command to help us identify just how full the pages and extents in our database are. We can then use that information to make some decisions about what we want to do to maintain our database. The command is actually an option for the *Database Consistency Checker* — or *DBCC*.

The syntax is pretty simple:

```
DBCC SHOWCONTIG
    [( (<table name>|<table id>|<view name>|<view id>)
      [, <index name>|<index id>])]
    [WITH {ALL_INDEXES|FAST [, ALL_INDEXES ]|TABLERESULTS [, ALL_INDEXES]}]
    [ , { FAST | ALL_LEVELS } ]
DBCC SHOWCONTIG ([<table object id>], [<index id>])
```

As an example, to get the information from the `PK_Order_Details` index in the `Order Details` table, we could run:

```
USE Northwind
GO

DBCC SHOWCONTIG (@id, @IdxID)
GO
```

The output is not really all that self-describing:

```
DBCC SHOWCONTIG scanning 'Order Details' table...
Table: 'Order Details' (325576198); index ID: 1, database ID: 6
TABLE level scan performed.
- Pages Scanned.....: 9
- Extents Scanned.....: 6
- Extent Switches.....: 5
- Avg. Pages per Extent.....: 1.5
- Scan Density [Best Count:Actual Count].....: 33.33% [2:6]
- Logical Scan Fragmentation .....: 0.00%
- Extent Scan Fragmentation .....: 16.67%
- Avg. Bytes Free per Page.....: 673.2
```

Chapter 9

- Avg. Page Density (full).....: 91.68%
DBCC execution completed. If DBCC printed error messages, contact your system administrator.

Some of this is probably pretty straightforward, but the following table will walk us 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 we are.
Avg. Pages per Extent	The average number of pages per extent. A fully populated extent would have 8.
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 us whether 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.

Stat	What It Means
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 and FILLFACTOR

As we saw earlier in the chapter, SQL Server gives us an option for controlling just how full our leaf level pages are, and, if we choose, another option to deal with non-leaf level pages. Unfortunately, these are proactive options — they are applied once, and then you need to re-apply them as necessary by rebuilding your indexes and reapplying the options.

To rebuild indexes, we can either drop them and create them again (if you do, using the `DROP_EXISTING` option usually is a good ideal), or make use of `DBREINDEX`. `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 re-establishes a base percentage that your pages are full. 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 on-going 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 ([Order Details], PK_Order_Details, 65)
```

We can then re-run the `DBCC SHOWCONTIG` to see the effect:

```
DBCC SHOWCONTIG scanning 'Order Details' table...
Table: 'Order Details' (325576198); index ID: 1, database ID: 6
TABLE level scan performed.
- Pages Scanned.....: 13
- Extents Scanned.....: 2
- Extent Switches.....: 1
- Avg. Pages per Extent.....: 6.5
- Scan Density [Best Count:Actual Count].....: 100.00% [2:2]
- Logical Scan Fragmentation .....: 0.00%
- Extent Scan Fragmentation .....: 50.00%
- Avg. Bytes Free per Page.....: 2957.2
- Avg. Page Density (full).....: 63.46%
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.

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).
- ☐ 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?

Exercises

1. Name at least two ways of determining what indexes can be found on the HumanResources.Employee table in the AdventureWorks database.
2. Create a non-clustered index on the ModifiedDate column of the Production.ProductModel table in the AdventureWorks database
3. Delete the index you created in Exercise 2.

10

Views

Up to this point, we've been dealing with base objects — objects that have some level of substance of their own. In this chapter, we're going to go virtual (well, mostly anyway), and take a look at 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

A view is, at its core, really nothing more than a stored query. What's great is that you can mix and match your data from base tables (or other views) to create what will, in most respects, function just like another base table. 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 5, plus a `SELECT` statement like 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:

Chapter 10

```
CREATE VIEW [schema_name].<view name> [(<column name list>)]  
[WITH [ENCRIPTION] [, SCHEMABINDING] [, VIEW_METADATA]]  
AS  
<SELECT statement>  
WITH CHECK OPTION
```

We'll be looking at each piece of this individually, but, for now, let's go ahead and dive right in with an extremely simple view.

Try It Out Creating a Simple View

We'll call this one our customer phone list, and create it as `CustomerPhoneList_vw` in our Accounting database:

```
USE Accounting  
GO  
  
CREATE VIEW CustomerPhoneList_vw  
AS  
    SELECT CustomerName, Contact, Phone  
    FROM Customers
```

Notice that when you execute the `CREATE` statement in the Management Studio, it works just like all the other `CREATE` statements we've done — it doesn't return any rows. It just lets us know that the view has been created:

```
Command(s) completed successfully.
```

Now switch to using the grid view (if you're not already there) to make it easy to see more than one result set. Then run a `SELECT` statement against your view — using it just as you would for a table — and another against the `Customers` table directly:

```
SELECT * FROM CustomerPhoneList_vw  
  
SELECT * FROM Customers
```

How It Works

What you get back looks almost identical — indeed, in the columns that they have in common, the two result sets *are* identical. To clarify how SQL Server is looking at your query on the view, let's break it down logically a bit.

The `SELECT` statement in your view is defined as:

```
SELECT CustomerName, Contact, Phone  
FROM Customers
```

So when you run:

```
SELECT * FROM CustomerPhoneList_vw
```

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. What’s nice about that is that we have reduced the complexity for the end user. In this day and age, where we have so many tools to make life easier for the user, this may not seem like all that big of deal — but to the user, it is.

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 existing 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 go with another view that illustrates what we can do in terms of hiding sensitive data. For this example, let’s go back to our `Employees` table in our `Accounting` database. Take a look at the table layout:

Employees
EmployeeID
FirstName
MiddleInitial
LastName
Title
SSN
Salary
HireDate
TerminationDate
ManagerEmpID
Department

Federal law in the U.S. protects some of this information — we must limit access to a “need to know” basis. Other columns, however, are free for anyone to see. What if we want to expose the unrestricted columns to a group of people, but don’t want them to be able to see the general table structure or data? One solution would be to keep a separate table that includes only the columns that we need:

Employees
EmployeeID
FirstName
MiddleInitial
LastName
Title
HireDate
TerminationDate
ManagerEmpID
Department

While on the surface this would meet our needs, it is extremely problematic:

- ❑ We use disk space twice.
- ❑ We have a synchronization problem if one table gets updated and the other doesn't.
- ❑ We have double I/O operations (you have to read and write the data in two places instead of one) whenever we need to insert, update, or delete rows.

Views provide an easy and relatively elegant solution to this problem. By using a view, the data is stored only once (in the underlying table or tables) — eliminating all of the problems just described. Instead of building our completely separate table, we can just build a view that will function in a nearly identical fashion.

Our `Employees` table is currently empty. To add some rows to it, load the `Chapter10.sql` file (supplied with the source code) into the Management Studio and run it. Then add the following view to the Accounting database:

```
USE Accounting
GO

CREATE VIEW Employees_vw
AS
SELECT    EmployeeID,
          FirstName,
          MiddleInitial,
          LastName,
          Title,
          HireDate,
          TerminationDate,
          ManagerEmpID,
          Department
FROM Employees
```

We are now ready to let everyone have access—directly or indirectly—to the data in the `Employees` table. Users who have the “need to know” can now be directed to the `Employees` table, but we continue to deny access to other users. Instead, the users who do not have that “need to know” can have access to our `Employees_vw` view. If they want to make use of it, they do it just the same as they would against a table:

```
SELECT *  
FROM Employees_vw
```

This actually gets into one of the sticky areas of naming conventions. Because I’ve been using the `_vw` suffix, it’s pretty easy to see that this is a view and not a table. Sometimes, you’d like to make things a little more hidden than that, so you might want to deliberately leave the `_vw` off. Doing so means that you have to use a different name (`Employees` is already the name of the base table), but you’d be surprised how many users won’t know that there’s a difference between a view and a table if you do it this way.

Views as Filters

This will probably be one of the shortest sections in the book. Why? Well, it doesn’t get much simpler than this.

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.

Let’s take our `Employees_vw` view from the last section, and beef it up a bit by making it a list of only current employees. To do this, there are really only two changes that need to be made.

First, we have to filter out employees who no longer work for the company. Would a current employee have a termination date? Probably not, so, if we limit our results to rows with a `NULL TerminationDate`, then we’ve got what we’re after.

The second change illustrates another simple point about views working just like queries—the column(s) contained in the `WHERE` clause do not need to be included in the `SELECT` list. In this case, it doesn’t make any sense to include the termination date in the result set as we’re talking about current employees.

Try It Out Using a View to Filter Data

With these two things in mind, let’s create a new view by changing our old view around just a little bit:

```
CREATE VIEW CurrentEmployees_vw  
AS  
SELECT    EmployeeID,  
          FirstName,  
          MiddleInitial,  
          LastName,  
          Title,  
          HireDate,
```


Chapter 10

```
        ManagerEmpID,  
        Department  
FROM Employees  
WHERE TerminationDate IS NULL
```

In addition to the name change and the `WHERE` clause we've added, note that we've also eliminated the `TerminationDate` column from the `SELECT` list.

Let's test out how this works a little bit by running a straight `SELECT` statement against our `Employees` table and limiting our `SELECT` list to the things that we care about:

```
SELECT  EmployeeID,  
        FirstName,  
        LastName,  
        TerminationDate  
FROM Employees
```

This gets us back a few columns from all the rows in the entire table:

EmployeeID	FirstName	LastName	TerminationDate
1	Joe	Dokey	NULL
2	Peter	Principle	NULL
3	Steve	Smith	1997-01-31 00:00:00
4	Howard	Kilroy	NULL
5	Mary	Contrary	1998-06-15 00:00:00
6	Billy	Bob	NULL

(6 row(s) affected)

Now let's check out our view:

```
SELECT  EmployeeID,  
        FirstName,  
        LastName  
FROM CurrentEmployees_vw
```

Our result set has become a bit smaller:

EmployeeID	FirstName	LastName
1	Joe	Dokey
2	Peter	Principle
4	Howard	Kilroy
6	Billy	Bob

(4 row(s) affected)

A few people are missing versus our first select—just the way we wanted it.

How It Works

As we’ve discussed before, the view really is just a SELECT statement that’s been hidden from the user so that they can ignore what the SELECT statement says, and instead just consider the results it produces just as if it were a table—you can liken this to the derived tables we discussed back in Chapter 7. Because our data was filtered down before we referenced the view by name, our query doesn’t even need to consider that data (the view has done that for us).

More Complex Views

Even though I use the term “complex” here—don’t let that scare you. The toughest thing in views is still, for the most part, simpler than most other things in SQL.

What we’re doing with more complex views is really just adding joins, summarization, and perhaps some column renaming.

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 *Northwind* 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 parts and who placed them. So, we create a view that they can perform very simple queries on—remember that we are creating this one in *Northwind*:

```
USE Northwind
GO

CREATE VIEW CustomerOrders_vw
AS
SELECT    cu.CompanyName,
          o.OrderID,
          o.OrderDate,
          od.ProductID,
          p.ProductName,
          od.Quantity,
          od.UnitPrice,
          od.Quantity * od.UnitPrice AS ExtendedPrice
FROM      Customers AS cu
INNER JOIN Orders AS o
          ON cu.CustomerID = o.CustomerID
INNER JOIN [Order Details] AS od
          ON o.OrderID = od.OrderID
INNER JOIN Products AS p
          ON od.ProductID = p.ProductID
```

Chapter 10

Now do a SELECT:

```
SELECT *
FROM CustomerOrders_vw
```

You wind up with a bunch of rows—over 2,000—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 they are looking for:

```
SELECT CompanyName, ExtendedPrice
FROM CustomerOrders_vw
WHERE OrderDate = '9/3/1996'
```

The user didn’t need to know how to do a four-table join—that was hidden in the view. Instead, they only need limited skill (and limited imagination for that matter) in order to get the job done.

CompanyName	ExtendedPrice
LILA-Supermercado	201.6000
LILA-Supermercado	417.0000
LILA-Supermercado	432.0000

(3 row(s) affected)

However, we could make our query even more targeted. Let’s say that we only want our view to return yesterday’s sales. We’ll make only slight changes to our query:

```
USE Northwind
GO
```

```
CREATE VIEW YesterdaysOrders_vw
AS
SELECT    cu.CompanyName,
          o.OrderID,
          o.OrderDate,
          od.ProductID,
          p.ProductName,
          od.Quantity,
          od.UnitPrice,
          od.Quantity * od.UnitPrice AS ExtendedPrice
FROM      Customers AS cu
INNER JOIN Orders AS o
ON cu.CustomerID = o.CustomerID
INNER JOIN [Order Details] AS od
ON o.OrderID = od.OrderID
INNER JOIN Products 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 Northwind 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 Northwind

DECLARE @Ident int

INSERT INTO Orders
(CustomerID,OrderDate)
VALUES
('ALFKI', DATEADD(day,-1,GETDATE()))

SELECT @Ident = @@IDENTITY

INSERT INTO [Order Details]
(OrderID, ProductID, UnitPrice, Quantity)
VALUES
(@Ident, 1, 50, 25)

SELECT 'The OrderID of the INSERTed row is ' + CONVERT(varchar(8),@Ident)

```

I'll be explaining all of what is going on here in our chapter on scripts and batches. For now, just trust me that you'll need to run all of this in order for us to have a value in *Northwind* 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 11087

(1 row(s) affected)

```

Be aware that some of the messages shown above will only appear on the Messages tab if you are using the Management Studio's Results In Grid mode.

The *OrderID* might vary, but the rest should hold pretty true.

Now let's run a query against our view and see what we get:

```
SELECT CompanyName, OrderID, OrderDate FROM YesterdaysOrders_vw
```

You can see that the 11087 does indeed show up:

CompanyName	OrderID	OrderDate
Alfreds Futterkiste	11087	2000-08-05 17:37:52.520

(1 row(s) affected)

Don't get stuck on the notion that your `OrderID` numbers are going to be the same as mine — these are set by the system (since `OrderID` 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.

The `DATEADD` and `CONVERT` Functions

The join, while larger than most of the ones we've done this far, is still pretty straightforward. We keep adding tables, joining a column in each new table to a matching column in the tables that we've already named. As always, note that the columns do not have to have the same name — they just have to have data that relates to one another.

Since this was a relatively complex join, let's take a look at what we are doing in the query that supports this view.

The `WHERE` clause is where things get interesting:

```
WHERE CONVERT (varchar (12) ,o.OrderDate,101) =  
      CONVERT (varchar (12) ,DATEADD (day,-1,GETDATE()) ,101)
```

It's a single comparison, but we have several functions that are used to come up with our result.

It would be very tempting to just compare the `OrderDate` in the `Orders` table to `GETDATE()` (today's date) minus one day — the subtraction operation is what the `DATEADD` function is all about. `DATEADD` can add (you subtract by using negative numbers) any amount of time you want to deal with. You just tell it what date you want to operate on, what unit of time you want to add to it (days, weeks, years, minutes, and so on). On the surface, you should just be able to grab today's date with `GETDATE()` and then use `DATEADD` to subtract one day. The problem is that `GETDATE()` includes the current time of day, so we would only get back rows from the previous day that happened at the same time of day down to 3.3333 milliseconds — not a likely match. So we took things one more step and used the `CONVERT` function to equalize the dates on both sides of the equation to the same time-of-day-less format before comparison. Therefore, the view will show any sale that happened any time on the previous date.

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. The problem here is the complexity of `INSTEAD OF` triggers and 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 vs. 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 briefly at `INSTEAD OF` triggers in Chapter 15.

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 (more on this in a moment). 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 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.

Try It Out WITH CHECK OPTION

To illustrate `WITH CHECK OPTION`, let’s continue working with the Northwind database, and create a view to show only Oregon shippers. We have only limited fields to work with in our `Shippers` table, so we’re going to have to make use of the `Area Code` in order to figure out where the shipper is from (make sure that you use Northwind):

```
CREATE VIEW OregonShippers_vw
AS
SELECT    ShipperID,
          CompanyName,
          Phone
FROM      Shippers
WHERE     Phone LIKE '(503)%'
          OR Phone LIKE '(541)%'
          OR Phone LIKE '(971)%'
WITH CHECK OPTION
```

Run a `SELECT *` against this view, and, as it happens, you return all the rows in the table (because all the rows in the table meet the criteria):

ShipperID	CompanyName	Phone
1	Speedy Express	(503) 555-9831
2	United Package	(503) 555-3199
3	Federal Shipping	(503) 555-9931
4	Speedy Shippers, Inc.	(503) 555-5566

(4 row(s) affected)

Now try to update one of the rows using the view — set the phone value to have anything other than a value starting with (503), (541) or (971):

```
UPDATE OregonShippers_vw
SET Phone = '(333) 555 9831'
WHERE ShipperID = 1
```

SQL Server promptly tells you that you are a scoundrel and that you should be burned at the stake for your actions — well, not really, but it does make its point.

```
Msg 550, Level 16, State 1, Line 1
The attempted insert or update failed because the target view either specifies WITH CHECK OPTION or spans a view that specifies WITH CHECK OPTION and one or more rows resulting from the operation did not qualify under the CHECK OPTION constraint.
The statement has been terminated.
```

How It Works

Our `WHERE` clause filters things in the view to just (503), (541) or (971) area codes, and the `WITH CHECK OPTION` says any `INSERT` or `UPDATE` statements must meet that where clause criteria (which a (333) area code doesn’t).

Since our update wouldn't meet the `WHERE` clause criteria, it is thrown out; however, if we insert the row right into the base table:

```
UPDATE Shippers
SET Phone = '(333) 555 9831'
WHERE ShipperID = 1
```

SQL Server is a lot friendlier:

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

The restriction applies only to the view — not to the underlying table. This can actually be quite handy in a rare circumstance or two. Imagine a situation where you want to allow some users to insert or update data in a table, but only when the updated or inserted data meets certain criteria. We could easily deal with this restriction by adding a `CHECK` constraint to our underlying table—but this might not always be an ideal solution.

Imagine now that we've added a second requirement—we still want other users to be able to `INSERT` data into the table without meeting these criteria. Uh oh, the `CHECK` constraint will not discriminate between users. By using a view together with a `WITH CHECK OPTION`, we can point the restricted users to the view, and let the unrestricted users make use of the base table or a view that has no such restriction.

Just for confirmation — this works on an `INSERT`, too. Run an `INSERT` that violates the `WHERE` clause:

```
INSERT INTO OregonShippers_vw
VALUES
('My Freight Inc.', '(555) 555-5555')
```

And you see your old friend, the “terminator” error, exactly as before:

```
Msg 550, Level 16, State 1, Line 1
The attempted insert or update failed because the target view either specifies WITH
CHECK OPTION or spans a view that specifies WITH CHECK OPTION and one or more rows
resulting from the operation did not qualify under the CHECK OPTION constraint.
The statement has been terminated.
```

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.

Chapter 10

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 re-establish 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.

Creating and Editing Views in the Management Studio

For people who really don't know what they are doing, this has to be a rather cool feature in the Management Studio. Building views is a snap, and you really don't have to know all that much about queries in order to get it done.

To take a look at this, fire up the Management Studio, open up the Northwind database sub-node of the Databases node and right-click on Views. You should see the window shown in Figure 10-1.

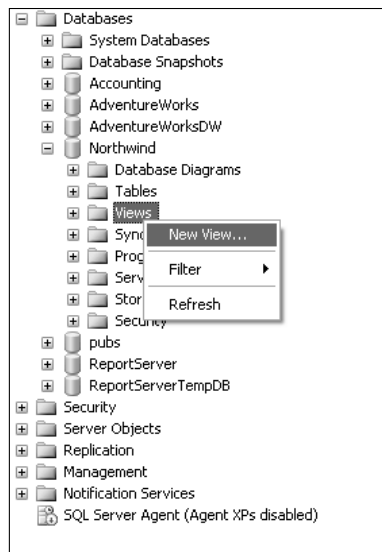


Figure 10-1

Now select `New View`, and up comes a new dialog.

This dialog makes it easy for us to choose what tables we're going to be including data from. Categories is selected in Figure 10-2, but we're going to be working with not only a different table — but *four* other tables.

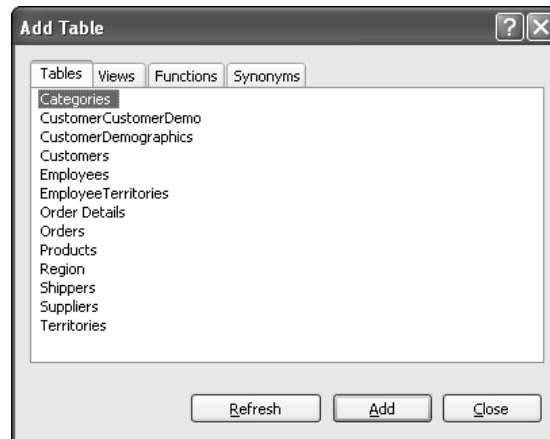


Figure 10-2

This should beg the question: "How do I select more than one table?" Easy — just hold down your control key while selecting all the tables you want. For now, start by clicking on the Customers table, and then press and hold your Control key while you also select Orders, Order Details, and Products. You should wind up with all of them highlighted, as in Figure 10-3.

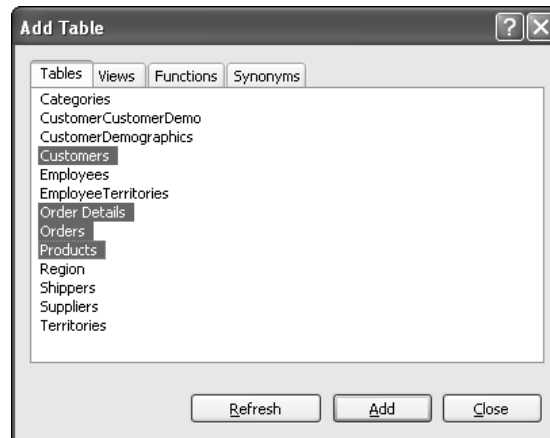


Figure 10-3

Now click Add, and SQL Server will add several tables to our view (indeed, on most systems, you should be able to see them being added to the view editor that we're about to take a look at).

Chapter 10

Before we close the Add Table dialog, take a moment to notice the Views, Functions, and Synonym tabs along the top of this dialog. Because views can reference these objects directly, the dialog gives you a way of adding them directly.

For now, however, just click Add and check out the view editor that is brought up, as in Figure 10-4.

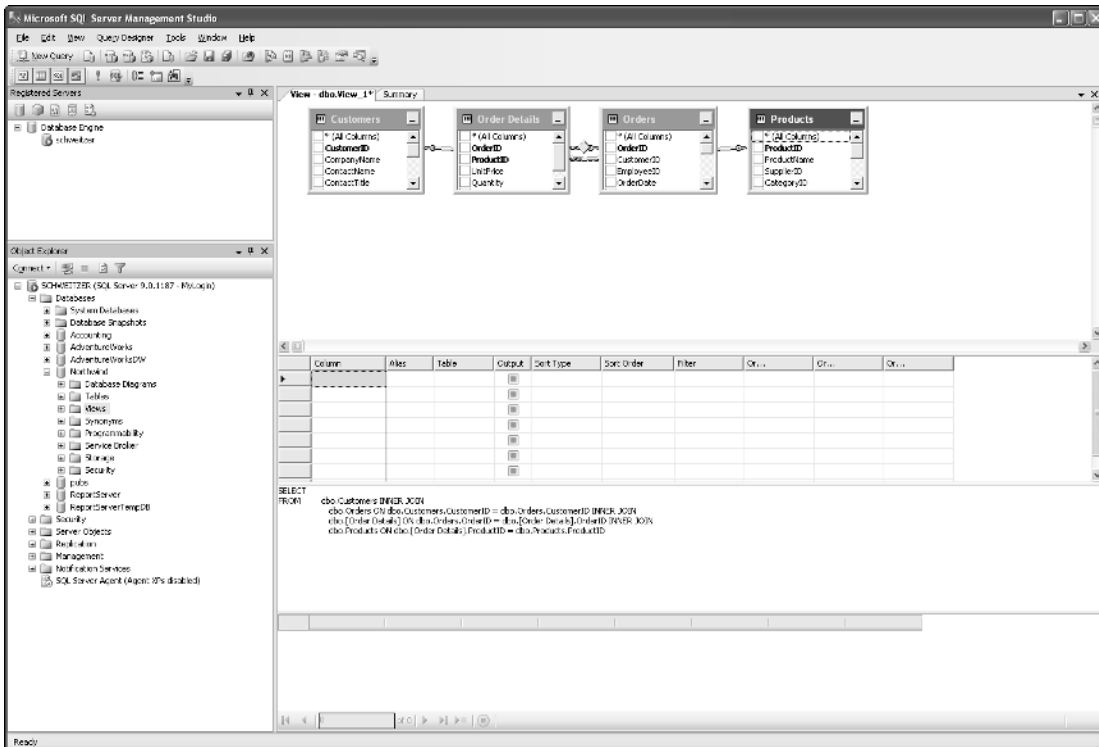


Figure 10-4

There are four panes to the View Builder — each of which can be independently turned on or off:

- ☐ The Diagram pane
- ☐ The Criteria pane
- ☐ The SQL pane
- ☐ The Results pane

For those of you who have worked with Access at all, the Diagram pane works much as it does in Access queries. You can add and remove tables, and even define relationships. Each of those added tables, checked columns, and defined relationships will automatically be reflected in the SQL pane in the form of the SQL required to match the diagram. To identify each of the icons on the toolbar, just hover your mouse pointer over them for a moment or two, and you will get a tooltip that indicates the purpose of each button.

You can add tables either by right-clicking in the Diagram pane (the top one in Figure 10-4) and choosing **Add Table** or by clicking on the **Add table** toolbar button (the one with an arrow pointing right in the very top left of the icon).

Now let's select some columns, as shown in Figure 10-5.



Figure 10-5

Note that I am just showing the diagram pane here to save space. If I have the Grid pane up while you check the above, then you would see each column appear in the Grid pane as you select it. With the SQL pane up, you will also see it appear in the SQL code.

In case you haven't recognized it yet, we're building the same view that we built as our first complex view (CustomerOrders_vw). The only thing that's tricky at all is the computed column (ExtendedPrice). To do that one, either we have to manually type the equation into the SQL pane, or we can type it into the Column column in the Grid pane along with its alias (see Figure 10-6).

Column	Alias	Table	Output	Sort Type	Sort Order	Filter	Or...
OrderDate		Orders	☒				
CompanyName		Customers	☒				
ProductName		Products	☒				
ProductID		[Order Deta...	☒				
UnitPrice		[Order Deta...	☒				
Quantity		[Order Deta...	☒				
dbo.[Order Details].Quantity * dbo.[Order Details].UnitPrice	ExtendedPrice		☒				
SELECT dbo.Orders.OrderDate, dbo.Customers.CompanyName, dbo.Products.ProductName, dbo.[Order Details].ProductID, dbo.[Order Details].UnitPrice, FROM dbo.Customers INNER JOIN dbo.Orders ON dbo.Customers.CustomerID = dbo.Orders.CustomerID INNER JOIN dbo.[Order Details] ON dbo.Orders.OrderID = dbo.[Order Details].OrderID INNER JOIN dbo.Products ON dbo.[Order Details].ProductID = dbo.Products.ProductID							

Figure 10-6

When all is said and done, the View Builder gives us the following SQL code:

```
SELECT dbo.Orders.OrderDate,
       dbo.Customers.CompanyName,
       dbo.Products.ProductName,
       dbo.[Order Details].ProductID,
       dbo.[Order Details].UnitPrice,
       dbo.[Order Details].Quantity,
       dbo.[Order Details].Quantity * dbo.[Order Details].UnitPrice AS
ExtendedPrice
FROM   dbo.Customers INNER JOIN
       dbo.Orders ON dbo.Customers.CustomerID = dbo.Orders.CustomerID INNER JOIN
```

```
        dbo.[Order Details] ON dbo.Orders.OrderID = dbo.[Order Details].OrderID
INNER JOIN
        dbo.Products ON dbo.[Order Details].ProductID = dbo.Products.ProductID
```

While it's not formatted the same, if you look it over, you'll find that it's basically the same code we wrote by hand!

If you've been struggling with learning your T-SQL query syntax, you can use this tool to play around with the syntax of a query. Just drag and drop some tables into the Diagram pane, select the column you want from each table, and, for the most part, SQL Server will build you a query—you can then use the syntax from the view builder to learn how to build it yourself next time.

Now go ahead and save it (the disk icon in the toolbar is how I do it) as `CustomerOrders2_vw` and close the View Builder.

Editing Views in the Management Studio

Modifying our view in the Management Studio is as easy as creating it was. The only real difference is that you need to navigate to your specific view and right-click it—then choose `Modify`, and you'll be greeted with the same friendly query designer that we used with our query when it was created.

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 like 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, as 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 Northwind database—`Alphabetical List of Products`:

```
EXEC sp_helptext [Alphabetical list of products]
```

SQL Server obliges us with the code for the view:

Text

```
-----
create view "Alphabetical list of products" AS
SELECT Products.*, Categories.CategoryName
FROM Categories INNER JOIN Products ON Categories.CategoryID = Products.CategoryID
WHERE (((Products.Discontinued)=0))
```

I must admit to finding this one of the more peculiar examples that Microsoft ever provided (probably part of why Northwind is not their default sample any more — too bad since it is, for most of it, a better teaching database than AdventureWorks is) — I attribute it to this database being migrated from Access. Why? Well, the one seemingly simple thing that we *cannot* do in views is use an `ORDER BY` clause. The exception to the rule on the `ORDER BY` clause is that you can use `ORDER BY` as long as you also use the `TOP` predicate. Microsoft says this view is in alphabetical order, but there is no guarantee of that. Indeed, if you run the query, odds are it won't come out in alphabetical order!

Note that the restriction on using the `ORDER BY` clause applies only to the code within the view. Once the view is created, you can still use an `ORDER BY` clause when you reference the view in a query.

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 syscomments sc
JOIN sysobjects so
  ON sc.id = so.id
WHERE so.name = 'Alphabetical list of products'
```

Again, you get the same block of code (indeed, all `sp_helptext` does is run what amounts to this same query):

Text

```
-----
create view "Alphabetical list of products" AS
SELECT Products.*, Categories.CategoryName
```

```
FROM Categories INNER JOIN Products ON Categories.CategoryID = Products.CategoryID
WHERE (((Products.Discontinued)=0))
(1 row(s) affected)
```

I can't stress enough my recommendation that you avoid the system tables where possible—but I do like you to know your options.

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.

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 our `CustomerOrders_vw` view that we created in Northwind. If you haven't yet created the `CustomerOrders_vw` view, then just change the `ALTER` to `CREATE` (don't forget to run this against Northwind):

```
ALTER VIEW CustomerOrders_vw
WITH ENCRYPTION
AS
SELECT    cu.CompanyName,
          o.OrderDate,
          od.ProductID,
          p.ProductName,
          od.Quantity,
          od.UnitPrice,
          od.Quantity * od.UnitPrice AS ExtendedPrice
FROM      Customers AS cu
INNER JOIN Orders AS o
          ON cu.CustomerID = o.CustomerID
INNER JOIN [Order Details] AS od
          ON o.OrderID = od.OrderID
INNER JOIN Products 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 object comments have been encrypted.

The heck you say, and promptly go to the `syscomments` table:

```
SELECT sc.text FROM syscomments sc
JOIN sysobjects so
  ON sc.id = so.id
WHERE so.name = 'CustomerOrders_vw'
```

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 the 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 re-writing 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 updateable. 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 only supported in the Enterprise Edition (OK, the Developer and Evaluation Editions also supported it, too, 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 us 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), 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 making a few alterations to the `CustomerOrders_vw` object that we created earlier in the chapter:

```
ALTER VIEW CustomerOrders_vw
WITH SCHEMABINDING
AS
SELECT    cu.CompanyName,
          o.OrderID,
          o.OrderDate,
          od.ProductID,
          p.ProductName,
          od.Quantity,
          od.UnitPrice
FROM      dbo.Customers AS cu
INNER JOIN  dbo.Orders AS o
          ON cu.CustomerID = o.CustomerID
INNER JOIN  dbo.[Order Details] AS od
          ON o.OrderID = od.OrderID
INNER JOIN  dbo.Products AS p
          ON od.ProductID = p.ProductID
```

The big things to notice here are:

- ❑ We had to make our view use the `SCHEMABINDING` option.
- ❑ In order to utilize the `SCHEMABINDING` option, we had to go to two-part naming for the objects (in this case, all tables) that we reference.

We had to remove our calculated column—while you can build indexed views with non-aggregate expressions, the query optimizer will ignore them.

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 (CompanyName, OrderID, ProductID)
```

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, everything appears to be fine—but try displaying the graphical showplan as shown in Figure 10-7 (Display Estimated Execution Plan is the tooltip for this, and you'll find it toward the center of the toolbar).

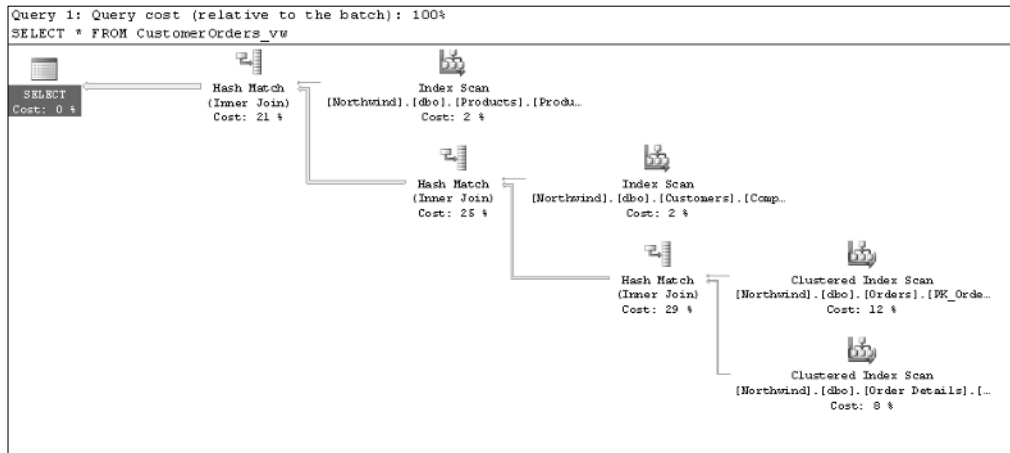


Figure 10-7

I mentioned a paragraph or two ago that we had a small problem—the evidence is in this showplan. If you look through all the parts of this, you’ll see that our index isn’t being used at all!

At issue here is the size of our tables. The *Northwind* database doesn’t have enough data. You see, the optimizer runs a balance between how long it will take to run the first plan that it finds versus the amount of work it takes to keep looking for a better plan. For example, does it make sense to spend two more seconds thinking about the plan when the plan you already know about could be done in less than one second?

In our example above, SQL Server looks at the underlying table, sees that there really isn’t that much data out there, and decides that the plan it has is “good enough” before the optimizer gets far enough to see that the index on the view might be faster.

Keep this issue of “just how much data is there” versus “what will it cost to keep looking for a better plan” in mind when deciding on any index—not just indexed views. For small datasets, there’s a very high possibility that SQL Server will totally ignore your index in favor of the first plan that it comes upon. In such a case, you pay the cost of maintaining the index (slower INSERT, UPDATE, and DELETE executions) without any benefit in the SELECT.

Just so we get a chance to see a difference, however, let’s create a database that will have enough data to make our index more interesting. You can download and execute a population script called `CreateAndLoadNorthwindBulk.sql`.

*Figure that, if you load the default amount of data, you’re going to use up somewhere in the area of 55MB of disk space for *NorthwindBulk*. Also, be aware that the population script can take a while to run, as it has to generate and load thousands and thousands of rows of data.*

Now just re-create your view and index in your new *NorthwindBulk* database.

```
USE NorthwindBulk
GO

CREATE VIEW CustomerOrders_vw
```

```

WITH SCHEMABINDING
AS
SELECT    cu.CompanyName,
          o.OrderID,
          o.OrderDate,
          od.ProductID,
          p.ProductName,
          od.Quantity,
          od.UnitPrice
FROM      dbo.Customers AS cu
INNER JOIN  dbo.Orders AS o
          ON cu.CustomerID = o.CustomerID
INNER JOIN  dbo.[Order Details] AS od
          ON o.OrderID = od.OrderID
INNER JOIN  dbo.Products AS p
          ON od.ProductID = p.ProductID
GO

CREATE UNIQUE CLUSTERED INDEX ivCustomerOrders
ON CustomerOrders_vw(CompanyName, OrderID, ProductID)

```

Now re-run the original query, but against NorthwindBulk:

```

USE NorthwindBulk

SELECT * FROM CustomerOrders_vw

```

And check out your new query plan (see Figure 10-8).

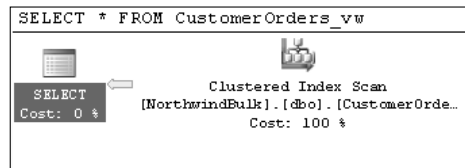


Figure 10-8

This time, SQL Server has enough data that it does a more thorough query plan. In this case, it accepts the index view that exists on our table. The overall performance of this view is now much faster (row for row) than the previous model would have been.

Summary

Views tend to be either the most over- or most under-used 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. 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 user of views for production queries — they add additional overhead and hurt performance.

Common uses for views include:

- ❑ Filtering rows
- ❑ Protecting sensitive data
- ❑ Reducing database complexity
- ❑ Abstracting multiple physical databases into one logical database

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 `Orders` table, and then used information from the freshly inserted row in an insert into the `Order Details` table. Batches and scripting will lead us right into stored procedures — the closest thing that SQL Server has to its own programs.

Exercises

1. Add a view called `Managers` in the Northwind database that shows only employees that supervise other employees.
2. Change the view you just created to be encrypted.
3. Create and index the existing Northwind view called "Products by Category" based on the columns `CategoryName` and `ProductName`.

11

Writing Scripts and Batches

Whether you've realized it or not, you've 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, however, 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. The SQL Server Management Studio provides many tools to help you with your script writing. The basic query window is color coded to help you not only recognize keywords, but also understand their nature. In addition, you have a step debugger, code templates, the object browser, and more.

Chapter 11

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 the script that we used to `INSERT` order records in the chapter on views (Chapter 10):

```
USE Northwind

DECLARE @Ident int

INSERT INTO Orders
(CustomerID, OrderDate)
VALUES
('ALFKI', DATEADD(day, -1, GETDATE()))

SELECT @Ident = @@IDENTITY

INSERT INTO [Order 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, `Northwind`). 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.

Next we have a `DECLARE` statement to declare a variable. We've talked about `DECLARE` statements briefly before, but let's expand on this some.

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. Technically, we could have got away without declaring this variable — instead, we could have chosen to just use `@@IDENTITY` directly. `@@IDENTITY` is a system function. It is always available, and supplies the last identity value that was assigned in the current connection. As with most system functions, you should make a habit of explicitly moving the value in `@@IDENTITY` to a local variable. That way, you're sure that it won't get changed accidentally. There was no danger of that in this case, but, as always, be consistent.

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.

So why have two ways of doing this? Actually, I still don't know. After publishing two books that ask this very question, I figured someone would e-mail me and give me a good answer — they didn't. Suffice to say that `SET` is now part of the ANSI standard, and that's why it's been put in there. However, I can't find anything wrong with the same functionality in `SELECT` — even ANSI seems to think that it's OK. I'm sure there's a purpose in the redundancy, but what it is I can't tell you. That said, there are some differences in the way they are typically put to use.

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 would be:

```
SET @TotalCost = 10  
SET @TotalCost = @UnitCost * 1.1
```


Chapter 11

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 Northwind

DECLARE @Test money

SET @Test = MAX(UnitPrice) FROM [Order Details]
SELECT @Test
```

Causes an error, but:

```
USE Northwind

DECLARE @Test money

SET @Test = (SELECT MAX(UnitPrice) FROM [Order Details])
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, our last illustration above would be far more typically done using a `SELECT`:

```
USE Northwind

DECLARE @Test money

SELECT @Test = MAX(UnitPrice) FROM [Order Details]
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 6+ 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
@@CURSOR_ROWS	Returns how many rows are currently in the last cursor set opened on the current connection.	SQL 7 can populate cursors asynchronously. Be aware that the value in this variable may change if the cursor is still in the process of being populated.
@@DATEFIRST	Returns what is currently set as the first day of the week (say, Sunday vs. Monday).	Is a system-wide setting — if someone changes the setting, you may not get the result you expect.
@@ERROR	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.
@@FETCH_STATUS	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.
@@IDENTITY	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.
@@OPTIONS	Returns information about options that have been set using the <code>SET</code> command.	Since you get only 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 is set, you must use the option value together with a bitwise operator.

Table continued on following page

Variable	Purpose	Comments
@@REMSERVER	Used only in stored procedures. Returns the value of the server that called the stored procedure.	Handy when you want the sproc to behave differently depending on the remote server (often a geographic location) from which it was called. Still, in this era of .NET, I would question whether anything needing this variable might have been better written using other functionality found in .NET.
@@ROWCOUNT	One of the most used system functions. Returns the number of rows affected by the last statement.	Commonly used in non runtime error checking. For example, if you try to DELETE a row using a WHERE clause, and no rows are affected, then that would imply that something unexpected happened. You can then raise an error manually.
@@SERVERNAME	Returns the name of the local server that the script is running from.	Can be changed by using sp_addserver 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 ROLLBACK TRAN statement decrements @@TRANCOUNT to 0 unless you are using savepoints. BEGIN TRAN increments @@TRANCOUNT by 1, COMMIT TRAN decrements @@TRANCOUNT by 1.
@@VERSION	Returns the current version of SQL Server as well as the date, processor and O/S architecture.	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.

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 5? An identity column is one where we don't supply a value, and SQL Server inserts a numbered value automatically.

In our example case, we obtain the value of @@IDENTITY right after performing an insert into the `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 `Order Details` table, but we need to know the value of the primary key in the associated record in the `Orders` table (remember, there is a foreign key constraint on the `Order Details` table that references the `Orders` table). Because 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 because 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, however, 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 `Order Details` table, which, since `Order Details` has no identity column, means that @@IDENTITY would have been 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.

Let's create a couple of tables to try this out:

```
CREATE TABLE TestIdent
(
    IDCol    int    IDENTITY
    PRIMARY KEY
)

CREATE TABLE TestChild1
(
    IDcol    int
    PRIMARY KEY
    FOREIGN KEY
        REFERENCES TestIdent(IDCol)
)

CREATE TABLE TestChild2
(
    IDcol    int
    PRIMARY KEY
    FOREIGN KEY
        REFERENCES TestIdent(IDCol)
)
```

What we have here is a parent table—it has an identity column for a primary key (as it happens, that's the only column it has). We also have two child tables. They each are the subject of an identifying relationship—that is, they each take at least part (in this case all) of their primary key by placing a foreign key on another table (the parent). So what we have is a situation where the two child tables need to get their key from the parent. Therefore, we need to insert a record into the parent first, and then retrieve the identity value generated so we can make use of it in the other tables.

Try It Out

Now that we have some tables to work with, we're ready to try a little test script:

```
/*
*****
* This script illustrates how the identity
* value gets lost as soon as another INSERT
* happens
***** */

DECLARE @Ident    int    -- This will be a holding variable
                        -- We'll use it to show how we can
                        -- move values from system functions
                        -- into a safe place.

INSERT INTO TestIdent
    DEFAULT VALUES

SET @Ident = @@IDENTITY
PRINT 'The value we got originally from @@IDENTITY was ' +
    CONVERT(varchar(2),@Ident)
PRINT 'The value currently in @@IDENTITY is ' + CONVERT(varchar(2),@@IDENTITY)
/* On this first INSERT using @@IDENTITY, we're going to get lucky.
** We'll get a proper value because there is nothing between our
** original INSERT and this one. You'll see that on the INSERT that
** will follow after this one, we won't be so lucky anymore. */
INSERT INTO TestChild1
VALUES
    (@@IDENTITY)

PRINT 'The value we got originally from @@IDENTITY was ' +
    CONVERT(varchar(2),@Ident)
IF (SELECT @@IDENTITY) IS NULL
    PRINT 'The value currently in @@IDENTITY is NULL'
ELSE
    PRINT 'The value currently in @@IDENTITY is ' + CONVERT(varchar(2),@@IDENTITY)

-- The next line is just a spacer for our print out
PRINT ''

/* The next line is going to blow up because the one column in
** the table is the primary key, and primary keys can't be set
** to NULL. @@IDENTITY will be NULL because we just issued an
** INSERT statement a few lines ago, and the table we did the
** INSERT into doesn't have an identity field. Perhaps the biggest
** thing to note here is when @@IDENTITY changed - right after
** the next INSERT statement. */
INSERT INTO TestChild2
VALUES
    (@@IDENTITY)
```

How It Works

What we're doing in this script is seeing what happens if we depend on @@IDENTITY directly rather than moving the value off to a safe place. When we execute the preceding script, everything's going to work just fine until the final INSERT. That final statement is trying to make use of @@IDENTITY directly, but the preceding INSERT statement has already changed the value in @@IDENTITY. Because that statement is on a table with no identity column, the value in @@IDENTITY is set to NULL. Because we can't have a NULL value in our primary key, the last INSERT fails:

```
(1 row(s) affected)
The value we got originally from @@IDENTITY was 1
The value currently in @@IDENTITY is 1

(1 row(s) affected)
The value we got originally from @@IDENTITY was 1
The value currently in @@IDENTITY is NULL

Msg 515, Level 16, State 2, Line 44
Cannot insert the value NULL into column 'IDcol', table 'master.dbo.TestChild2';
column does not allow nulls. INSERT fails.
```

The statement has been terminated.

If we make just one little change (to save the original @@IDENTITY value):

```
/* *****
 * This script illustrates how the identity
 * value gets lost as soon as another INSERT
 * happens
 * ***** */

DECLARE @Ident    int    -- This will be a holding variable
                        -- We'll use it to show how we can
                        -- move values from system functions
                        -- into a safe place.

INSERT INTO TestIdent
    DEFAULT VALUES

SET @Ident = @@IDENTITY
PRINT 'The value we got originally from @@IDENTITY was ' +
    CONVERT(varchar(2),@Ident)
PRINT 'The value currently in @@IDENTITY is ' + CONVERT(varchar(2),@@IDENTITY)
/* On this first INSERT using @@IDENTITY, we're going to get lucky.
** We'll get a proper value because there is nothing between our
** original INSERT and this one. You'll see that on the INSERT that
** will follow after this one, we won't be so lucky anymore. */
INSERT INTO TestChild1
VALUES
    (@@IDENTITY)

PRINT 'The value we got originally from @@IDENTITY was ' +
    CONVERT(varchar(2),@Ident)
```

Chapter 11

```
IF (SELECT @@IDENTITY) IS NULL
    PRINT 'The value currently in @@IDENTITY is NULL'
ELSE
    PRINT 'The value currently in @@IDENTITY is ' + CONVERT(varchar(2),@@IDENTITY)

-- The next line is just a spacer for our print out
PRINT ''

/* This time all will go fine because we are using the value that
** we have placed in safekeeping instead of @@IDENTITY directly.*/
INSERT INTO TestChild2
VALUES
    (@Ident)
```

This time everything runs just fine:

```
(1 row(s) affected)
The value we got originally from @@IDENTITY was 1
The value currently in @@IDENTITY is 1
(1 row(s) affected)
The value we got originally from @@IDENTITY was 1
The value currently in @@IDENTITY is NULL
(1 row(s) affected)
```

In this example, it was fairly easy to tell that there was a problem because of the attempt at inserting a `NULL` into the primary key. Now, imagine a far less pretty scenario—one where the second table did have an identity column. You could easily wind up inserting bogus data into your table and not even knowing about it—at least not until you already had a very serious data integrity problem on your hands!

Using @@ROWCOUNT

In the many queries that we ran up to this point, it's always been pretty easy to tell how many rows a statement affected—Query Analyzer tells us. For example, if we run:

```
USE Northwind

SELECT * FROM Categories
```

then we see all the rows in `Categories`, 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):

```
(8 row(s) affected)
```

But what if we need to *programmatically* know how many rows were affected? Much like `@@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 Northwind
GO

DECLARE @RowCount int -- Notice the single @ sign

SELECT * FROM Categories

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 8
```

We'll take a look at ways this might be useful when we look at stored procedures later in the book. For now, just realize that this provides us with a way to learn something about what a statement did, and it's not limited to use `SELECT` statements — `UPDATE`, `INSERT`, and `DELETE` also set this value.

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 safe keeping 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.
- ❑ 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 = 'ALFKI' 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 above 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, however, 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 only recognized 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.

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 cancelled 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 14, 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 only prevent 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 12).

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`

Chapter 11

- ❑ 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 my case, it happened to be the `master` database, so that's where my 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 (OK, 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 your code and finds that you are referencing a database with your `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:

```
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.
```

Chapter 11

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`, however, and everything is working fine:

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

SQLCMD

SQLCMD is a utility that allow 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 -Pmypass -Q "SELECT * FROM Northwind..Shippers"
```

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:

```

C:\>Osql -Usa -Pmypass -Q "SELECT * FROM Northwind..Shippers"
ShipperID      CompanyName      Phone
-----
1              Speedy Express    (503) 555-9831
2              United Package    (503) 555-3199
3              Federal Shipping  (503) 555-9931
(3 rows affected)
C:\>

```

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
```

This should take you down to a blank line (with no prompt of any kind), where you can enter in this:

```
SELECT * FROM Northwind..Shippers
```

Then press *F6* and *Return* (this ends the creation of our text file). You should get back a message like:

```
1 file(s) copied.
```

Chapter 11

Now let's retry our earlier query using a script file this time. The command line at the prompt has only a slight change to it:

```
C:\>sqlcmd -Usa -Pmypass -i testsql.sql
```

This should get us exactly the same results as we had 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.

Try It Out

As a final example of `SQLCMD`, let's utilize it to generate a text file that we might import into another application for analysis (Excel for example).

If you look back in Chapter 10, we create a view that listed yesterday's orders for us. First, we're doing to take the core query of that view, and stick it into a text file.

```
C:\copy con YesterdaysOrders.sql
```

This should again take you down to a blank line (with no prompt of any kind), where you can enter this:

```
SELECT    cu.CompanyName,
          o.OrderID,
          o.OrderDate,
          od.ProductID,
          p.ProductName,
          od.Quantity,
          od.UnitPrice,
          od.Quantity * od.UnitPrice AS ExtendedPrice
FROM      Customers AS cu
INNER JOIN Orders AS o
          ON cu.CustomerID = o.CustomerID
INNER JOIN [Order Details] AS od
          ON o.OrderID = od.OrderID
INNER JOIN Products AS p
          ON od.ProductID = p.ProductID
WHERE     CONVERT(varchar(12),o.OrderDate,101) =
          CONVERT(varchar(12),DATEADD(day,-1,GETDATE()),101)
```

Again press F6 to tell Windows to save the file for us.

We now have our text file source for our query, and are nearly ready to have `SQLCMD` help us generate our output. First, however, is that we need there to be some data from yesterday (none of the sample data is going to have data from yesterday unless you just ran the script to generate some orders. Just to be sure of which script I'm talking about here, I mean the one shown in the views chapter, and reviewed it again earlier in this chapter. So, with this in mind, let's run that generation script one more time (you can do this through the Query Window if you like):

```
USE Northwind

DECLARE @Ident int

INSERT INTO Orders
```

```

(CustomerID,OrderDate)
VALUES
('ALFKI', DATEADD(day,-1,GETDATE()))

SELECT @Ident = @@IDENTITY

INSERT INTO [Order Details]
(OrderID, ProductID, UnitPrice, Quantity)
VALUES
(@Ident, 1, 50, 25)

SELECT 'The OrderID of the INSERTed row is ' + CONVERT(varchar(8),@Ident)

```

OK, so we have at least one row that is an order with yesterday's date now. So we're most of the way ready to go; however, we've said we want our results to a text file, so we'll need to add some extra parameters to our SQLCMD command line this time around to tell SQL Server where to put the output:

```

C:\Documents and Settings\robov.BARNICLE>sqlcmd -UMyLogin -PMYPass
-iYesterdaysOrders.sql -oYesterdaysOrders.txt

```

There won't be anything special or any fanfare when SQLCMD is done running this — you'll simply get your Windows drive prompt again (C:\ most likely), but check out what is in our YesterdaysOrders.txt file now:

```

C:\>TYPE YesterdaysOrders.txt

```

This gives us our one row:

```

Changed database context to 'Northwind'.
CompanyName      OrderID    OrderDate                ProductID    ProductName
Quantity UnitPrice
ExtendedPrice
-----
-----
-----
Alfreds Futterkiste      11078 2005-08-27 14:35:50.163      1    Chai
25    50.0000
                1250.0000

(1 rows affected)

```

How It Works

We started out by bundling the SQL commands we would need into a single script — first, the `USE` command, and then the actual `SELECT` statement.

We then execute our statement using `SQLCMD`. The `-U` and `-P` commands provided the login user name and password information just as they did earlier in the chapter. The `-i` parameter told `SQLCMD` that we had an input file, and we included that file name *immediately* following the `-i` parameter. Finally, we included the `-o` parameter to tell `SQLCMD` that we wanted the output written to a file (we, of course, then provide a file name — `YesterdaysOrders.txt`). Don't get confused by the two files both named `YesterdaysOrders` — they are separate files with the `.sql` and `.txt` files separating what their particular use is for.

There is 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

OK, 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 run time?

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 run time. 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 Northwind database by creating a dummy table to grab our dynamic information out of:

```
USE Northwind
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,
    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 TABLE_NAME
FROM Information_Schema.Tables
WHERE TABLE_TYPE = 'BASE TABLE'
```

This should get us a response something like:

```
(17 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.

OK, 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 run time by using its ID. For example, I’ll pull out all the data for the table with an ID of 1:

```
/* First, declare a variable to hold the table name. Remember,
** object names can be 128 characters long
*/
DECLARE @TableName      varchar(128)

-- Now, grab the table name that goes with our ID
SELECT @TableName = TableName
      FROM DynamicSQLExample
      WHERE TableID = 14

-- Finally, pass that value into the EXEC statement
EXEC ('SELECT * FROM ' + @TableName)
```

If your table names went into the `DynamicSQLExample` table the way mine did, then a `TableID` of 14 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):

CategoryID	CategoryName	Description	
1	Beverages	Soft drinks, coffees, teas, beers, and ales	...
2	Condiments	Sweet and savory sauces, relishes, spreads, and seasonings	...
3	Confections	Desserts, candies, and sweet breads	...
4	Dairy Products	Cheeses	...
5	Grains/Cereals	Breads, crackers, pasta, and cereal	...
6	Meat/Poultry	Prepared meats	...
7	Produce	Dried fruit and bean curd	...
8	Seafood	Seaweed and fish	...

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 14).
- ❑ 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 can not 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 Northwind

/* First, we'll declare two 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 Employees WHERE EmployeeID = 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 Employees WHERE EmployeeID = 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 13
Must declare the scalar variable '@OutVar'.
Msg 137, Level 15, State 1, Line 16
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 Northwind

-- 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 Employees WHERE EmployeeID = 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 Nancy
```

Notice the way that I'm using two quote marks right next to each other to indicate that I really want a quote mark rather than to terminate my string.

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 your 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 Northwind

EXEC('SELECT * FROM Customers')
SELECT 'The Rowcount is ' + CAST(@@ROWCOUNT as varchar)
```

This yields us (after the result set):

```
The Rowcount is 91
```

Security Contexts and EXEC

This is a tough one to cover at this point because we haven't covered the issues yet with stored procedures and security. Still, the discussion of the EXEC command belonged here rather than in the sprocs chapter, so here we are (this is the only part of this discussion that gets wrapped up in sprocs, so bear with me).

When you give someone the right to run a stored procedure, you imply that they also gain 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 they do not have rights to the `Employees` table directly. This is really handy for reasons we will explore later in our sprocs chapter.

Developers usually assume that this same implied right is valid for an EXEC statement also — it isn't. 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 `spNewEmployees`, but I do not have rights to the `Employees` table. If `spNewEmployees` gets the values by running a simple SELECT statement, then everything is fine. If, however, `spNewEmployees` 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 `Employees` table.

Since we don't have that much information on sprocs yet, I'm going to bypass further discussion of this for now, but we will come back to it when we discuss sprocs later on.

Use of Functions in Concatenation and EXEC

This one is actually more of a nuisance than anything else because 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 Northwind

-- This won't work
DECLARE @NumberOfLetters int
SET @NumberOfLetters = 15
EXEC('SELECT LEFT(CompanyName, ' + CAST(@NumberOfLetters AS varchar) + ') AS
ShortName
FROM Customers')
GO

-- But this does
DECLARE @NumberOfLetters AS int
SET @NumberOfLetters = 15
DECLARE @str AS varchar(255)
SET @str = 'SELECT LEFT(CompanyName, ' + CAST(@NumberOfLetters AS varchar) + ') AS
ShortName FROM Customers'
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:

```
ShortName
-----
Alfreds Futterk
Ana Trujillo Em
...
...
Wolski Zajazd
```

EXEC and UDFs

This is a tough one to touch on because we haven't gotten to user-defined functions as yet, but suffice to say that you are not allowed to use `EXEC` to run dynamic SQL within a UDF—period. (`EXEC` to run a sproc is, however, legal in a few cases.)

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).

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.

Exercises

1. Write a simple script that creates two integer variables (one called Var1 and one called Var2), places the values 2 and 4 in them respectively, and then outputs the value of the two variables added together.
2. Create a variable called MinOrder and populate it with the smallest line item amount after discount for the Northwind CustomerNo 'ALFKI' (Careful: we're dealing with currency here, so don't just assume you're going to use an int.) Output the final value of MinOrder.
3. Use SQLCMD to output the results of the query `SELECT COUNT(*) FROM Customers` to the console window.

12

Stored Procedures

Ah, the good stuff. If you're a programmer coming from a procedural language, then this is probably the part you've been waiting for. It's time to get down to the main variety of "code" of SQL Server, but before we get going too far down that road, I need to prepare you for what lies ahead — there's probably a lot less than you're expecting, and, at the very same time, a whole lot more. The good news is that, with SQL Server 2005, you have .NET support — giving us a veritable "oo la la!" of possibilities.

You see, a *stored procedure*, sometimes referred to as a *sproc* (which I usually say as one word, but I've sometimes heard this pronounced as "ess-proc"), is really just something of a script — or more correctly speaking, a *batch* — that is stored in the database rather than in a separate file. Now this comparison is not an exact one by any means — sprocs have things such as input parameters, output parameters, and return values that a script doesn't really have, but the comparison is not that far off either.

For now, SQL Server's only "programming" language continues to be T-SQL, and that leaves us miles short of the kind of procedural horsepower that you expect when you think of a true programming language. However, T-SQL blows C, C++, Visual Basic, Java, Delphi, or whatever away when it comes to what T-SQL is supposed to do — work on data definition, manipulation, and access. But T-SQL's horsepower stops right about there — at data access and management. In short, it has an adequate amount of power to get most simple things done, but it's not always the place to do it.

For this chapter, we're not going to worry all that much about T-SQL's shortcomings — instead, we'll focus on how to get the most out of T-SQL, and even toss in a smattering of what .NET has added to the picture. We'll take a look at parameters, return values, control of flow, looping structures, both basic and advanced error trapping, and more. In short, this is a big chapter that deals with many subjects. All of the major subject areas are broken up into their own sections, so you can take them one step at a time, but let's start right out with the basics of getting a sproc created.

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 that you first saw when we took a look at 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, we still have our 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 just fine, but as always, I recommend that you be consistent regarding which one you choose (personally, I like the saved keystrokes of `PROC`). 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 we'll 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 examples of basic sproc syntax is to get down to 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 would hope that, by now, you have the query that would return all the contents of a table down cold (Hint: `SELECT * FROM . . .`). If not, then I would suggest a return to the chapter on basic query syntax. In order to create a sproc that performs this basic query, we just add the query in the code area of the sproc syntax:

```
USE Northwind

GO
CREATE PROC spShippers
AS
  SELECT * FROM Shippers
```

Not too rough—eh? If you're wondering why I put the `GO` keyword in before the `CREATE` syntax (if we were just running a simple `SELECT` statement, we wouldn't need it), it's because most non-table `CREATE` statements cannot share a batch with any other code. Indeed, even with a `CREATE TABLE` statement, leaving

out the GO can become rather dicey. In this case, having the USE command together with our CREATE PROC statement would have been a no-no, and would have generated an error.

Now that we have our sproc created, let's execute it to see what we get:

```
EXEC spShippers
```

We get exactly what we would have gotten if we had run the SELECT statement that's embedded in the sproc:

ShipperID	CompanyName	Phone
1	Speedy Express	(503) 555-9831
2	United Package	(503) 555-3199
3	Federal Shipping	(503) 555-9931

(3 row(s) affected)

You've just written your first sproc. It was easy of course, and frankly, for most situations, sproc writing isn't nearly as difficult as most database people would like to have you think (job preservation), but there are lots of possibilities, and we've only seen the beginning.

Changing Stored Procedures with ALTER

I'm going to admit something here—I cut and pasted almost all the text you're about to read in this and the next section ("Dropping Sprocs"), from the chapter on views. What I'm pointing out by telling you this is that they work almost identically 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:

- ❑ ALTER PROC expects to find an existing sproc, where 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 is all still there.
- ❑ ALTER PROC retains any dependency information on other objects that may call the sproc being altered.

The latter of these two is the biggie.

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 then you will need to entirely re-establish your permissions on who can and can't use the sproc.

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 (more on that later), but it wouldn't be much help in most circumstances if it couldn't accept some data to tell it what to do. For example, it doesn't do much good to have anspDeleteShipper stored procedure if we can't tell it what shipper we want to delete, so we use an *input parameter*. Likewise, we often want to get information back 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 where we update several records in a table and we'd like to know just how many we updated. Often, this isn't easily handed back in recordset form, so we 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 datatype
- ☐ The default value
- ☐ The direction

The syntax is:

```
@parameter_name [AS] datatype [= default|NULL] [VARYING] [OUTPUT|OUT]
```

The name has a pretty simple set of rules to it. First, it must start with the @ sign. Other than that, the rules for naming are pretty much the same as the rules for naming described in Chapter 1, except that they cannot have embedded spaces.

The datatype, much like the name, must be declared just as you would for a variable — with a valid SQL Server built-in or user defined datatype.

One special thing in declaring the datatype 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 and well outside of the scope of this book, but keep it in mind in case you see it in books online or other documentation and wonder what that's all about.

Note also that OUTPUT can be abbreviated to OUT.

The default is the first place we start to see any real divergence from variables. Where variables are always initialized to a NULL value, parameters are not. Indeed, if you don't supply a default value, then the parameter is assumed to be required, and a beginning value must be supplied when the sproc is called. To supply a default, you simply add an = sign after the datatype and then provide the default value. Once you've done this, the users of your sproc can decide to supply no value for that parameter, or they can provide their own value.

Let's create another sproc, only this time we'll make use of a few input parameters to create a new record in the Shippers table:

```
USE Northwind
GO

CREATE PROC spInsertShipper
    @CompanyName    nvarchar(40),
    @Phone          nvarchar(24)
AS
    INSERT INTO Shippers
    VALUES
        (@CompanyName, @Phone)
```

Our last sproc told us what data is currently in the Shippers table, but let's use our new sproc to insert something new:

```
EXEC spInsertShipper 'Speedy Shippers, Inc.', '(503) 555-5566'
```

If this is executed from the Query Analyzer, we see the results of our stored procedure run just as if we had run the INSERT statement ourselves:

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

Now let's run our first sproc again and see what we get:

```
EXEC spShippers
```

ShipperID	CompanyName	Phone
-----	-----	-----
1	Speedy Express	(503) 555-9831
2	United Package	(503) 555-3199
3	Federal Shipping	(503) 555-9931
4	Speedy Shippers, Inc.	(503) 555-5566

```
(4 row(s) affected)
```

Sure enough, our record has been inserted, and a new identity has been filled in for it.

Because we didn't supply any default values for either of the parameters, both parameters are considered to be required. That means that, in order to have success running this sproc, we *must* supply both parameters. You can easily check this out by executing the sproc again with only one or no parameters supplied:

```
EXEC spInsertShipper 'Speedy Shippers, Inc.'
```

SQL Server wastes no time in informing you of the error of your ways:

```
Msg 201, Level 16, State 4, Procedure spInsertShipper, Line 0
Procedure or Function 'spInsertShipper' expects parameter '@Phone', which was not
supplied.
```

Supplying Default Values

To make a parameter optional, you have to supply a default value. To do this, you just add an = together with the value you want to use for a default after the datatype but before the comma.

Let’s try building our INSERT sproc again, only this time we won’t require the phone number:

```
USE Northwind
GO
```

```
CREATE PROC spInsertShipperOptionalPhone
    @CompanyName    nvarchar(40),
    @Phone          nvarchar(24) = NULL
AS
    INSERT INTO Shippers
    VALUES
        (@CompanyName, @Phone)
```

Now we’re ready to re-issue our command, but using the new sproc this time:

```
EXEC spInsertShipperOptionalPhone 'Speedy Shippers, Inc'
```

This time everything works just fine, and our new row is inserted:

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

```
EXEC spShippers
```

ShipperID	CompanyName	Phone
1	Speedy Express	(503) 555-9831
2	United Package	(503) 555-3199
3	Federal Shipping	(503) 555-9931
4	Speedy Shippers, Inc.	(503) 555-5566
5	Speedy Shippers, Inc.	NULL

(5 row(s) affected)

In this particular case, we set the default to NULL, but the value could have been anything that was compatible with the datatype of the parameter for which we are establishing the default. Also, notice that we didn’t have to establish a default for both values — we can make one have a default, and one not — we decide which parameters are required (have no default), and which are not (have a default).

Creating Output Parameters

Sometimes, you want to pass non-recordset information out to whatever called your sproc. One example of this would create a modified version of our last two sprocs.

Let's say, for example, that we are performing an insert into a table (like we did in the last example), but we are planning to do additional work using the inserted record.

Or more specifically, maybe we're inserting a new record into our `Orders` table in `Northwind`, but we also need to insert detail records in the `Order Details` table. In order to keep the relationship intact, we have to know the identity of the `Orders` record before we can do our inserts into the `Order Details` table. The sproc will look almost exactly like our `spInsertShipper` did, except that it will have parameters that match up with the different columns in the table and, most importantly of all, it will have an output parameter for the identity value that is generated by our insert:

```
USE Northwind
GO

CREATE PROC spInsertOrder
    @CustomerID    nvarchar(5),
    @EmployeeID    int,
    @OrderDate     datetime      = NULL,
    @RequiredDate  datetime      = NULL,
    @ShippedDate   datetime      = NULL,
    @ShipVia       int,
    @Freight       money,
    @ShipName      nvarchar(40)  = NULL,
    @ShipAddress   nvarchar(60)  = NULL,
    @ShipCity      nvarchar(15)  = NULL,
    @ShipRegion    nvarchar(15)  = NULL,
    @ShipPostalCode nvarchar(10)  = NULL,
    @ShipCountry   nvarchar(15)  = NULL,
    @OrderID       int           OUTPUT
AS
    /* Create the new record */
    INSERT INTO Orders
    VALUES
        (
            @CustomerID,
            @EmployeeID,
            @OrderDate,
            @RequiredDate,
            @ShippedDate,
            @ShipVia,
            @Freight,
            @ShipName,
            @ShipAddress,
            @ShipCity,
            @ShipRegion,
            @ShipPostalCode,
            @ShipCountry
        )

    /* Move the identity value from the newly inserted record into
       our output variable */
    SELECT @OrderID = @@IDENTITY
```

Chapter 12

Now, let’s try this baby out, only this time, let’s set our parameter values by reference rather than by position. In order to see how our output parameter is working, we’ll also need to write a little bit of test code in the script that executes the sproc:

```
USE Northwind
GO

DECLARE    @MyIdent    int

EXEC spInsertOrder
    @CustomerID = 'ALFKI',
    @EmployeeID = 5,
    @OrderDate = '5/1/1999',
    @ShipVia = 3,
    @Freight = 5.00,
    @OrderID = @MyIdent OUTPUT

SELECT @MyIdent AS IdentityValue

SELECT OrderID, CustomerID, EmployeeID, OrderDate, ShipName
FROM Orders
WHERE OrderID = @MyIdent
```

Notice that we didn’t supply all of the parameters. Some of them were optional, and we decided to leave some of those off—which would take the default value. If we had been calling the sproc and passing values in using positional parameters, then we would have had to address each position in the parameter list at least until the last parameter for which we wanted to supply a value.

Let’s see what this gives us—keep in mind that your identity value may vary from mine depending on what modifications you’ve already made in the `Orders` table:

```
(1 row(s) affected)
IdentityValue
-----
11078
(1 row(s) affected)

OrderID      CustomerID      EmployeeID      OrderDate      ShipName
-----
11078      ALFKI          5              1999-05-01 00:00:00.000      NULL
(1 row(s) affected)
```

The first row affected line is really feedback from the sproc itself—it inserted one row. The second resultset provides us with the identity value that was inserted—for me, this value was 11078—this is positive proof that our identity value was indeed passed out of the sproc by the output parameter. Finally, we selected several columns from that row in the `Orders` table to verify that the row was indeed inserted using the data we expected.

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 was 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 run-time error (you won't get any messages about it), but the value for the output parameter won't be moved into your variable (you'll just 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 our previous sproc, the internal parameter was called `@OrderID`, but the variable the value was passed to was called `@MyIdent`.
- ❑ The `EXEC` (or `EXECUTE`) keyword was required since 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) — personally, I recommend that you train yourself to use it regardless.

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. 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 (aka `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 closest 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.

Chapter 12

This brings us back to one of the most common traps that I see SQL programmers fall into — improper use 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 below), 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 above. 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.

Let's create a new edition of our last query, and deal with the situation where someone supplies an OrderDate that is older than we want to accept.

Our sales manager is upset because someone has been putting in orders long after she has already completed her sales analysis for the time-period in which that order is. She has established a new policy that says that an order must be entered into the system within seven days after the order is taken, or the order date is considered to be invalid and is to be set to NULL.

How do we change the value of the order date? That's where our IF . . . ELSE statement comes in.

We need to perform a simple test, in which we'll need to make use of the DATEDIFF function. The syntax for DATEDIFF is:

```
DATEDIFF(<datepart>, <startdate>, <enddate>)
```

DATEDIFF compares our two dates — in this case the supplied order date and the current date. It can actually compare any part of the datetime data supplied from the year down to the millisecond. In our case, a simple dd for day will suffice, and we'll put it together with an IF statement:

```
IF DATEDIFF(dd, @OrderDate, GETDATE()) > 7
```

In the event that our returned value is over 7 — that is, over 7 days old — then we want to change the value that we insert:

```
SELECT @OrderDate = NULL
```

Now that we've got ourselves set with our IF statement, let's write that new version of the spInsertOrder sproc:

```
USE Northwind
GO

CREATE PROC spInsertDateValidatedOrder
```

```

    @CustomerID      nvarchar(5),
    @EmployeeID       int,
    @OrderDate        datetime      = NULL,
    @RequiredDate     datetime      = NULL,
    @ShippedDate      datetime      = NULL,
    @ShipVia          int,
    @Freight          money,
    @ShipName         nvarchar(40) = NULL,
    @ShipAddress      nvarchar(60) = NULL,
    @ShipCity         nvarchar(15) = NULL,
    @ShipRegion       nvarchar(15) = NULL,
    @ShipPostalCode   nvarchar(10) = NULL,
    @ShipCountry      nvarchar(15) = NULL,
    @OrderID          int          OUTPUT

AS

/* Test to see if supplied date is over seven days old, if so
   replace with NULL value */
IF DATEDIFF(dd, @OrderDate, GETDATE()) > 7
    SELECT @OrderDate = NULL

/* Create the new record */
INSERT INTO Orders
VALUES
(
    @CustomerID,
    @EmployeeID,
    @OrderDate,
    @RequiredDate,
    @ShippedDate,
    @ShipVia,
    @Freight,
    @ShipName,
    @ShipAddress,
    @ShipCity,
    @ShipRegion,
    @ShipPostalCode,
    @ShipCountry
)

/* Move the identity value from the newly inserted record into
   our output variable */
SELECT @OrderID = @@IDENTITY

```

Now let's run the same test script we used for the original `spInsertOrder` sproc with only minor modifications to deal with our new situation:

```

USE Northwind
GO

DECLARE    @MyIdent    int

EXEC spInsertDateValidatedOrder
    @CustomerID = 'ALFKI',
    @EmployeeID = 5,

```

```
@OrderDate = '5/1/1999',
@ShipVia = 3,
@Freight = 5.00,
@OrderID = @MyIdent OUTPUT

SELECT @MyIdent AS IdentityValue

SELECT OrderID, CustomerID, EmployeeID, OrderDate, ShipName
FROM Orders
WHERE OrderID = @MyIdent
```

This time, even though most of the sproc is the same, we change what we put into the database, and therefore, what we see in our selected results:

```
(1 row(s) affected)
IdentityValue
-----
11079
(1 row(s) affected)

OrderID      CustomerID      EmployeeID      OrderDate      ShipName
-----
11079        ALFKI           5               NULL            NULL
(1 row(s) affected)
```

Even though we supplied the same date as last time (5/1/1999), that isn't the value that was inserted — our `IF` statement picked off the illegal value and changed it before the insert.

The ELSE Clause

Now this thing about being able to change the data on the fly 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 example just a bit, let's look at the oldest records that are currently in the `Orders` table of `Northwind`:

```
USE Northwind
GO

SELECT TOP 5 OrderID, OrderDate
FROM Orders
WHERE OrderDate IS NOT NULL
ORDER BY OrderDate
```

There's something interesting about the results:

OrderID	OrderDate
-----	-----
10248	1996-07-04 00:00:00.000
10249	1996-07-05 00:00:00.000
10250	1996-07-08 00:00:00.000
10251	1996-07-08 00:00:00.000
10252	1996-07-09 00:00:00.000

(5 row(s) affected)

None of the dates has a time component — OK, technically they're all at midnight, but I suspect you get the picture. It's likely that this was done on purpose, as it makes date (without time) comparisons much easier.

What we want to do is convert our sproc to make sure that we store all dates as just dates — no times. The current sproc won't work because it will insert the entire date, including time — but to verify that, let's test it out:

```
USE Northwind
GO

DECLARE @MyIdent int
DECLARE @MyDate smalldatetime

SELECT @MyDate = GETDATE()

EXEC spInsertDateValidatedOrder
    @CustomerID = 'ALFKI',
    @EmployeeID = 5,
    @OrderDate = @MyDate,
    @ShipVia = 3,
    @Freight = 5.00,
    @OrderID = @MyIdent OUTPUT

SELECT @MyIdent AS IdentityValue

SELECT OrderID, CustomerID, EmployeeID, OrderDate, ShipName
FROM Orders
WHERE OrderID = @MyIdent
```

When we insert our date, the time comes along with it:

```
(1 row(s) affected)
IdentityValue
-----
11080
(1 row(s) affected)
OrderID      CustomerID      EmployeeID      OrderDate      ShipName
-----
11080        ALFKI          5              2000-07-22 16:48:00.000  NULL
(1 row(s) affected)
```

So, what we have is an either/or situation. Either we now want the date changed to `NULL`, or we want the time truncated from the date. Unfortunately, SQL Server doesn't give us a function that does it automatically (another severe let down in my not so humble opinion). Fortunately, however, we again have a workaround.

Truncating the Time from a Datetime Field

In order to truncate a date, we can either take the date apart piece by piece and reassemble it without the time, or, as I prefer, we can use the `CONVERT` function on it, to convert it to a timeless day and then convert it back.

`CONVERT()` is just one of many functions that are available to us in SQL Server. Originally, it was the one and only method to convert data between datatypes. These days, `CONVERT` should be getting much less use in scripts because much of its functionality is duplicated by `CAST()`, which is ANSI-compliant (whereas `CONVERT` isn't). Still, `CONVERT` has some special date formatting capabilities that can't be duplicated by `CAST`.

`CONVERT` works with this syntax:

```
CONVERT(<target data type>, <expression to be converted>, <style>)
```

The first two parameters are pretty self-describing, but the last one isn't—it only applies when dealing with dates and its purpose is to tell SQL Server in which format you want the date to be. Examples of common date formats include 1, for standard U.S. mm/dd/yy format and 12 for the standard ISO format (yymmdd). Adding 100 to any of the formats adds the full century to the date scheme (for example, standard U.S. format with a four-digit year — mm/dd/yyyy — has a style of 101).

For example, it would look something like this for the `GETDATE` function:

```
SELECT CONVERT(datetime, (CONVERT(varchar, GETDATE(), 112)))
```

This takes things to an ANSI date format and then back again:

```
-----  
2000-06-06 00:00:00.000  
(1 row(s) affected)
```

Implementing the ELSE Statement in Our Sproc

Now that we've figured out how to do the pieces, it's time to move that into our actual sproc. This time, however, we're going to make use of the `ALTER` command rather than creating a separate procedure. Remember that, even when using an `ALTER` statement, we must entirely redefine the procedure:

```
USE Northwind  
GO
```

```
ALTER PROC spInsertDateValidatedOrder  
    @CustomerID    nvarchar(5),  
    @EmployeeID    int,  
    @OrderDate      datetime      = NULL,
```

```

@RequiredDate    datetime    = NULL,
@ShippedDate     datetime    = NULL,
@ShipVia         int,
@Freight         money,
@ShipName        nvarchar(40) = NULL,
@ShipAddress     nvarchar(60) = NULL,
@ShipCity        nvarchar(15) = NULL,
@ShipRegion      nvarchar(15) = NULL,
@ShipPostalCode  nvarchar(10) = NULL,
@ShipCountry     nvarchar(15) = NULL,
@OrderID         int         OUTPUT

AS

/* I don't like altering input parameters—I find that it helps in debugging
** if I can refer to their original values at any time. Therefore, I'm going
** to declare a separate variable to assign the end value we will be
** inserting into the table. */
DECLARE @InsertedOrderDate smalldatetime

/* Test to see if supplied date is over seven days old, if so
** replace with NULL value
** otherwise, truncate the time to be midnight */
IF DATEDIFF(dd, @OrderDate, GETDATE()) > 7
    SELECT @InsertedOrderDate = NULL
ELSE
    SELECT @InsertedOrderDate =
        CONVERT(datetime, (CONVERT(varchar, @OrderDate, 112)))

/* Create the new record */
INSERT INTO Orders
VALUES
(
    @CustomerID,
    @EmployeeID,
    @InsertedOrderDate,
    @RequiredDate,
    @ShippedDate,
    @ShipVia,
    @Freight,
    @ShipName,
    @ShipAddress,
    @ShipCity,
    @ShipRegion,
    @ShipPostalCode,
    @ShipCountry
)

/* Move the identity value from the newly inserted record into
our output variable */
SELECT @OrderID = @@IDENTITY

```

Now, if we re-run the original batch, we have the effect we were after:

```
(1 row(s) affected)
IdentityValue
-----
11081
(1 row(s) affected)

OrderID      CustomerID      EmployeeID      OrderDate      ShipName
-----
11081        ALFKI           5               2000-07-22 00:00:00.000  NULL
(1 row(s) affected)
```

We now have a sproc that handles the insert differently depending on the specific values that are given to the sproc.

If you look closely, you'll note that I changed more than just the IF . . . ELSE statement for this version of the sproc—I also changed things so that a holding variable was declared for the order date.

The purpose behind this has to do with a general philosophy I have about changing input parameter values. With the exception of where you are changing parameter values for the express purpose of passing out a changed value, I don't think you should change parameter values. Why? Well, part of it is a clarity issue—I don't want people to have to look in multiple places for where my variables are declared if possible. The other reason is perhaps a more convincing one—debugging. I like to retain my input values for as long as possible so that, when I need to debug, I can easily check my input value against the various places in the code I make use of the input value. That is, I want to simplify being able to tell if things are working correctly.

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, only consider 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, 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 our last order insert sproc. This time, we're going to provide a little bit of useful information to our user as we go through code that alters what the caller of the sproc has provided. This can act as something of a lead-in for the upcoming section on error handling.

Any time we decide to change the data we're inserting to be something other than what the user supplied, we also need to inform the user of exactly what we're doing. We'll use a PRINT statement to output the specifics of what we've done. We'll add these PRINT statements as part of the code in our IF...ELSE statement so the information can be topical. Note that a PRINT statement doesn't generate any kind of error—it just provides textual information regardless of error status.

We'll discuss this further in the error handling section.

```

USE Northwind
GO

ALTER PROC spInsertDateValidatedOrder
    @CustomerID          nvarchar(5),
    @EmployeeID           int,
    @OrderDate            datetime    = NULL,
    @RequiredDate         datetime    = NULL,
    @ShippedDate          datetime    = NULL,
    @ShipVia              int,
    @Freight              money,
    @ShipName             nvarchar(40) = NULL,
    @ShipAddress          nvarchar(60) = NULL,
    @ShipCity             nvarchar(15) = NULL,
    @ShipRegion           nvarchar(15) = NULL,
    @ShipPostalCode       nvarchar(10) = NULL,
    @ShipCountry          nvarchar(15) = NULL,
    @OrderID              int         OUTPUT

```

```

AS

```

```

/* I don't like altering input paramters—I find that it helps in debugging

```


Chapter 12

```
** if I can refer to their original value at any time. Therefore, I'm going
** to declare a separate variable to assign the end value we will be
** inserting into the table. */
DECLARE @InsertedOrderDate smalldatetime

/* Test to see if supplied date is over seven days old, if so
** replace with NULL value
** otherwise, truncate the time to be midnight*/
IF DATEDIFF(dd, @OrderDate, GETDATE()) > 7
BEGIN
    SELECT @InsertedOrderDate = NULL
    PRINT 'Invalid Order Date'
    PRINT 'Supplied Order Date was greater than 7 days old.'
    PRINT 'The value has been reset to NULL'
END
ELSE
BEGIN
    SELECT @InsertedOrderDate =
        CONVERT(datetime, (CONVERT(varchar, @OrderDate, 112)))
    PRINT 'The Time of Day in Order Date was truncated'
END

/* Create the new record */
INSERT INTO Orders
VALUES
(
    @CustomerID,
    @EmployeeID,
    @InsertedOrderDate,
    @RequiredDate,
    @ShippedDate,
    @ShipVia,
    @Freight,
    @ShipName,
    @ShipAddress,
    @ShipCity,
    @ShipRegion,
    @ShipPostalCode,
    @ShipCountry
)

/* Move the identity value from the newly inserted record into
our output variable */
SELECT @OrderID = @@IDENTITY
```

Now when we execute our test batch, we get slightly different results. First, the test batch using the current date:

```
USE Northwind
GO

DECLARE @MyIdent int
```

```

DECLARE    @MyDate    smalldatetime

SELECT @MyDate = GETDATE()

EXEC spInsertDateValidatedOrder
    @CustomerID = 'ALFKI',
    @EmployeeID = 5,
    @OrderDate  = @MyDate,
    @ShipVia    = 3,
    @Freight    = 5.00,
    @OrderID    = @MyIdent OUTPUT

SELECT OrderID, CustomerID, EmployeeID, OrderDate, ShipName
FROM Orders
WHERE OrderID = @MyIdent

```

Note that we've deleted the line `SELECT @MyIdent AS IdentityValue` from the test batch for brevity's sake.

And we can see that not only was our value truncated in terms of the actual data, but we also have the message that explicitly tells us:

```

The Time of Day in Order Date was truncated
(1 row(s) affected)

```

OrderID	CustomerID	EmployeeID	OrderDate	ShipName
11080	ALFKI	5	1999-08-30 00:00:00.000	NULL

```

(1 row(s) affected)

```

Next, we run the older version of the test batch that manually feeds an older date:

```

USE Northwind
GO

DECLARE    @MyIdent    int

EXEC spInsertDateValidatedOrder
    @CustomerID = 'ALFKI',
    @EmployeeID = 5,
    @OrderDate  = '1/1/1999',
    @ShipVia    = 3,
    @Freight    = 5.00,
    @OrderID    = @MyIdent OUTPUT

SELECT OrderID, CustomerID, EmployeeID, OrderDate, ShipName
FROM Orders
WHERE OrderID = @MyIdent

```

Again we see an explicit indication of what happened to our data:

```

Invalid Order Date
Supplied Order Date was greater than 7 days old.
The value has been reset to NULL

```

```
(1 row(s) affected)
OrderID      CustomerID      EmployeeID      OrderDate      ShipName
-----
11085        ALFKI            5              NULL           NULL
(1 row(s) affected)
```

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++, 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.

Let’s move away from our previous example (the searched CASE) for the time being (don’t worry, we’ll be back to it), and look at a simple CASE statement from a couple of different perspectives.

A Simple CASE

A simple CASE takes an expression that equates to a Boolean result. Let’s get right to an example:

```
USE Northwind
GO

SELECT TOP 10 OrderID, OrderID % 10 AS 'Last Digit', Position =
CASE OrderID % 10
    WHEN 1 THEN 'First'
    WHEN 2 THEN 'Second'
    WHEN 3 THEN 'Third'
    WHEN 4 THEN 'Fourth'
    ELSE 'Something Else'
END
FROM Orders
```

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 only gives you 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 ten, 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.

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 Northwind
GO

SELECT TOP 10 OrderID % 10 AS "Last Digit",
    ProductID,
    "How Close?" = CASE OrderID % 10
        WHEN ProductID THEN 'Exact Match!'
        WHEN ProductID - 1 THEN 'Within 1'
        WHEN ProductID + 1 THEN 'Within 1'
        ELSE 'More Than One Apart'
    END
```

```
FROM [Order Details]
WHERE ProductID < 10
ORDER BY OrderID DESC
```

Notice that we’ve used equations at every step of the way on this one, yet it still works . . .

Last Digit	ProductID	How Close?
7	8	Within 1
7	7	Exact Match!
7	6	Within 1
7	4	More Than One Apart
7	3	More Than One Apart
7	2	More Than One Apart
6	6	Exact Match!
5	2	More Than One Apart
2	2	Exact Match!
1	7	More Than One Apart
(10 row(s) affected)		

As long as the expression evaluates to a specific value that is of compatible type to the input expression, then 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:

```
USE Northwind
GO

SELECT TOP 10 OrderID % 10 AS "Last Digit",
ProductID,
"How Close?" = CASE
    WHEN (OrderID % 10) < 3 THEN 'Ends With Less Than Three'
    WHEN ProductID = 6 THEN 'ProductID is 6'
    WHEN ABS(OrderID % 10 - ProductID) <= 1 THEN 'Within 1'
    ELSE 'More Than One Apart'
END
FROM [Order Details]
WHERE ProductID < 10
ORDER BY OrderID DESC
```

This is substantially different from our simple CASE examples, but it still works:

Last Digit	ProductID	How Close?
7	8	Within 1
7	7	Within 1
7	6	ProductID is 6
7	4	More Than One Apart
7	3	More Than One Apart
7	2	More Than One Apart
6	6	ProductID is 6
5	2	More Than One Apart
2	2	Ends With Less Than Three
1	7	Ends With Less Than Three

(10 row(s) affected)

There are a couple 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 OrderID, 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.

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 \$10.13 example above), 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¢ to 75¢ should be changed to end in 75¢, and prices ending with more than 75¢ should be changed to end with 95¢. Let's 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

Chapter 12

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 Northwind
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, ProductName, UnitPrice,
    UnitPrice * @Multiplier AS "Marked Up Price", "New Price" =
    CASE WHEN FLOOR(UnitPrice * @Multiplier + .24)
        > FLOOR(UnitPrice * @Multiplier)
        THEN FLOOR(UnitPrice * @Multiplier) + .95
    WHEN FLOOR(UnitPrice * @Multiplier + .5) >
        FLOOR(UnitPrice * @Multiplier)
        THEN FLOOR(UnitPrice * @Multiplier) + .75
    ELSE FLOOR(UnitPrice * @Multiplier) + .49
    END
FROM Products
ORDER BY ProductID DESC      -- Just because the bottom's a better example
                               -- in this particular case
```

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	ProductName	UnitPrice	Marked Up Price	New Price
77	Original Frankfurter grüne Soße	13.0000	14.3000	14.4900
76	Lakkalikööri	18.0000	19.8000	19.9500
75	Rhönbräu Klosterbier	7.7500	8.5250	8.7500
74	Longlife Tofu	10.0000	11.0000	11.4900
73	Röd Kaviar	15.0000	16.5000	16.7500
72	Mozzarella di Giovanni	34.8000	38.2800	38.4900
71	Flotemysost	21.5000	23.6500	23.7500
70	Outback Lager	15.0000	16.5000	16.7500
69	Gudbrandsdalsost	36.0000	39.6000	39.7500
68	Scottish Longbreads	12.5000	13.7500	13.7500

(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.

Now, for one final example with this CASE statement, and to put something like this more into the context of sprocs, let's convert this to something the marketing department can call themselves.

In order to convert something like this to a sproc, we need to know what information is going to be changing each time we run it. In this case, the only thing that will change will be the markup percentage. That means that only the markup percent needs to be accepted as a parameter — any other variables can remain internal to the sproc.

To change this particular script then, we only need to change one variable to a parameter, add our CREATE statements, and we should be ready to go. However, we are going to make just one more change to clarify the input for the average user:

```
USE Northwind
GO
```

```
CREATE PROC spMarkupTest
    @MarkupAsPercent money
AS
    DECLARE @Multiplier money

    -- We want the end price, not the amount
    SELECT @Multiplier = @MarkupAsPercent / 100 + 1 /*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, ProductName, UnitPrice,
    UnitPrice * @Multiplier AS "Marked Up Price", "New Price" =
    CASE WHEN FLOOR(UnitPrice * @Multiplier + .24)
        > FLOOR(UnitPrice * @Multiplier)
        THEN FLOOR(UnitPrice * @Multiplier) + .95
    WHEN FLOOR(UnitPrice * @Multiplier + .5) >
        FLOOR(UnitPrice * @Multiplier)
        THEN FLOOR(UnitPrice * @Multiplier) + .75
    ELSE FLOOR(UnitPrice * @Multiplier) + .49
END
FROM Products
ORDER BY ProductID DESC -- Just because the bottom's a better example
                        -- in this particular case
```

Now, to run our sproc, we only need to make use of the EXEC command and supply a parameter:

```
EXEC spMarkupTest 10
```

Our results should be exactly as they were when the code was in script form. By putting it into sproc form, however, we:

- ❑ Simplified the use for inexperienced users
- ❑ Sped up processing time

The simplified use for the end user seems pretty obvious. They probably would be pretty intimidated if they had to look at all that code in the script—even if they only had to change just one line. Instead, they can enter in just three words—including the parameter value.

The performance boost is actually just about nothing in an interactive scenario like this case, but, rest assured, the process will run slightly faster (just milliseconds in many cases—longer in others) as a sproc—we'll look into this much further before the chapter's done.

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.

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 a.m. and write a log entry of that fact to both the SQL Server log and the Windows NT 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 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 the 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 net-send 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.

Chapter 12

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 a.m., after which execution of the code would continue with whatever the next statement was after the `WAITFOR`.

TRY/CATCH Blocks

I'm actually going to defer much of the discussion on this to our upcoming section on error handling. Still, it's important to touch on this from a control of flow point of view. `TRY/CATCH` blocks are new with SQL Server 2005, and for those of you coming from programming languages that don't have `TRY/CATCH`, I'll preface the upcoming discussion by saying they are all about handling exceptions.

For now, we'll just address them in the most simplistic of terms — when they do what they do. In short, if your code runs without any kind of exception, or an error "level" (more on those in a bit) that is 10 or below, then the code will execute according to the `TRY` block. The moment, however, that your code has an error that is above 10 (11 or higher), then it will immediately move to the first line in the `CATCH` block and proceed from there.

Note that the `CATCH` block will execute only if the error is not of a variety that immediately terminates your script. Again, we'll cover this further in our section on error handling, but some forms of errors will immediately terminate all execution of your `sproc` — in this case, that means that even your `CATCH` block will *not* be executed.

Confirming Success or Failure with Return Values

You'll see return values 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 on 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 back 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 is a 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 back from our sproc to the calling code, we 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. That is, no matter where you are in your sproc, not one single more line of code will execute after you have issued a `RETURN` statement.

By unconditionally, I don't mean that a `RETURN` statement is executed regardless of where it is in code. On the contrary, you can have many `RETURN` statements in your sproc, and they will only be executed when the normal conditional structure of your code issues the command. Once that happens however, there is no turning back.

Let's illustrate this idea of how a `RETURN` statement affects things by writing a very simple test sproc:

```
USE Northwind
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
```

OK, now we have a sproc, but we need a small script to test out a couple of things for us. What we want to see is:

- ❑ What gets printed out
- ❑ What value the RETURN statement returns

In order to capture the value of a RETURN statement, we need to assign it to a variable during our EXEC statement. For example, the following code would assign whatever the return value is to @ReturnVal:

```
EXEC @ReturnVal = spMySproc
```

Now let's put this into a more useful script to test out our sproc:

```
DECLARE @Return int

EXEC @Return = spTestReturns
SELECT @Return
```

Short but sweet — when we run it, we 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)
```

We also got back the return value for our sproc, which was zero. Notice that the value was zero even though we 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, then that means that the default return is also, in effect, “No Errors”. This has some serious dangers to it. The key point here is to 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.

Now, just for grins, let's alter that sproc to verify that we can send whatever integer value we want back as the return value:

```
USE Northwind
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 re-run 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. We don't need this section. I mean, our code never has errors, and we never run into problems, right? OK, well, now that we've had our moment of fantasy for today, let's get down to reality. Things go wrong—it's just the way that life works in the wonderful world of software engineering. Fortunately, we can do something about it. Unfortunately, you're probably not going to be happy with the tools you have. 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.

Three common error types can happen in SQL Server:

- ❑ Errors that create runtime errors and stop your code from proceeding further.
- ❑ Errors that SQL Server knows about, but that don't create runtime errors such that your code stops running. These can also be referred to as "inline" errors.
- ❑ 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 we go down a very much winding road . . .

As I write this, most SQL Server texts for 2005 are not out—but I'll go ahead and venture a guess that most beginning books will not discuss much in the way of prior versions. Indeed, I've generally avoided it as it just adds more complexity. That said, I'm very much going to touch on prior versions in this section. Why? Well, most database developers will either work with prior versions at some point in time, or, at the very least, work with code that pre-dates SQL Server 2005. In this section, that is critical because there was no formal error handler in SQL Server 2000 and earlier.

With this in mind, I'm going to give you a "slimmed down" version of how error handling used to be — if for no other reason than to help you grasp the "why they did it that way" in older code you may come across. If you're certain that you're going to be a "SQL Server 2005 code only" kinda DBA, then, by all means, feel free to skip ahead to error handling in the TRY/CATCH era.

One thing remains common between the old and new error handling models — higher level runtime errors.

It is possible to generate errors that will 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 what we had before — but we still have times where our sproc doesn't even know that something bad happened. 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 did not have an option that essentially said, "If any error happens, go run this code over in this other spot." Instead, we had to monitor for error conditions within our own code, and then decide what to do at the point we 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, let's try to insert a record into the `Order Details` table that doesn't have a corresponding record in the `Orders` table:

```
USE Northwind
GO

INSERT INTO [Order Details]
    (OrderID, ProductID, UnitPrice, Quantity, Discount)
VALUES
    (999999, 11, 10.00, 10, 0)
```

SQL Server won't perform this insert for us because there is a `FOREIGN KEY` constraint on `Order Details` that references the `PRIMARY KEY` in the `Orders` table. Since there is no record in the `Orders` table with an `OrderID` of 999999, the record we are trying to insert into `Order Details` violates that constraint and is rejected:

```
Msg 547, Level 16, State 0, Line 2
The INSERT statement conflicted with the FOREIGN KEY constraint
"FK_Order_Details_Orders". The conflict occurred in database "Northwind", table
"Orders", column 'OrderID'.
The statement has been terminated.
```

Pay attention to that error 547 up there — that's something of which we can make use.

Making Use of @@ERROR

We've already talked some about this bad boy when we were looking at scripting, but it's time to get a lot friendlier with this particular system function.

To review, @@ERROR contains the error number of the last T-SQL statement executed. If the value is zero, then no error occurred.

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.

Let's play with this just a bit using our INSERT example from before:

```
USE Northwind
GO

DECLARE    @Error    int

-- Bogus INSERT - there is no OrderID of 999999 in Northind
INSERT INTO [Order Details]
    (OrderID, ProductID, UnitPrice, Quantity, Discount)
VALUES
    (999999,11,10.00,10, 0)

-- Move our error code into safe keeping. 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 our script, and we can examine how @@ERROR is affected:

```
Msg 547, Level 16, State 0, Line 5
The INSERT statement conflicted with the FOREIGN KEY constraint
"FK_Order_Details_Orders". The conflict occurred in database "Northwind", table
"Orders", column 'OrderID'.
The statement has been terminated.

The Value of @Error is 547
The Value of @@ERROR is 0
```


Chapter 12

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 our code from executing. Indeed, the only part of that message that our 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), but rather because of the difference in scope. The @ or @@ is part of the name, so just the number of @ symbols on the front makes each one separate and distinct from the other.

Using @@ERROR in a Sproc

Let's go back to our spInsertDateValidatedOrder stored procedure that we started back when we were dealing with IF...ELSE statements. All the examples we worked with in that sproc ran just fine. Of course they did — they were well-controlled examples. However, that's not the way things work in the real world. Indeed, you never have any idea what a user is going to throw at your code. The world is littered with the carcasses of programmers who thought they had thought of everything, only to find that their users had broken something (you might say they thought of something else) within the first few minutes of operation.

We can break that sproc in no time at all by just changing one little thing in our test script:

```
USE Northwind
GO

DECLARE @MyIdent int
DECLARE @MyDate smalldatetime

SELECT @MyDate = GETDATE()

EXEC spInsertDateValidatedOrder
    @CustomerID = 'ZXXZX',
    @EmployeeID = 5,
    @OrderDate = @MyDate,
    @ShipVia = 3,
    @Freight = 5.00,
    @OrderID = @MyIdent OUTPUT

SELECT OrderID, CustomerID, EmployeeID, OrderDate, ShipName
FROM Orders
WHERE OrderID = @MyIdent
```

This seemingly simple change creates all kinds of havoc with our sproc:

```
The Time of Day in Order Date was truncated
Server: Msg 547, Level 16, State 1, Procedure spInsertDateValidatedOrder, Line 44
INSERT statement conflicted with COLUMN FOREIGN KEY constraint
'FK_Orders_Customers'. The conflict occurred in database 'Northwind', table
'Customers', column 'CustomerID'.
```

The statement has been terminated.

OrderID	CustomerID	EmployeeID	OrderDate	ShipName
-----	-----	-----	-----	-----
(0 row(s) affected)				

Our row wasn't inserted. It shouldn't have been — after all, isn't that why we put in constraints — to ensure that bad records don't get inserted into our database?

The nasty thing here is that we get a big ugly message that's almost impossible for the average person to understand. What we need to do is test the value of @@ERROR and respond accordingly.

We can do this easily using an IF . . . ELSE statement together with either @@ERROR (if we can test the value immediately and only need to test it once), or a local variable, into which we have previously moved the value of @@ERROR.

Personally, I like my code to be consistent, so I always move it into a local variable and then do all my testing with that — even when I only need to test it once. I have to admit to being in the minority on that one though. Doing this when you don't need to takes up slightly more memory (the extra variable) and requires an extra assignment statement (to move @@ERROR to your local variable). Both of these pieces of overhead are extremely small and I gladly trade them for the idea of people who read my code knowing that they are going to see the same thing done the same way every time.

In addition, it doesn't make much sense to still select out the inserted row, so we'll want to skip that part since it's irrelevant.

So let's add a couple of changes to deal with this referential integrity issue and skip the code that doesn't apply in this error situation.

```
USE Northwind
GO

ALTER PROC spInsertDateValidatedOrder
    @CustomerID    nvarchar(5),
    @EmployeeID    int,
    @OrderDate     datetime      = NULL,
    @RequiredDate  datetime      = NULL,
    @ShippedDate   datetime      = NULL,
    @ShipVia       int,
    @Freight       money,
    @ShipName      nvarchar(40) = NULL,
    @ShipAddress   nvarchar(60) = NULL,
    @ShipCity      nvarchar(15) = NULL,
    @ShipRegion    nvarchar(15) = NULL,
    @ShipPostalCode nvarchar(10) = NULL,
    @ShipCountry   nvarchar(15) = NULL,
    @OrderID       int          OUTPUT
AS
```

```
-- Declare our variables
DECLARE @Error int
```

Chapter 12

```
DECLARE    @InsertedOrderDate    smalldatetime

/* Test to see if supplied date is over seven days old, if so
** replace with NULL value
** otherwise, truncate the time to be midnight*/
IF DATEDIFF(dd, @OrderDate, GETDATE()) > 7
BEGIN
    SELECT @InsertedOrderDate = NULL
    PRINT 'Invalid Order Date'
    PRINT 'Supplied OrderDate was greater than 7 days old.'
    PRINT 'The value has been reset to NULL'
END
ELSE
BEGIN
    SELECT @InsertedOrderDate =
        CONVERT(datetime, (CONVERT(varchar, @OrderDate, 112)))
    PRINT 'The Time of Day in Order Date was truncated'
END

/* Create the new record */
INSERT INTO Orders
VALUES
(
    @CustomerID,
    @EmployeeID,
    @InsertedOrderDate,
    @RequiredDate,
    @ShippedDate,
    @ShipVia,
    @Freight,
    @ShipName,
    @ShipAddress,
    @ShipCity,
    @ShipRegion,
    @ShipPostalCode,
    @ShipCountry
)

-- Move it to our local variable and check for an error condition
SELECT @Error = @@ERROR

IF @Error != 0
BEGIN
    -- Uh, oh—something went wrong.

    IF @Error = 547
    -- The problem is a constraint violation. Print out some informational
    -- help to steer the user to the most likely problem.
    BEGIN
        PRINT 'Supplied data violates data integrity rules'
        PRINT 'Check that the supplied customer number exists'
        PRINT 'in the system and try again'
    END
END
ELSE
```

```
-- Oops, it's something we haven't anticipated, tell them that we
-- don't know, print out the error.
BEGIN
    PRINT 'An unknown error occurred. Contact your System Administrator'
    PRINT 'The error was number ' + CONVERT(varchar, @Error)
END
-- Regardless of the error, we're going to send it back to the calling
-- piece of code so it can be handled at that level if necessary.
RETURN @Error
END
```

```
/* Move the identity value from the newly inserted record into
   our output variable */
SELECT @OrderID = @@IDENTITY
```

```
RETURN
```

Now we need to run our test script again, but it's now just a little inadequate to test our sproc — we need to accept the return value so we know what happened. In addition, we have no need to run the query to return the row just inserted if the row couldn't be inserted — so we'll skip that in the event of error.

```
USE Northwind
GO
```

```
DECLARE    @MyIdent    int
DECLARE    @MyDate      smalldatetime
DECLARE    @Return      int
```

```
SELECT @MyDate = GETDATE()
```

```
EXEC @Return = spInsertDateValidatedOrder
    @CustomerID = 'ZXZXZ',
    @EmployeeID = 5,
    @OrderDate = @MyDate,
    @ShipVia = 3,
    @Freight = 5.00,
    @OrderID = @MyIdent OUTPUT
```

```
IF @Return = 0
    SELECT OrderID, CustomerID, EmployeeID, OrderDate, ShipName
    FROM Orders
    WHERE OrderID = @MyIdent
ELSE
    PRINT 'Value Returned was ' + CONVERT(varchar, @Return)
```

Realistically, not much changed — just five lines. Nonetheless, the behavior is quite a bit different when we have an error. Run this script, and we wind up with a different result than before we had our error checking:

```
The Time of Day in Order Date was truncated
Server: Msg 547, Level 16, State 1, Procedure spInsertDateValidatedOrder, Line 42
INSERT statement conflicted with COLUMN FOREIGN KEY constraint
'FK_Orders_Customers'. The conflict occurred in database 'Northwind', table
'Customers', column 'CustomerID'.
```

```
The statement has been terminated.  
Supplied data violates data integrity rules  
Check that the supplied customer number exists  
in the system and try again  
Value Returned was 547
```

We didn't have an error handler in the way most languages operate these days, but we were able to handle it nonetheless.

Handling Errors Before They Happen

Sometimes you have errors that SQL Server doesn't really have an effective way to even know about, let alone tell you about. Other times we want to prevent the errors before they happen. These we need to check for and handle ourselves.

Sticking with the main example sproc we've used for this chapter, let's address some business rules that are logical in nature, but not necessarily implemented in the database. For example, we've been allowing nulls in the database, but maybe we don't want to do that as liberally anymore. We've decided that we should no longer allow a null `OrderDate`. We still have records in there that we don't have values for, so we don't want to change over the column to disallowing nulls at the table level. What to do?

The first thing we need to take care of is editing our sproc to no longer allow `NULL` values. This seems easy enough—just remove the `NULL` default from the parameter, right? That has two problems to it:

- ❑ SQL Server will generate an error if the parameter is not supplied, but will still allow a user to explicitly supply a `NULL`.
- ❑ Even when the user fails to provide the parameter, the error information is vague.

We get around these problems by actually continuing with our `NULL` default just as it is, but this time we're testing for it. If the parameter contains a `NULL`, we then know that one was either not supplied or the value supplied was `NULL` (which we don't allow anymore)—then we act accordingly. So the question becomes, "How do I test to see if it's a `NULL` value?" Simple: just the way we did in our `WHERE` clauses in queries:

```
IF @OrderDate IS NULL  
    <abort the INSERT and print a message>
```

Let's make the modifications to our now very familiar sproc:

```
USE Northwind  
GO  
  
ALTER PROC spInsertDateValidatedOrder  
    @CustomerID      nvarchar(5),  
    @EmployeeID      int,  
    @OrderDate       datetime = NULL,  
    @RequiredDate     datetime = NULL,  
    @ShippedDate      datetime = NULL,  
    @ShipVia          int,  
    @Freight          money,  
    @ShipName         nvarchar(40) = NULL,
```

```

        @ShipAddress      nvarchar(60) = NULL,
        @ShipCity         nvarchar(15) = NULL,
        @ShipRegion       nvarchar(15) = NULL,
        @ShipPostalCode   nvarchar(10) = NULL,
        @ShipCountry      nvarchar(15) = NULL,
        @OrderID          int          OUTPUT

AS

-- Declare our variables
DECLARE    @Error          int
DECLARE    @InsertedOrderDate  smalldatetime

/* Here we're going to declare our constants. SQL Server doesn't really
** have constants in the classic sense, but I just use a standard
** variable in their place. These help your code be more readable
**--particularly when you match them up with a constant list in your
** client. */

DECLARE    @INVALIDDATE    int

/* Now that the constants are declared, we need to initialize them.
** Notice that SQL Server ignores the white space in between the
** variable and the "=" sign. Why I put in the spacing would be more
** obvious if we had several such constants--the constant values
** would line up nicely for readability
**/
SELECT @INVALIDDATE = -1000

/* Test to see if supplied date is over seven days old, if so
** it is no longer valid. Also test for NULL values.
** If either case is true, then terminate sproc with error
** message printed out. */
IF DATEDIFF(dd, @OrderDate, GETDATE()) > 7 OR @OrderDate IS NULL
BEGIN
    PRINT 'Invalid Order Date'
    PRINT 'Supplied Order Date was greater than 7 days old '
    PRINT 'or was NULL. Correct the date and resubmit.'
    RETURN @INVALIDDATE
END

-- We made it this far, so it must be OK to go on with things.
SELECT @InsertedOrderDate =
    CONVERT(datetime, (CONVERT(varchar, @OrderDate, 112)))
    PRINT 'The Time of Day in Order Date was truncated'

/* Create the new record */
INSERT INTO Orders
VALUES
(
    @CustomerID,
    @EmployeeID,
    @InsertedOrderDate,
    @RequiredDate,
    @ShippedDate,

```

```
@ShipVia,
@Freight,
@ShipName,
@ShipAddress,
@ShipCity,
@ShipRegion,
@ShipPostalCode,
@ShipCountry
)

-- Move it to our local variable, and check for an error condition
SELECT @Error = @@ERROR

IF @Error != 0
BEGIN
    -- Uh, oh--something went wrong.

    IF @Error = 547
    -- The problem is a constraint violation. Print out some informational
    -- help to steer the user to the most likely problem.
    BEGIN
        PRINT 'Supplied data violates data integrity rules'
        PRINT 'Check that the supplied customer number exists'
        PRINT 'in the system and try again'
    END
    ELSE
    -- Oops, it's something we haven't anticipated, tell them here that we
    -- don't know, print out the error.
    BEGIN
        PRINT 'An unknown error occurred. Contact your System Administrator'
        PRINT 'The error was number ' + CONVERT(varchar, @Error)
    END
    -- Regardless of the error, we're going to send it back to the calling
    -- piece of code so it can be handled at that level if necessary.
    RETURN @Error
END

/* Move the identity value from the newly inserted record into
   our output variable */
SELECT @OrderID = @@IDENTITY

RETURN
```

We're going to want to test this a couple of different ways. First, we need to put back in a valid customer number, and then we need to run it. Assuming it succeeds, then we can move on to supplying an unacceptable date:

```
USE Northwind
GO

DECLARE @MyIdent int
DECLARE @MyDate smalldatetime
```

```

DECLARE    @Return          int

SELECT @MyDate = '1/1/1999'

EXEC @Return = spInsertDateValidatedOrder
    @CustomerID = 'ALFKI',
    @EmployeeID = 5,
    @OrderDate = @MyDate,
    @ShipVia = 3,
    @Freight = 5.00,
    @OrderID = @MyIdent OUTPUT

IF @Return = 0
    SELECT OrderID, CustomerID, EmployeeID, OrderDate, ShipName
    FROM Orders
    WHERE OrderID = @MyIdent
ELSE
    PRINT 'Value Returned was ' + CONVERT(varchar, @Return)

```

This time, when we run it, we get an error message:

```

Invalid Order Date
Supplied Order Date was greater than 7 days old
or was NULL. Correct the date and resubmit.
Value Returned was -1000

```

Note that this wasn't a SQL Server error — as far as SQL Server's concerned, everything about life is just fine. What's nice though, is that, were we using a client program (say one you wrote in VB, C++, or some other language), we would be able to track the -1000 against a known constant and send a very specific message to the end user.

Manually Raising Errors

Sometimes we have errors that SQL Server doesn't really know about, but we wish it did. For example, perhaps in our previous example we don't want to return -1000. Instead, we'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, we make use of 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 what message is sent out 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 will include any messages you've manually added to your system using the `sp_addmessage` stored procedure or through the 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`. For example:

```
RAISERROR ('Hi there, I'm an error', 1, 1)
```

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 we 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

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 essentially being informational (severities 1–18), to being considered as system level (19–25), and even catastrophic (20–25). If you raise an error of severity 19 or higher (system level), then the `WITH LOG` option must also be specified. 20 and higher will automatically terminate the users' connections (they *hate* that!).

So, let's 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. They fall into six major groupings:

1–9	Purely informational only, but will return the specific error code in the message information. No matter what you set the state (discussed next) to in your <code>RAISERROR</code> , it will wind up coming out with the same value as the error number (don't ask me why — it just does).
10	Also informational, but will not raise an error in the client and will not provide any specific error information other than the error text.
11–16	These terminate execution of the procedure and raise an error at the client. From this point forward, the state is shown to be whatever value you set it to.
17	Usually, only SQL Server should use this severity. Basically, it indicates that SQL Server has run out of resources (for example <code>tempdb</code> was full) and can't complete the request.
18–19	Both of these are severe errors, and imply that the underlying cause requires system administrator attention. With 19, the <code>WITH LOG</code> option is required, and the event will show up in the NT or Win2K Event Log if you are using that OS family.
20–25	Your world has just caved in — so has the user's connection. Essentially, this is a fatal error. The connection is terminated. As with 19, you must use the <code>WITH LOG</code> option, and a message will, if applicable, show up in the Event Log.

State

State is an ad hoc value. It is 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 of what some of these mean. I'm told that, if you make a tech support call to MS, they are likely to ask and make use of this state information.

Error Arguments

Some predefined errors will accept arguments. These allow the error to be somewhat more dynamic in nature by changing to the specific nature of the error. You can also format your error messages to accept arguments.

When you want to make use of dynamic information in what is otherwise a static error message, you need to format the fixed portion of your message such that it leaves room for the parameterized section of the message. This is done using placeholders. If you're coming from the C or C++ world, then you'll recognize the parameter placeholders immediately — they are very similar to the `printf` command arguments. If you're not from the C world, these may seem a little odd to you. All of the placeholders start with the `%` sign, and are then coded for what kind of information you'll be passing to them:

Placeholder Type Indicator	Type of Value
D	Signed integer — note that Books Online also indicates that <code>i</code> is an OK 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 — makes a difference only when you supply a fixed width.
+ (plus sign)	Indicate the positive or negative nature if the parameter is a signed numeric type.

Table continued on following page

Flag	What It Does
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)	Only applies to octal and hex values. Tells SQL Server to use the appropriate prefix (0 or 0x) depending on whether it is octal or hex.
' '	Pad 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 how much space we want to hold for the parameterized value. You can also specify a *, in which case SQL Server will automatically determine the width depending on 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.

To 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)
```

If you execute this, you get back something that looks a little different from what’s in the quotes:

```
Msg 50000, Level 1, State 50000
This is a sample parameterized string, along with a zero padding and a sign+000012121
```

The extra values supplied were inserted, in order, into our placeholders, with the final value being reformatted as specified.

WITH <option>

There are currently three options that you can mix and match 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 NT application log (the latter applies to installations on NT only). This option is required with severity levels that are 19 or higher.

WITH SETERROR

By default, a RAISERROR command does not 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

We can make use of a special system stored procedure to add messages to the system. The spoc is called `sp_addmessage`, and the syntax looks like this:

```
sp_addmessage [@msgnum =] <msg 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 in `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 both the SQL Server error log and the NT application log when raised (the latter only when running under NT). 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, then 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 a generic handling of some or all of the errors. In short, if you're going to be building multiple SQL Server apps 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 we create ad hoc messages using `RAISERROR`.

As an example, let's add our own custom message that tells the user 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 spoc 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 will 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.

It's also worth noting that you can add and delete custom messages using Enterprise Manager (right-click on a server, and then go to All Tasks | Manage SQL Server messages). While this is quick and easy, it makes it more problematic to create and test the scripts I recommend in the paragraph above. In short, I don't recommend its use.

Removing an Existing Custom Message

To get rid of the custom message, use:

```
sp_dropmessage <msg num>
```

Putting Our Error Trap to Use

Now it's time to put all the different pieces we've been talking about to use at once.

First, if you tried out the `sp_dropmessage` on our new error 60000 — quit that! Add the message back so we can make use of it in this example.

What we want to do is take our sproc to the next level up. We're going to modify our sproc again so that it takes advantage of the new error features we know about. When we're done, we'll be able to generate a trappable runtime error in our client so we can take appropriate action at that end.

All we need to do is change our `PRINT` statement to have a `RAISERROR`:

```
USE Northwind
GO

ALTER PROC spInsertDateValidatedOrder
    @CustomerID      nvarchar(5),
    @EmployeeID       int,
    @OrderDate        datetime      = NULL,
    @RequiredDate     datetime      = NULL,
    @ShippedDate      datetime      = NULL,
    @ShipVia          int,
    @Freight          money,
    @ShipName         nvarchar(40) = NULL,
    @ShipAddress      nvarchar(60) = NULL,
    @ShipCity         nvarchar(15) = NULL,
    @ShipRegion       nvarchar(15) = NULL,
    @ShipPostalCode   nvarchar(10) = NULL,
    @ShipCountry      nvarchar(15) = NULL,
    @OrderID          int          OUTPUT
AS

-- Declare our variables
DECLARE @Error      int
DECLARE @BadDate    varchar(12)
DECLARE @InsertedOrderDate smalldatetime

/* Test to see if supplied date is over seven days old, if so
** it is no longer valid. Also test for null values.
** If either case is true, then terminate sproc with error
** message printed out. */
IF DATEDIFF(dd, @OrderDate, GETDATE()) > 7 OR @OrderDate IS NULL
BEGIN
    --RAISERROR doesn't have a date data type, so convert it first
    SELECT @BadDate = CONVERT(varchar, @OrderDate)
    RAISERROR (60000,1,1, @BadDate) WITH SETERROR
    RETURN @@ERROR
END

-- We made it this far, so it must be OK to go on with things.
SELECT @InsertedOrderDate =
    CONVERT(datetime, (CONVERT(varchar, @OrderDate, 112)))
    PRINT 'The Time of Day in Order Date was truncated'
/* Create the new record */
INSERT INTO Orders
VALUES
(
```

```
@CustomerID,
@EmployeeID,
@InsertedOrderDate,
@RequiredDate,
@ShippedDate,
@ShipVia,
@Freight,
@ShipName,
@ShipAddress,
@ShipCity,
@ShipRegion,
@ShipPostalCode,
@ShipCountry
)

-- Move it to our local variable, and check for an error condition
SELECT @Error = @@ERROR

IF @Error != 0
BEGIN
    -- Uh, Oh--something went wrong.

    IF @Error = 547
    -- The problem is a constraint violation. Print out some informational
    -- help to steer the user to the most likely problem.
    BEGIN
        PRINT 'Supplied data violates data integrity rules'
        PRINT 'Check that the supplied customer number exists'
        PRINT 'in the system and try again'
    END
    ELSE
    -- Oops, it's something we haven't anticipated, tell them that we
    -- don't know, print out the error.
    BEGIN
        PRINT 'An unknown error occurred. Contact your System Administrator'
        PRINT 'The error was number ' + CONVERT(varchar, @Error)
    END
    -- Regardless of the error, we're going to send it back to the calling
    -- piece of code so it can be handled at that level if necessary.
    RETURN @Error
END

END

/* Move the identity value from the newly inserted record into
   our output variable */
SELECT @OrderID = @@IDENTITY

RETURN
```

What a Sproc Offers

Now that we've spent some time looking at how to build a sproc, we probably ought to ask the question as to why to 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. The primary benefits of sprocs include:

- ❑ Making processes that require procedural action callable
- ❑ Security
- ❑ Performance

Creating Callable Processes

As I've already indicated, a sproc is something of a script that is stored in the database. The nice thing is that, because it is a database object, we 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*). For SQL Server 2005, you can nest up to 32 levels deep. This gives you the capability of reusing separate sprocs much as you would make use of 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. As an example, let's create a mini sproc to perform the same function as the test script that we've been using for most of this chapter:

```
USE Northwind
GO

CREATE PROC spTestInsert
    @MyDate    smalldatetime
AS
DECLARE    @MyIdent    int
DECLARE    @Return      int

EXEC @Return = spInsertDateValidatedOrder
    @CustomerID = 'ALFKI',
    @EmployeeID = 5,
    @OrderDate  = @MyDate,
    @ShipVia    = 3,
    @Freight    = 5.00,
    @OrderID    = @MyIdent OUTPUT

IF @Return = 0
    SELECT OrderID, CustomerID, EmployeeID, OrderDate, ShipName
    FROM Orders
    WHERE OrderID = @MyIdent
ELSE
    PRINT 'Error Returned was ' + CONVERT(varchar, @Return)
```

Now just call the sproc supplying a good date, then a bad date (to test the error handling). First the good date:

```
DECLARE @Today smalldatetime

SELECT @Today = GETDATE()

EXEC spTestInsert
    @MyDate = @Today
```

Using today's date gets what we expect:


```
The Time of Day in Order Date was truncated
(1 row(s) affected)
```

OrderID	CustomerID	EmployeeID	OrderDate	ShipName
-----	-----	-----	-----	-----
11097	ALFKI	5	2000-09-18 00:00:00.000	NULL

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

Then a bad date:

```
EXEC spTestInsert '1/1/2004'
```

Again, this yields us what we expect—in this case an error message:

```
Msg 18054, Level 16, State 1, Procedure spInsertDateValidatedOrder, Line 33
Error 60000, severity 1, state 1 was raised, but no message with that error number
was found in sys.messages. If error is larger than 50000, make sure the user-
defined message is added using sp_addmessage.
Error Returned was 60000
```

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. Much like views, we 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 we grant someone authority to execute a sproc that returns all the records in the `Customers` table, but not access to the actual `Customers` table, then the user will still be able to get data out of the `Customers` table provided they do it by using the sproc (trying to access the table directly won't work).

What can be really handy here is that we can give someone access to modify data through the sproc, but then only give them read access to the underlying table. They will be able to modify data in the table provided that they do it through your sproc (which will likely be enforcing some business rules). They can then hook directly up to your SQL Server using Excel, Access, or whatever to build their own 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 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, then consider using replication or backup and restores to create a completely separate copy of the database for them to use. This will help insure you 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, however, 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 we create a sproc, the process works something like what you see in Figure 12-1.

We start by running our `CREATE PROC` procedure. This parses the query to make sure that the code should actually run. The one difference versus running the script directly is that the `CREATE PROC` command can make use of what's called *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.

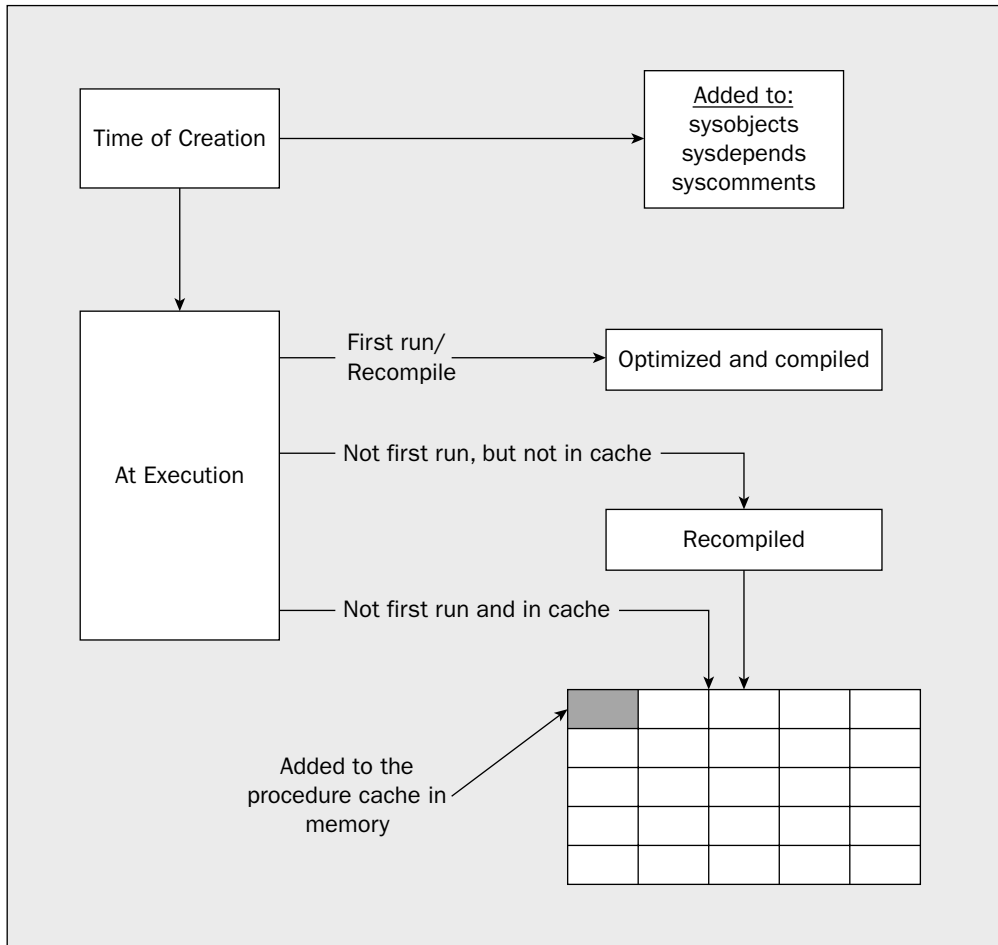


Figure 12-1

After the sproc has been created, it sits in wait 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 we run our sproc will, unless we specify otherwise using the `WITH RECOMPILE` option, use that cached query plan rather than creating a new one. 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 (1 second is still 100 percent faster than 2 seconds). The difference can become even more extreme when we have the need to make several calls or when we are in a looping situation.

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 is what saves the sproc time, but it’s a double-edged sword. If our query is dynamic in nature (the query is built up as it goes using the `EXEC` command), then 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 us mix and match several criteria for a search. For example, let’s say that we wanted to add a sproc to the Northwind database that would support a web page that allows users to search for an order based on:

- ☐ Customer number
- ☐ 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 we 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 we execute our sproc, it is 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 we ran the sproc (and 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 the sproc ran. In short, the query performance is really going to suffer.

Using the `WITH RECOMPILE` Option

We 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 us get around this issue of not using the right query plan because we’re certain that a new plan was created just for this run. To do this, we make use of the `WITH RECOMPILE` option, which can be included in two different ways.

First, we can include the `WITH RECOMPILE` at runtime. We simply include it with our execution script:

```
EXEC spTestInsert '1/1/2004'  
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 we’ve executed the sproc using the `WITH RECOMPILE` option.

We can also choose to make things more permanent by including the `WITH RECOMPILE` option right within the sproc. If we do things this way, we add the `WITH RECOMPILE` option immediately before our `AS` statement in our `CREATE PROC` or `ALTER PROC` statements.

If we create our sproc with this option, then the sproc will be recompiled each time that it runs, regardless of other options chosen at run time.

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—when you hit those times where basic T-SQL and the other features of SQL Server just wouldn’t give you what you needed.

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.

For purposes of this book, I’ll merely say that SQL Server does allow for the idea of externally written code that runs as a .DLL to SQL Server. XPs are created using some form of low-level programming language. Currently, the only languages actively supported are C and C++.

A Brief Look at Recursion

Recursion is one of those things that aren’t used very often in programming. Still, it’s also one of those things for which, 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 all 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 now—it’s just that it’s an example that can be understood by just about anyone, so here we go.

So what is that classic example? Factorials. For those who have had a while since math class (or their last recursion discussion), a factorial is the value you get when you take a number and multiply it successively

Chapter 12

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 $5*4*3*2*1$.

So, let's 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
```

When you run this CREATE script, you will wind up with an informational message that indicates that:

Cannot add rows to sysdepends for the current stored procedure because it depends on the missing object spFactorial. The stored procedure will still be created.

Whenever SQL Server creates objects, it stores away dependency information so it knows which objects are dependent on what other objects. In this case, our sproc is dependent upon itself—but how can SQL Server set the dependency information on a sproc that doesn't exist yet? It's something of a "What came first, the chicken or the egg" kind of thing. For the most part, this is informational and not really something to worry about.

So, what we're doing is accepting a value in (that's the value we want a factorial of), and providing a value back out (the factorial value we've computed). The surprising part is that our sproc does not, 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 will deal with just one number's worth, and then again call itself. This can go on and on up to a limit of 32 levels of recursion. Once SQL Server gets 32 levels deep, it will raise an error and end processing.

Note that any calls into .NET assemblies count as an extra level in your recursion count, but anything you do within those assemblies does not count against the recursion limit.

Let's try out our 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 us 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 our factorial grows too large for the int (or even bigint) datatype
- ❑ 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 our sproc. This time, we'll determine the *triangular* of the number. This is very similar to finding the factorial, except that we use addition rather than multiplication. Therefore, 5 triangular is just 15 (5+4+3+2+1). Let's create a new sproc to test this one out — 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
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, we only need to change our sproc call and the PRINT text for our 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 our 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, then come all the way back out of the call stack, then 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.

Debugging

Real-live debugging tools first made an appearance in SQL Server in SQL Server 2000. Much like SQL Server 2005, you needed your settings to be just perfect and several starts to align in order to get it working—once it did, it was wonderful.

The debugging effort for SQL Server 2005 is highly integrated with Visual Studio, and really does work fairly well.

I'm not going to kid you—the debugging tools are 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 once you get it working.

Setting Up SQL Server for Debugging

Depending on the nature of your installation, you may have to do absolutely nothing in order to get debugging working. If, however, you took the default path and installed your SQL Server to run using the LocalSystem account, then debugging will either not work at all. The upshot of this is that, if you want to use debugging, then you really need to configure the SQL Server service to run using an actual user account—specifically, one with admin access to the box the SQL Server is running on.

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 we 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 direction anymore, so pay attention to the steps we’ll have to walk through in order to find it and you’ll see what I mean.

OK, to get the Debugger going, you’ll need to start up Visual Studio and create a project of any type that includes data sources by default (Integration Services is an example of a project type that is installed with SQL Server and includes data sources as a project node). Go to the Server Explorer (found under the View menu), and right-click Data Sources; then select New Data Source (if you don’t have one already). Add in the connection information in the Add Connection dialog, as shown in Figure 12-2.

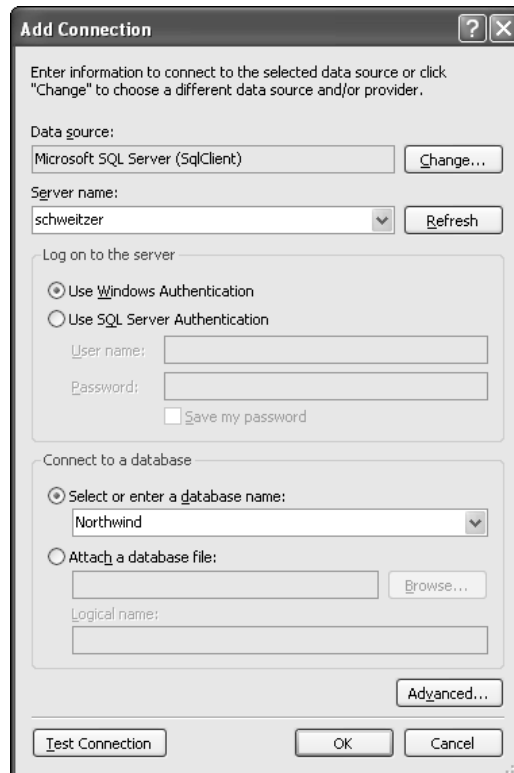


Figure 12-2

Chapter 12

Next, we need to navigate to the sproc (or UDF) that we want to debug and right-click. In our case, we want to navigate to the `spTriangular` stored procedure that we created in the last section, and right-click on it — then choose **Step Into Stored Procedure**, as shown in Figure 12-3.

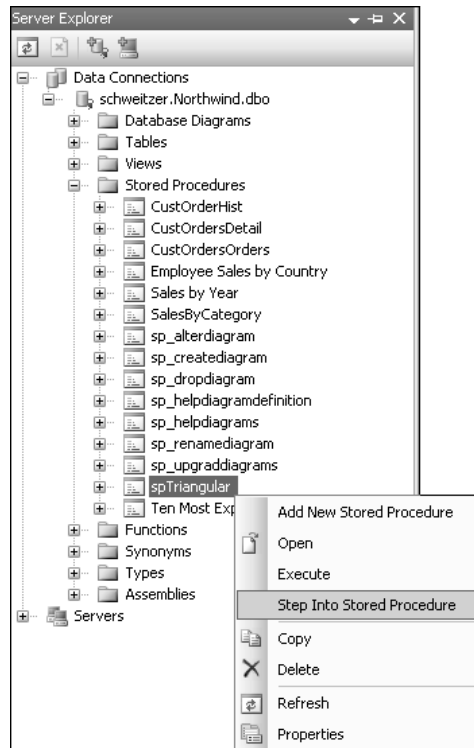


Figure 12-3

This brings up the Run Stored Procedure dialog, and prompts us to fill in the required information for the parameters our stored procedure has, much like Figure 12-4.

We need to set each non-optional parameter's value before the sproc can run. For the `@ValueIn`, we'll just set it to 3 — that will allow us to recurse just a bit and let us look into a few extra features. For the `@ValueOut`, let's use the `Set to null` option, as shown in Figure 12-5.

Then just click OK. The results are shown in Figure 12-6.

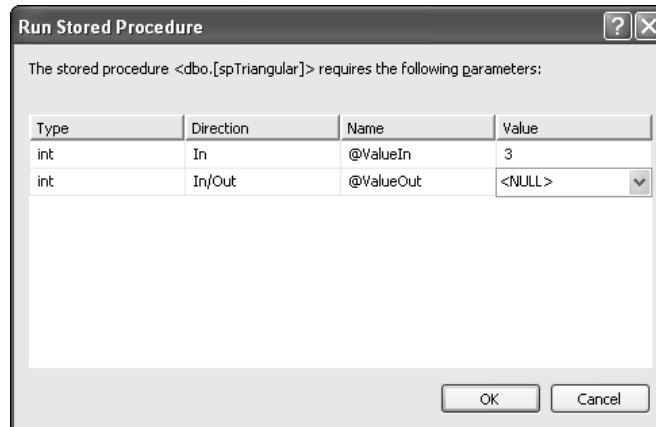


Figure 12-4

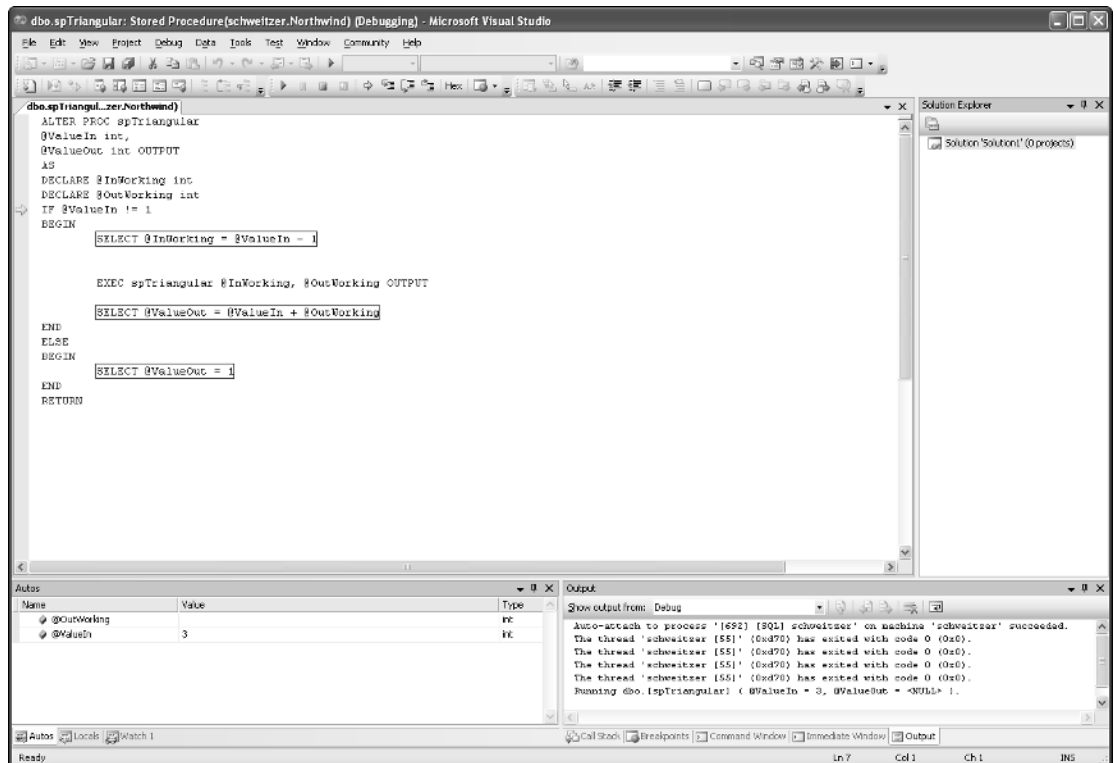


Figure 12-5

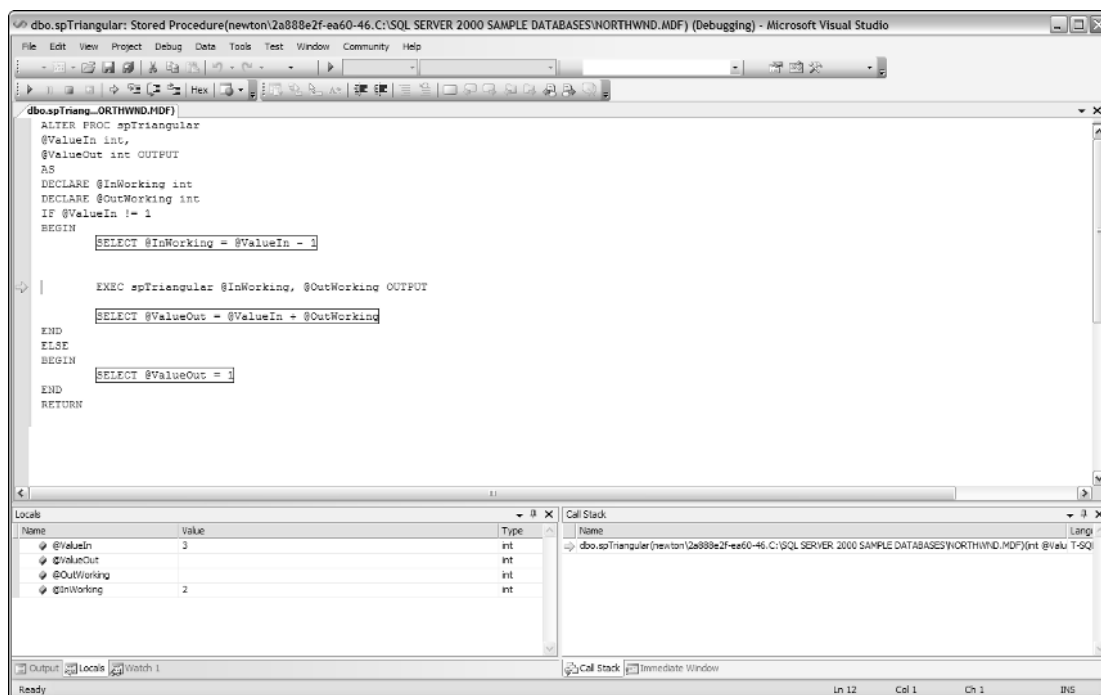


Figure 12-6

Parts of the Debugger

Several things are worth noticing when the Debugger window first comes up:

- ❑ The yellow arrow on the left indicates the *current row* — this is the next line of code that will be executed if we do a “go” or we start stepping through the code.
- ❑ There are icons at the top to indicate our different options, including:
 - ❑ **Continue:** This will run 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 us to the next statement that is at the same level in the call stack. If you are not calling another sproc or a UDF, then this command will act just like a **Step Into**. If, however, you are calling another sproc or a UDF, then a **Step Over** will take 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, we will keep running until we reach the same level as whatever code called the level we are currently at.
- ❑ **Run To Cursor:** This works pretty much like the combination of a breakpoint and a Go. When this choice is made, the code will start running and keep 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 if 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 does not get executed).
- ❑ **Restart:** This does exactly what it says it does. It sets the parameters back 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 here!” 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 of what we’ll call “status” windows; let’s go through a few of the more important of these.

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 currently 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 datatype

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

Works much as it does in any modern debugger. You can set up variables here that you want to keep track of and even trigger break points based on a change in value of watched variables.

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

Much as this one sounds, 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 Once It's Started

Now that we have the preliminaries out of the way and the `Debugger` window up, we're ready to start walking through our code.

The first executable line of our sproc is the `IF` statement, so that's the line that is current when the `Debugger` starts up. You should notice that none of our variables has had any values set in it yet except for the `@ValueIn` that we passed in as a parameter to the sproc—it has the value of 3 that we passed in when we filled out the `Debug Procedure` dialog earlier.

Step forward one line by pressing `F11` or using the `Step Into` icon or menu choice.

Since the value of `@ValueIn` is indeed not equal to 1, we step into the `BEGIN . . . END` block specified by our `IF` statement. Specifically, we move to our `SELECT` statement that initializes the `@InWorking` parameter. As we'll see later, if the value of `@ValueIn` had indeed been one, we would have immediately dropped down to our `ELSE` statement.

Again, step forward one line by pressing `F11` or using the `Step Into` icon or menu choice, as shown in Figure 12-7.

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 our `SELECT` statement. Also notice that our `Callstack` window only has the current instance of our sproc in it—since we haven't stepped down into our nested versions of the sproc yet, we only see one instance.

Now go ahead and step into our next statement. Since this is the execution of a sproc, we're going to see a number of different things change in our `Debugger` window, shown in Figure 12-8.

Notice that it *appears* that our arrow that indicates the current statement jumped back up to the `IF` statement. Why? Well, this is a new instance of our sproc. We can tell this based on our `Callstack` window—notice that it now has two instances of our 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 we passed in from the outer instance of the sproc.

If you want to see the value of variables in the scope of the outer instance of the sproc, just double-click on that instance's line in the `Callstack` window (the one with the green arrow) and you'll see several things change again, as shown in Figure 12-9.

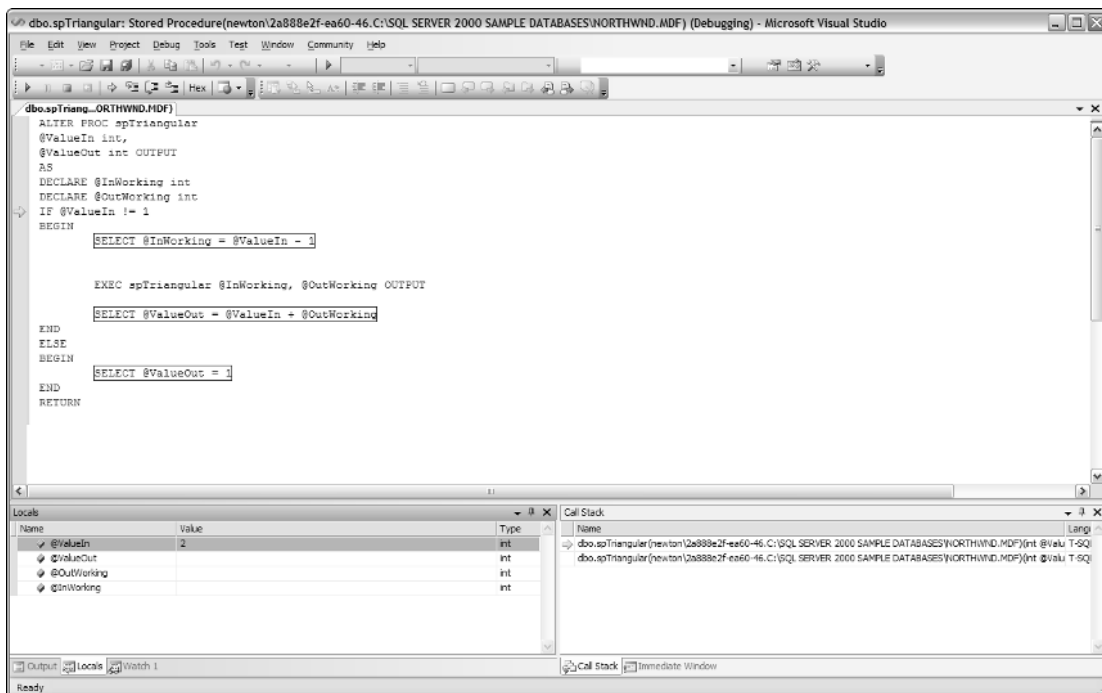


Figure 12-7

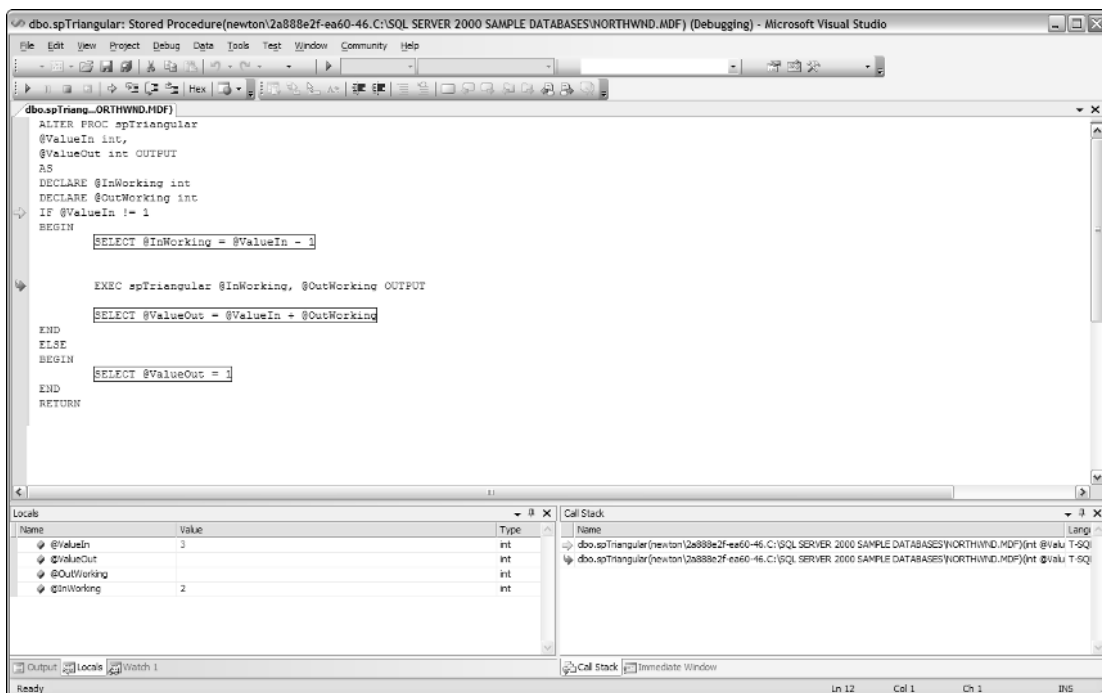


Figure 12-8

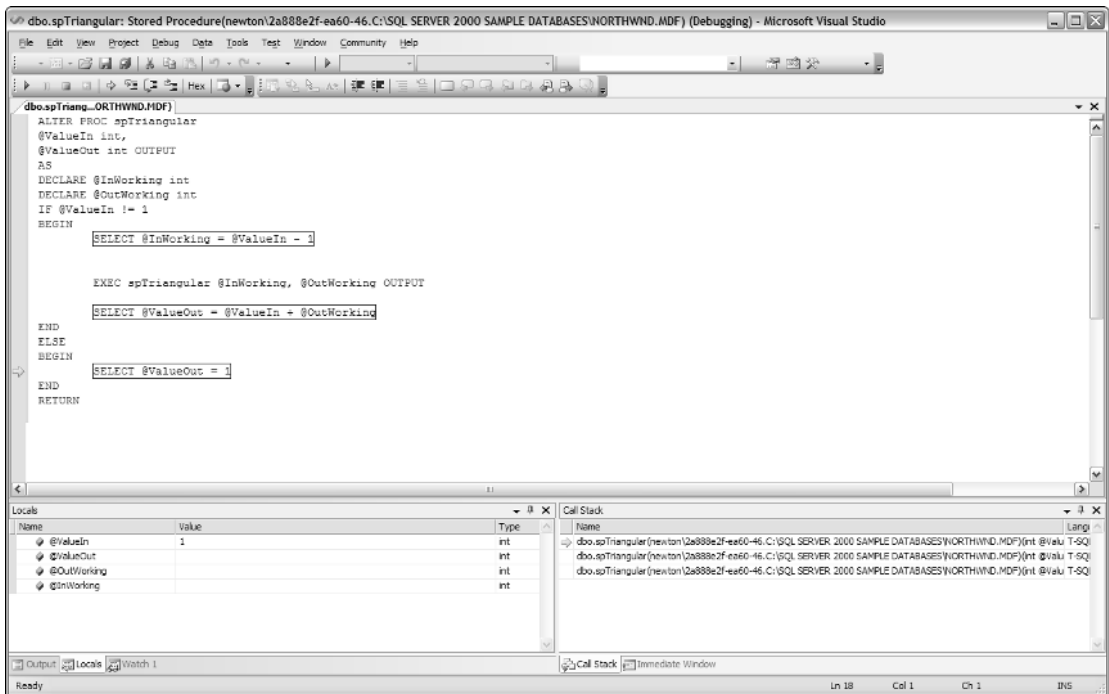


Figure 12-9

There are two things to notice here. First, the values of our variables have changed back to those in the scope of the outer (and currently selected) instance of the sproc. Second, the icon for our 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 is not the current line in the overall callstack.

Go back to the current instance by clicking on 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 our third instance of the sproc. Notice that our callstack has become three deep, and that the values of our variables and parameters in the Locals window have changed again. Last, but not least, notice that this time our @ValueIn parameter has a value of 1.

Step into the code one more time, and you'll see a slight change in behavior. This time, since the value in @ValueIn is indeed equal to 1, we move into the BEGIN . . . END block defined with our ELSE statement, as shown in Figure 12-10.

Since we've reached the bottom, we're ready to start going back up the callstack.

Notice that our callstack is back to only two levels. Also, notice that our output parameter (@OutWorking) has been appropriately set.

This time, let's do something different and do a Step Out (Shift+F11). If you're not careful, it will look like absolutely nothing has changed.

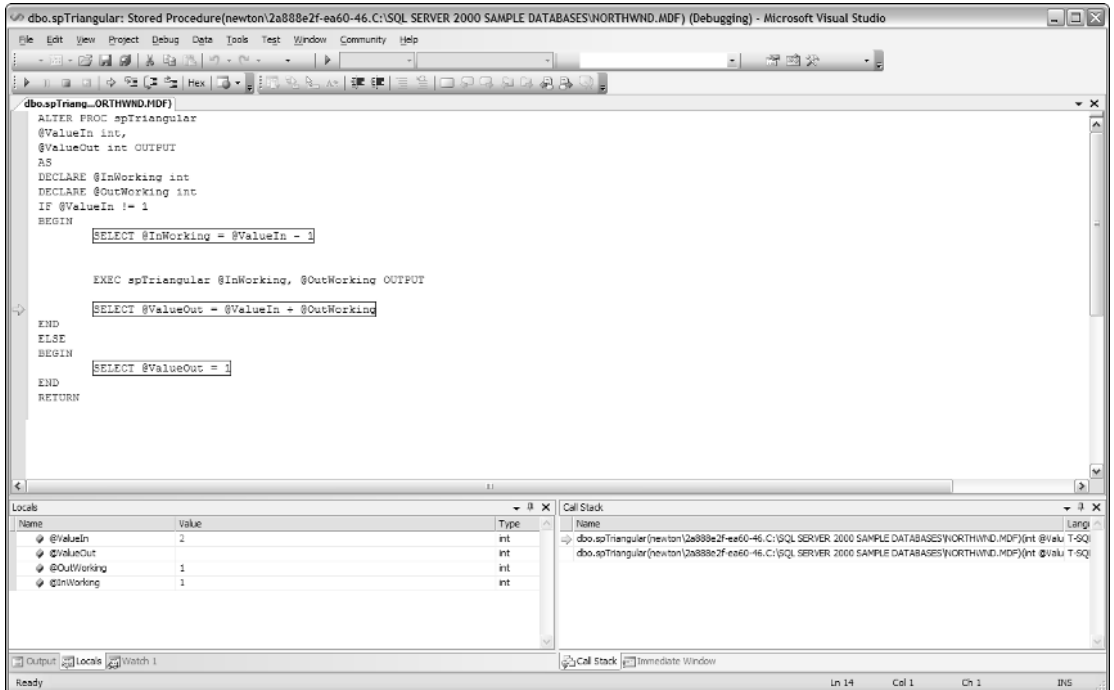


Figure 12-10

In this case, to use the old cliché, looks are deceiving. Again, notice the change in the Callstack window and in the values in the Locals window — we stepped *out* of what was then the current instance of the sproc and moved up a level in the callstack. If you now keep stepping into the code (*F11*), then our sproc has finished running and we'll see the final version of our status windows and their respective finishing values. A *big* word of caution here! If you want to be able to see the truly final values (such as an output parameter being set), make *sure* that you use the *Step Into* option to execute the last line of code.

If you use an option that executes several lines at once, such as a *Go* or *Step Out*, all you will get is the output window without any final variable information.

A workaround is to place a break point on the last point at which you expect to perform a *RETURN* in the outermost instance of your sproc. That way, you can run in whatever debug mode you want, but still have execution halt in the end so you can inspect your final variables.

So, you should now be able to see how the Debugger can be very handy indeed.

.NET Assemblies

Because of just how wide open of a topic these are, as well as their potential to add exceptional complexity to your database, these are largely considered out of scope for this title save for one thing — letting you know they are there.

.NET assemblies can be associated with your system and utilized to provide the power behind truly complex operations. You could, just as an example, use a .NET assemble in a user-defined function to provide data from a external datasource (perhaps one that has to be called on the fly, such as news feeds or stock quotes) even though the structure and complex communications required would have ruled out such a function in prior versions.

Without going into too much detail on them for now, let's look at the syntax for adding an assembly to your database:

```
CREATE ASSEMBLY <assembly name>
AUTHORIZATION <owner name>
FROM <path to assembly>
WITH PERMISSION_SET = [SAFE | EXTERNAL_ACCESS | UNSAFE]
```

The `CREATE ASSEMBLY` part of things works as pretty much all our `CREATE` statements have — it indicates the type of object being created and the object name.

Then comes the `AUTHORIZATION` — this allows you to set a context that the assembly is always to run under. That is, if it has tables it needs to access, how you set the user or rolename in `AUTHORIZATION` will determine whether it can access those tables or not.

After that, we go to the `FROM` clause. This is essentially the path to your assembly along with the manifest for that assembly.

Finally, we have `WITH PERMISSION_SET`. This has three options:

- ❑ **SAFE:** This one is, at the risk of sounding obvious, well . . . safe. It restricts the assembly from accessing anything that is external to SQL Server. Things like files or the network are not available to the assembly.
- ❑ **EXTERNAL_ACCESS:** This allows external access such as files or the network, but requires that the assembly still run as managed code.
- ❑ **UNSAFE:** This one is, at the risk of again sounding obvious — unsafe. It allows your assembly not only to access external system objects, but also to run unmanaged code.

I cannot stress enough the risks you are taking when running .NET assemblies in *anything* other than `SAFE` mode. Even in `EXTERNAL_ACCESS` mode you are allowing the users of your system to access your network, files, or other external resources is what is essentially an aliased mode — that is, they may be able to get at things that you would rather they not get at, and they will be aliased on your network to whatever your SQL Server login is while they are making those accesses. Be very, very careful with this stuff.

.NET assemblies will be discussed extensively in *Professional SQL Server 2005 Programming*.

Summary

Wow! That's a lot to have to take in for one chapter. Still, this is among the most important chapters in the book in terms of being able to function as a developer in SQL Server.

Sprocs are the backbone of code in SQL Server. We can create reusable code, and get improved performance and flexibility at the same time. We can use a variety of programming constructs that you might be familiar with from other languages, but sprocs aren't meant for everything.

Pros to sprocs include:

- ☐ Usually better performance
- ☐ Possible use as a security insulation layer (control how a database is accessed and updated)
- ☐ Reusable code
- ☐ Compartmentalization of code (can encapsulate business logic)
- ☐ Flexible execution depending on dynamics established at runtime

Cons to sprocs include:

- ☐ Not portable across platforms (Oracle, for example has a completely different kind of implementation of sprocs)
- ☐ May get locked into the wrong execution plan in some circumstances (actually hurting performance)

Sprocs are not the solution to everything, but they are still the cornerstones of SQL Server programming. In the next chapter, we'll take a look at the sprocs brand new, and very closely related cousin — the UDF.

Exercises

1. Write a simple stored procedure that returns the desired Customer record from the Northwind database given a parameter of the CustomerID.
2. Write a stored procedure that accepts a Territory ID, Territory Description, and Region ID and inserts them as new row in the Territories table in Northwind.
3. Alter the procedure you created in #2 to pre-check for the existence of the foreign key (RegionID) before attempting the insert. If the RegionID doesn't exist, throw an error with the error text "RegionID is not valid. Please check your RegionID and try again."
4. Alter the procedure you created in #2 to handle the exception after the fact when a RegionID doesn't exist. Trap all other errors and provide the generic error message: "An unhandled exception has occurred. Contact your system administrator."

13

User Defined Functions

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 under-utilized 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 “umph” to them. One of the best things about them from your point of view is, provided you’ve done the book in order, you already know most of what you need to write them. They are actually very, very similar to stored procedures — they just have certain behaviors and capabilities about them that set them apart and make them *the* answer in many situations.

In this chapter, we’re not only going to introduce what UDFs are, but we’re also going to take a look at the different types of UDFs, how they vary from stored procedures, and, of course, what kind of situations we might want to use them in. Finally, we’ll take a quick look at how you can use .NET to expand on their power.

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.

With a sproc, you can pass parameters in, and also get values in parameters passed back out. You can return a value, but that value is really intended to indicate success or failure rather than return data. You can also return result sets, but you can’t really use those result sets in a query without first inserting them into some kind of table (usually a temporary table) to work with them further.

With a UDF, however, you can pass parameters *in*, but not out. Instead, the concept of output parameters has been replaced with a much more robust return value. As with system functions, you can return a scalar value — what’s particularly nice, however, is that this value is not limited to just the integer datatype as it would be for a sproc. Instead, you can return most SQL Server datatypes (more on this in the next section).

As they like to say in late-night television commercials: “But Wait! There’s more!” The “more” is that you are actually not just limited to returning scalar values — you can also return tables. This is wildly powerful, and we’ll look into this fully later in the chapter.

So, to summarize, we have 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 datatype 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 each type individually.

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.

As I indicated earlier, one of the truly great things about a UDF is that you are not limited to an integer for a return value — instead, it can be of any valid SQL Server datatype (including user-defined datatypes!), 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:

- ❑ 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 to get our feet wet on the whole idea of how we might utilize them differently from a sproc. I'm not kidding when I say this is a simple one from a code point of view, but I think you'll see how it illustrates my sprocs versus UDFs point.

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 our 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
```

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 just 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 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).

Chapter 13

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 spoc. 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 stand-alone 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 you might expect, there is a lot more to UDFs than just readability though. You can embed queries in them and use them as an encapsulation method for subqueries. Almost anything you can do procedurally that returns a discrete value could also be encapsulated in a UDF and used inline with your queries.

Let's take a look at a very simple subquery example. The subquery version looks like this:

```
USE pubs
SELECT Title,
       Price,
       (SELECT AVG(Price) FROM Titles) AS Average, Price - (SELECT AVG(Price) FROM
Titles)
       AS Difference
FROM Titles
WHERE Type='popular_comp'
```

This gets us back a pretty simple set of data:

Title	Price	Average	Difference
But Is It User Friendly?	22.9500	14.7662	8.1838
Secrets of Silicon Valley	20.0000	14.7662	5.2338
Net Etiquette	NULL	14.7662	NULL

(3 row(s) affected)
Warning: Null value is eliminated by an aggregate or other SET operation.

Let's try it again, only this time we'll encapsulate both the average and the difference into two functions. The first encapsulates the task of calculating the average and the second does the subtraction.

```
CREATE FUNCTION dbo.AveragePrice()
RETURNS money
WITH SCHEMABINDING
AS
BEGIN
    RETURN (SELECT AVG(Price) FROM dbo.Titles)
END
GO

CREATE FUNCTION dbo.PriceDifference(@Price money)
RETURNS money
AS
BEGIN
    RETURN @Price - dbo.AveragePrice()
END
```

Notice that it's completely legal to embed one UDF in another one.

Note that the WITH SCHEMABINDING option works for functions just the way that it did for views—if a function is built using schema-binding, then any object that function depends on cannot be altered or dropped without first removing the schema-bound function. In this case, schema-binding wasn't really necessary, but I wanted to point out its usage and also prepare this example for something we're going to do with it a little later in the chapter.

Now let's run our query using the new functions instead of the old subquery model:

```
USE pubs
SELECT Title,
       Price,
       dbo.AveragePrice() AS Average,
       dbo.PriceDifference(Price) AS Difference
FROM Titles
WHERE Type='popular_comp'
```

This yields us the same results we had with our subquery, but without the warning!

Note that, beyond the readability issue, we also get added benefit of reuse out of this. For a little example like this, it probably doesn't seem like a big deal, but as your functions become more complex, it can be quite a time saver.

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, 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 datatype as far as a UDF is concerned. To illustrate this, we'll build a relatively simple one to start:

```
USE pubs
GO

CREATE FUNCTION dbo.fnAuthorList()
RETURNS TABLE
AS
RETURN (SELECT au_id,
              au_lname + ', ' + au_fname AS au_name,
              address AS address1,
              city + ', ' + state + ' ' + zip AS address2
              FROM authors)
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, and concatenating the three components to fill the address2 column.

At this point, we're ready to use our function just as we would use a table — the only exception is that, as was discussed with scalar functions, we *must* use the two-part naming convention:

```
SELECT *
FROM dbo.fnAuthorList()
```

The results are a bit lengthy, so I've clipped out the middle of them, but you should get the picture:

au_id	au_name	address1	address2
172-32-1176	White, Johnson	10932 Bigge Rd.	Menlo Park, CA 94025
213-46-8915	Green, Marjorie	309 63rd St. #411	Oakland, CA 94618
238-95-7766	Carson, Cheryl	539 Darwin Ln.	Berkeley, CA 94705
...			
...			
893-72-1158	McBadden, Heather	301 Putnam	Vacaville, CA 95688
899-46-2035	Ringer, Anne	67 Seventh Av.	Salt Lake City, UT 84152
998-72-3567	Ringer, Albert	67 Seventh Av.	Salt Lake City, UT 84152

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 we wanted this to show only authors who had sold at least certain quantity of books? We could do this with a view by joining with another table or two, but, then again, things get a bit messy and we would wind up having to include a column in our view we don't necessarily want (the sales quantity) and then use a `WHERE` clause. It might look something like this:

```
--CREATE our view
CREATE VIEW vSalesCount
AS
    SELECT au.au_id,
           au.au_lname + ', ' + au.au_fname AS au_name,
           au.address,
           au.city + ', ' + au.state + ' ' + zip AS address2,
           SUM(s.qty) AS SalesCount
    FROM authors au
    JOIN titleauthor ta
      ON au.au_id = ta.au_id
    JOIN sales s
      ON ta.title_id = s.title_id
    GROUP BY au.au_id,
           au.au_lname + ', ' + au.au_fname,
           au.address,
           au.city + ', ' + au.state + ' ' + zip
GO
```

This would yield us what was asked for, with a few twists. First, we can't parameterize things right in the view itself, so we're going to have to include a `WHERE` clause in our query. Second, we'll need to provide a specific `SELECT` list to filter out the `vSalesCount` column (remember, we want to show the authors who sold over a specific value, but not necessarily their actual sales):

```
SELECT au_name, address, Address2 FROM vSalesCount
WHERE SalesCount > 25
```

This should get you results that look something like this:

au_name	address	address2
Green, Marjorie	309 63rd St. #411	Oakland, CA 94618
Carson, Cheryl	589 Darwin Ln.	Berkeley, CA 94705
O'Leary, Michael	22 Cleveland Av. #14	San Jose, CA 95128
Dull, Ann	3410 Blonde St.	Palo Alto, CA 94301
DeFrance, Michel	3 Balding Pl.	Gary, IN 46403
MacFeather, Stearns	44 Upland Hts.	Oakland, CA 94612
Panteley, Sylvia	1956 Arlington Pl.	Rockville, MD 20853
Hunter, Sheryl	3410 Blonde St.	Palo Alto, CA 94301
Ringer, Anne	67 Seventh Av.	Salt Lake City, UT 84152
Ringer, Albert	67 Seventh Av.	Salt Lake City, UT 84152

Chapter 13

To simplify things a bit, we'll encapsulate everything in a function instead:

```
USE pubs
GO

CREATE FUNCTION dbo.fnSalesCount(@SalesQty bigint)
RETURNS TABLE
AS
RETURN (SELECT au.au_id,
              au.au_lname + ', ' + au.au_fname AS au_name,
              au.address,
              au.city + ', ' + au.state + ' ' + zip AS Address2
FROM authors au
JOIN titleauthor ta
  ON au.au_id = ta.au_id
JOIN sales s
  ON ta.title_id = s.title_id
GROUP BY au.au_id,
         au.au_lname + ', ' + au.au_fname,
         au.address,
         au.city + ', ' + au.state + ' ' + zip
HAVING SUM(qty) > @SalesQty
)

GO
```

Now we're set up pretty well—to execute it, we just call the function and provide the parameter:

```
SELECT *
FROM dbo.fnSalesCount(25)
```

And we get back the same result set—no WHERE clause, no filtering the SELECT list, and, as our friends down under would say, “no worries”; we can use this over and over again without having to use the old “cut and paste” trick. Note, also, that while you could have achieved similar results with a sproc and an EXEC command, you couldn't directly join the results of the sproc to another table.

To illustrate this, let's take our example just one step further. Let's say that you have a manager who wants a report listing the author and the publisher(s) for every author who's sold over 25 books. With a stored procedure, you couldn't join to the results, so you would pretty much be out of luck. (I can think of one several step processes to do this, but it isn't pretty at all.) With our function, this is no problem:

```
SELECT DISTINCT p.pub_name, a.au_name
FROM dbo.fnSalesCount(25) AS a
JOIN titleauthor AS ta
  ON a.au_id = ta.au_id
JOIN titles AS t
  ON ta.title_id = t.title_id
JOIN publishers AS p
  ON t.pub_id = p.pub_id
```

We get back our author listing along with all the different publishers that they have:

pub_name	au_name
Algodata Infosystems	Carson, Cheryl
Binnet & Hardley	DeFrance, Michel
Algodata Infosystems	Dull, Ann
Algodata Infosystems	Green, Marjorie
New Moon Books	Green, Marjorie
Algodata Infosystems	Hunter, Sheryl
Algodata Infosystems	MacFeather, Stearns
Binnet & Hardley	MacFeather, Stearns
Algodata Infosystems	O'Leary, Michael
Binnet & Hardley	O'Leary, Michael
Binnet & Hardley	Panteley, Sylvia
New Moon Books	Ringer, Albert
Binnet & Hardley	Ringer, Anne
New Moon Books	Ringer, Anne

As you can see, we joined to the function just as if it were a table or view. The only real difference is that we were allowed to parameterize it.

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, as we've seen in a single statement function — the only big difference 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 for Northwind. You have an `Employees` table, and it has a unary relationship that relates each employee to their boss through the `ReportsTo` column — that is, the way you know who someone's boss is by relating the `ReportsTo` 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 Andrew Fuller, we might write a query that would join the `Employees` table back to itself — something like:

```
Use Northwind
```

```
SELECT Emp.EmployeeID, Emp.LastName, Emp.FirstName, Emp.ReportsTo
FROM Employees AS Emp
```

```
JOIN Employees AS Mgr
  ON Mgr.EmployeeID = Emp.ReportsTo
WHERE Mgr.LastName = 'Fuller'
AND Mgr.FirstName = 'Andrew'
```

Again, at first glance, this might appear to give us what we want:

EmployeeID	LastName	FirstName	ReportsTo
1	Davolio	Nancy	2
3	Leverling	Janet	2
4	Peacock	Margaret	2
5	Buchanan	Steven	2
8	Callahan	Laura	2

(5 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 Andrew Fuller’s reporting chain—not just those who report to Andrew Fuller, but those who report to people who report to Andrew Fuller, and so on. You see that if you look at all the records in Northwind, you’ll find a number of employees who report to Steven Buchanan, 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 . . .

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 anytime it calls itself. We saw an example of this in our last chapter with our `spFactorial` and `spTriangular` stored procedures. 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,
        ReportsToID   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, ReportsTo
        FROM Employees
        WHERE EmployeeID = @EmployeeID
    /* Now we also need a primer for the recursive calls we're getting ready to start
    making
    ** 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 Employees
    WHERE ReportsTo = @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 Employees
            WHERE EmployeeID > @Employee
            AND ReportsTo = @EmployeeID
        END

    RETURN

END
GO

```

Chapter 13

I've written this one to provide just minimal information about the employee and their manager—I can join back to the `Employees` 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—Andrew Fuller is `EmployeeID #2`, so we'll feed that into our function:

```
SELECT * FROM fnGetReports(2)
```

This gets us not only the original five people who reported to Andrew Fuller, but also those who report to Steven Buchanan (who reports to Mr. Fuller) and Mr. Fuller himself (remember, I added him in as something of a starting point).

EmployeeID	ReportsToID
-----	-----
2	NULL
1	2
3	2
4	2
5	2
6	5
7	5
9	5
8	2

(9 row(s) affected)

As it happens, this is all of the employees in Northwind (unless you've added some yourself), but, if you play around with the data in the `ReportsTo` column some, you'll see we are indeed getting back the expected results. To test just a little further, however, you can feed in Steven Buchanan's ID (which is 5):

```
SELECT * FROM fnGetReports(5)
```

EmployeeID	ReportsToID
-----	-----
5	2
6	5
7	5
9	5

(4 row(s) affected)

We get the limited results we expected. 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 Andrew Fuller:

```
DECLARE @EmployeeID int

SELECT @EmployeeID = EmployeeID
  FROM Employees
  WHERE LastName = 'Fuller'
     AND FirstName = 'Andrew'

SELECT Emp.EmployeeID, Emp.LastName, Emp.FirstName, Mgr.LastName AS ReportsTo
```

```

FROM Employees AS Emp
JOIN dbo.fnGetReports(@EmployeeID) AS gr
  ON gr.EmployeeID = Emp.EmployeeID
JOIN Employees AS Mgr
  ON Mgr.EmployeeID = gr.ReportsToID

```

This gets us back all eight employees who are under Mr. Fuller:

EmployeeID	LastName	FirstName	ReportsTo
1	Davolio	Nancy	Fuller
3	Leverling	Janet	Fuller
4	Peacock	Margaret	Fuller
5	Buchanan	Steven	Fuller
6	Suyama	Michael	Buchanan
7	King	Robert	Buchanan
9	Dodsworth	Anne	Buchanan
8	Callahan	Laura	Fuller

(8 row(s) affected)

This should have you asking why Mr. Fuller didn't show up in the query — after all, we've already proven that he shows up in the results of the function. The reason that he doesn't show up is that the value in the ReportsTo column for his record is NULL, and so there's nothing to join back to the Employees table based on. The filtering happened because of the query, not because of the function.

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.

Chapter 13

- ❑ 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 only allowed 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—it's 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)
```

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!

We can compare this, however, with our `AveragePrice` function that we built in the `pubs` database. It looked something like this:

```
USE Pubs
GO

CREATE FUNCTION dbo.AveragePrice()
RETURNS money
WITH SCHEMABINDING
AS
BEGIN
    RETURN (SELECT AVG(Price) FROM dbo.Titles)
END
```

In this function we used schema-binding right from the beginning, so let's look at our `OBJECTPROPERTY`:

```
SELECT OBJECTPROPERTY(OBJECT_ID('AveragePrice'), 'IsDeterministic')
```

Despite being schema-bound, this one still comes back as being non-deterministic. That's because this function references a table that isn't local to the function (a temporary table or table variable created inside the function).

Under the heading of “one more thing,” it's also worth noting that the `PriceDifference` function we created at the same time as `AveragePrice` is also non-deterministic. For one thing, we didn't make it schema-bound, but, more important, it references `AveragePrice` — if you reference a non-deterministic function, then the function you're creating is non-deterministic by association.

Debugging User-Defined Functions

This actually works just the same as the `sproc` example we saw in Chapter 12.

In Visual Studio, use the Server Explorer to set up a connection and navigate to your UDF. Right click, and choose `Step Into` — it all works the same except for where you pick it out in the list (from functions instead of `sprocs`).

.NET in a Database World

As we discussed in Chapter 12, with SQL Server 2005 we gained the ability to use .NET assemblies in our stored procedures and functions. Much as it did with `sprocs`, this has enormous implications for functions.

Considering most who read this title will be beginners, it's hard to fully relate the impact that .NET has in our database world. The reality is that you won't use it all that often, and yet, when you do, the effects can be profound. Need to implement a complex formula for a special function? No problem. Need to access external data sources such as credit card authorization companies and such things? No problem. Need to access other complex data sources? No problem. In short, things we used to have to either skip or otherwise perform extremely complex development for (in some cases, it was all smoke and mirrors before) suddenly become relatively straightforward.

What does this mean in terms of functions? Well, I already gave the example of implementing a complex formula in a function. But now imagine something like external tabular data — let's say representing a .csv or some other data in a tabular fashion — very doable with a .NET assembly created as a function in SQL Server.

.NET assemblies in SQL Server remains, however, something of an advanced concept, and one I'll defer to the *Professional* series title for SQL Server 2005. That said, it's important to understand that the option is available and consider it as something worth researching in that “wow, I have no idea how we're going to do this!” situation.

Summary

What we added in this chapter was, in many ways, not new at all. Indeed, much of what goes into user-defined functions is the same set of statements, variables, and general coding practices that we have already seen in scripting and stored procedures. However, UDFs still provide us a wonderful new area of functionality that was not previously available in SQL Server. We can now encapsulate a wider range of code, and even use this encapsulated functionality inline with our queries. What's more, we can now also provide parameterized views and dynamically created tables.

User-defined functions are, in many ways, the most exciting of all the new functionality added to SQL Server. In pondering their uses, I have already come to realize that I'm only scratching the surface of their potential. Over the life of this next release, I suspect that developers will implement UDFs in ways I have yet to dream of — let's hope you'll be one of those developers!

Exercise

1. Reimplement the `spTriangular` function from Chapter 12 as a function instead of a stored procedure.

14

Transactions and Locks

This is one of those chapters that, when you go back to work, makes 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:

- ❑ Demystify 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."

Often, when dealing with data, we want to make sure that if one thing happens, another thing happens, or that neither of them does. 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 hyper-simplification 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 OK 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 we want to forget that it ever happened.
- ❑ **SAVE a transaction:** Establishing 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 *save points* — which we'll describe shortly.

The syntax for a `ROLLBACK` again looks pretty much the same, with the exception of allowance for a save point.

```
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 save point to which you want to roll back.

The syntax is simple enough:

```
SAVE TRAN[SACTION] [<save point name>| <@savepoint variable>]
```

The thing to remember about save points is that they are cleared on `ROLLBACK` — that is, even if you save five save points, once you perform one `ROLLBACK` they are all gone. You can start setting new save points again, and rolling back to those, but whatever save points you had when the `ROLLBACK` was issued are gone.

`SAVE TRAN` can get extremely confusing and I can't recommend it for the beginning user, but keep it in mind as being there.

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 14-1.

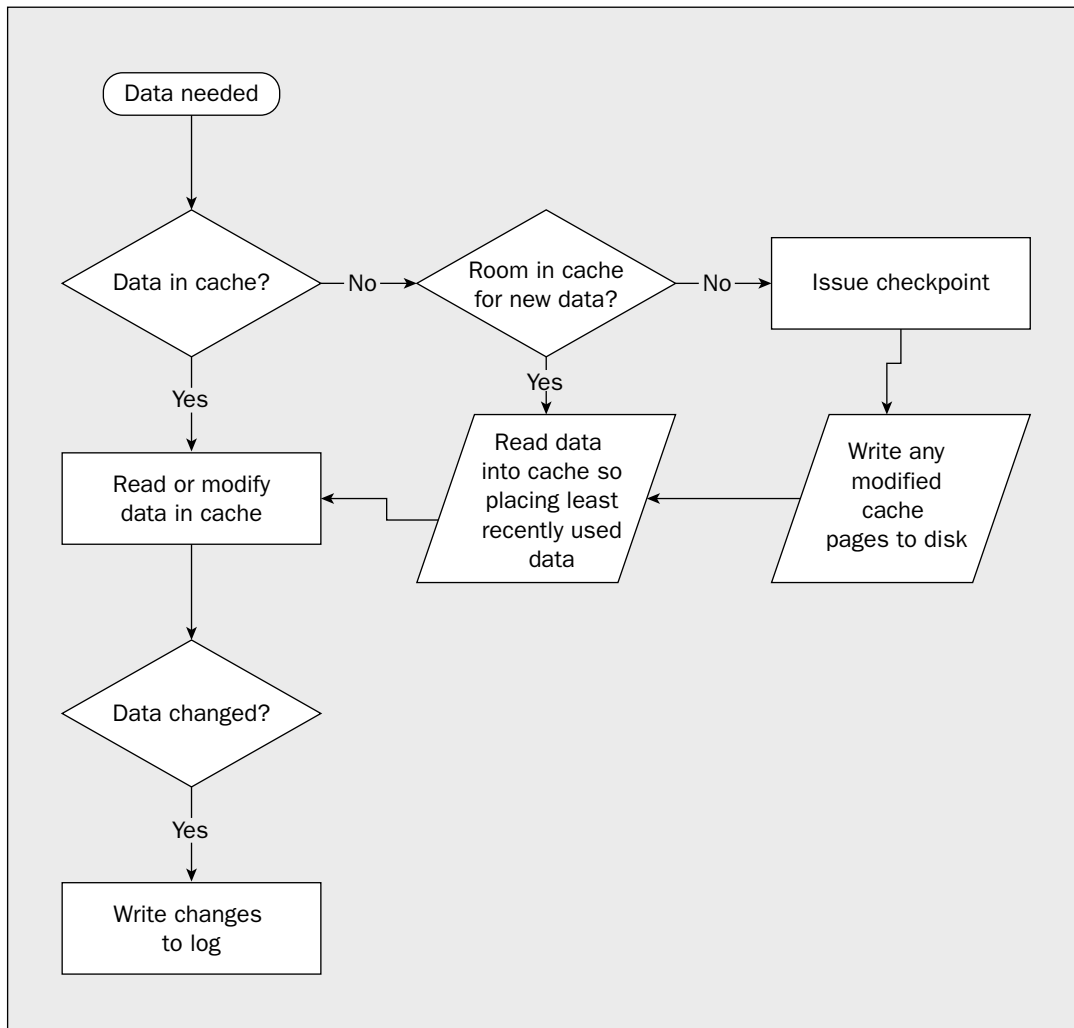


Figure 14-1

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

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 14-2.

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.

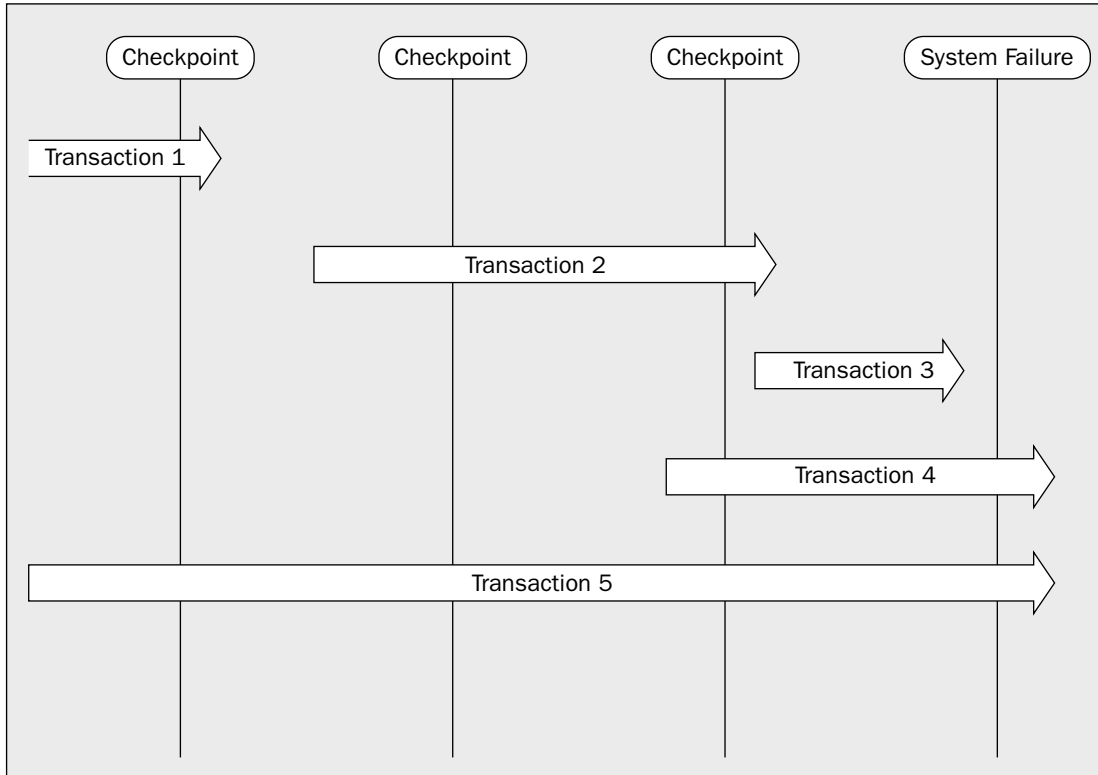


Figure 14-2

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.

Implicit transactions are something of a dangerous territory and are well outside the scope of this book. Suffice to say that 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).

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 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. (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, 1000, 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
	UPDATE value, SET value = value-50			75
IF @Var >=100	END TRAN	125	125	75
UPDATE value, SET value = value-100		125	125 (waiting for lock to clear)	75
(Finish, wait for lock to clear, then continue)		125	75	Either: -25 (If there isn't a CHECK constraint enforcing > 0) Or: Error 547 (If there is a CHECK)

Again, we have a problem. Transaction 1 has pre-scanned (which can be a good practice in some instances—remember that section, "Handling Errors Before They Happen" in Chapter 12?) 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 would 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 the `Employees` table 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 their credit limit, and would like an extension, so you pull up their 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 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, then 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, then 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 is 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 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 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 affect.

Optimizer hints are seriously in the “Advanced” side of things. They are often abused by people who are experienced SQL Server developers, and they are not to be trifled with.

Think of it this way — Microsoft has invested literally millions of dollars in such things as their query optimizer and knowing what locks to utilize in what situations. Query hints are meant to adjust for the little things the optimizer may not know about, but in the vast majority of cases, you are *not* going to know more than their optimizer team did. Shy away from these until the later stages of your SQL Server learning process (and I promise, I’ll cover them well in the *Professional* version of this book).

Determining Locks Using the Management Studio

Perhaps the nicest way of all to take a look at your locks is by using the Management Studio. The 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 the 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 14-3.

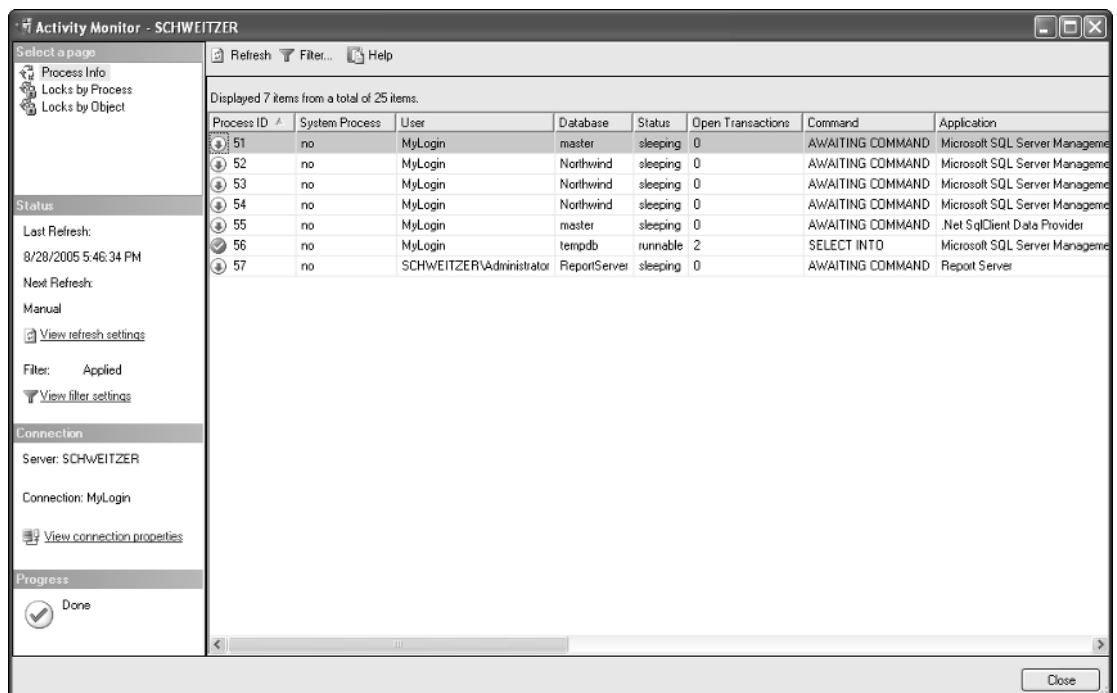


Figure 14-3

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 `REPEATABLE READ` (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, like 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, from 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 (aka “A 1205”)

OK. 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 five 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 (OK, 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 (described briefly below).

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.

Use 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. The 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), then 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.

Use 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.

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.

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.

15

Triggers

Ah, triggers. Triggers are cool, triggers are neat, and triggers are our friends. At the very same time, triggers are evil, triggers are ugly, and triggers are our enemy. In short, I am often asked, “Should I use triggers?” The answer is, like 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.

From a beginner’s point of view (and by this chapter in this book, I hope you’re a lot less of a beginner — but still . . .), you really want to be *certain* you know what you’re doing before you go the triggers route, so sit back, listen, learn, and decide for yourself whether they are right for you.

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?
- ☐ 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.

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, 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).
- ❑ 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 (like 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 only need to look into using these where you need extreme auditing of changes/history of your database structure. Their use is a fairly advanced concept, and as such, I'm covering them here as mostly a "be aware these exist" thing, and we'll move on to the meatier version of triggers.

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, there are three types of triggers, plus hybrids that come from mixing and matching the events and timing that fire them:

1. INSERT triggers
2. DELETE triggers
3. UPDATE triggers
4. A mix and match of any of the above

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 — 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** vs. 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 15-1).

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 10 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.

As positively glorious as this sounds, this is actually pretty complex stuff. It means that you need to anticipate every possibility. In addition, it means that you are effectively adding a preprocess to every queries that changes data in any way for this table (this is not a good thing performance wise). Cool as they sound, `INSTEAD OF` triggers fall in the category of fairly advanced stuff, and are well outside the scope of this book.

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.

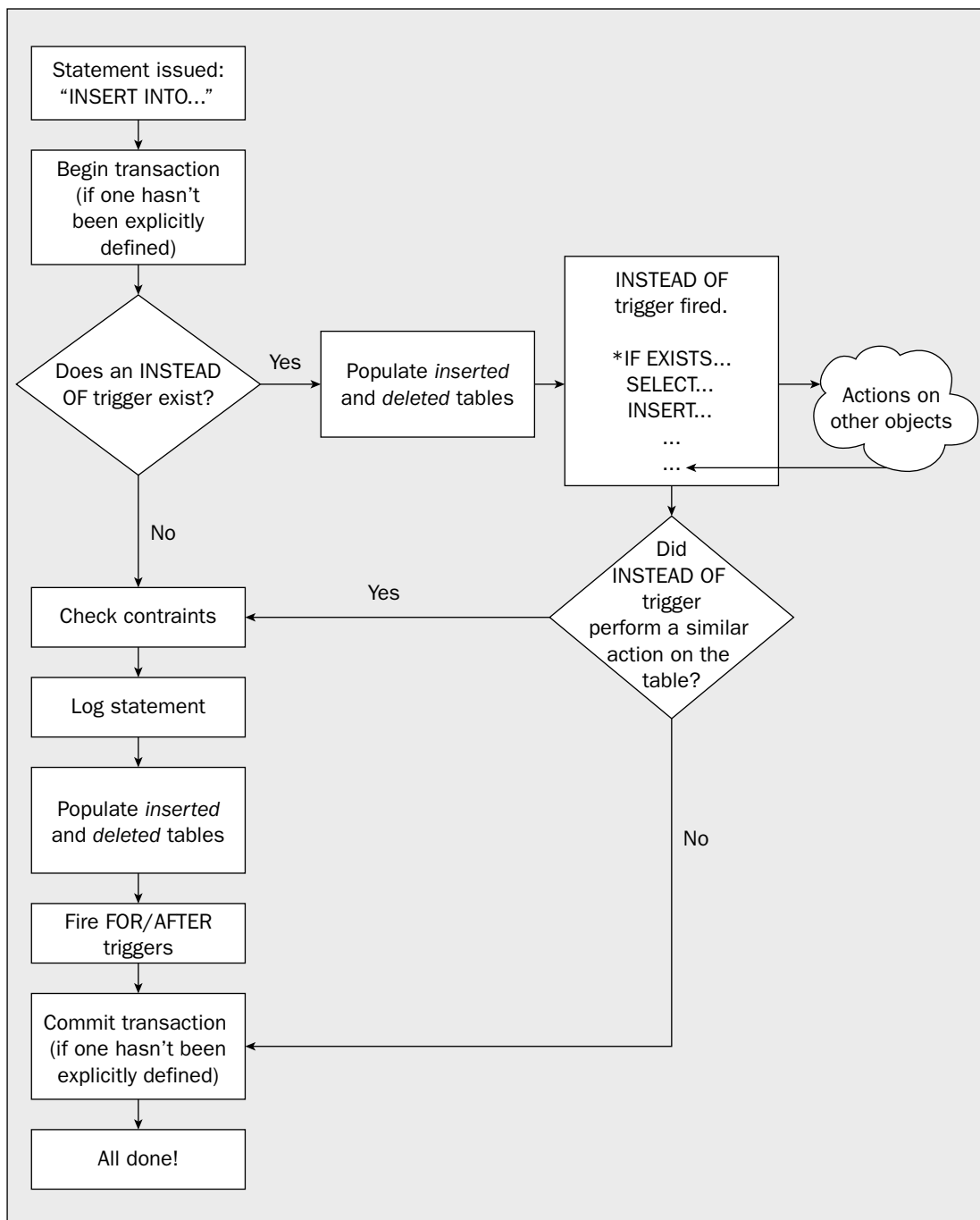


Figure 15-1

FOR|AFTER

The `FOR` (or, alternatively, you can use `AFTER`) 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:

```
FOR INSERT, DELETE
```

... or:

```
FOR UPDATE, INSERT
```

... or:

```
FOR DELETE
```

As was stated in the section about the `ON` clause, triggers declared using the `FOR` or `AFTER` clause can only be attached to tables—no views are allowed (see `INSTEAD OF` triggers for those).

INSERT Trigger

The code for any trigger that you mark as being `FOR INSERT` will be executed anytime 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` only applies 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, trigger can also perform the same functionality as a CHECK constraint or even a DEFAULT. The answer to the question "Should I use triggers vs. 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 6.

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 `Order Details` tables in the Northwind database. The relationship looks like Figure 15-2.

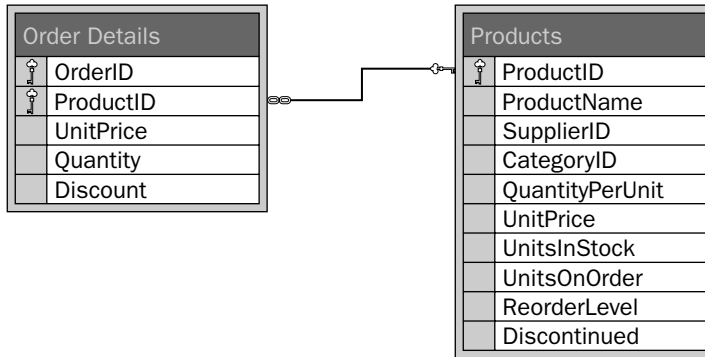


Figure 15-2

So, under normal DRI, you can be certain that no `Order Detail` item can be entered into the `Order Details` table unless there is a matching `ProductID` in the `Products` table. We are, however, looking for something more than just the “norm” here.

Our Inventory department has been complaining that our Customer Support people keep placing orders for products that are marked 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 `Order Details` table). Don't sweat it though — you can tell the Inventory department, “No problem!” You just need to use a trigger:

```

CREATE TRIGGER OrderDetailNotDiscontinued
ON [Order Details]
FOR INSERT, UPDATE
AS
    IF EXISTS
    (
        SELECT 'True'
        FROM Inserted i
        JOIN Products p
          ON i.ProductID = p.ProductID
        WHERE p.Discontinued = 1
    )
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 a record or two that will fail when it hits our trigger:

```
SELECT ProductID, ProductName FROM Products WHERE Discontinued = 1
```

ProductID	ProductName
-----	-----
5	Chef Anton's Gumbo Mix
9	Mishi Kobe Niku
17	Alice Mutton
24	Guaraná Fantástica
28	Rössle Sauerkraut
29	Thüringer Rostbratwurst
42	Singaporean Hokkien Fried Mee
53	Perth Pasties
(8 row(s) affected)	

So let's go ahead and add an `Order Details` item that violates this constraint:

```
INSERT [Order Details]
  (OrderID, ProductID, UnitPrice, Quantity, Discount)
VALUES
  (10000, 5, 21.35, 5, 0)
```

This gets the rejection that we expect:

```
Msg 50000, Level 16, State 1, Line -1074284106
Order Item is discontinued. Transaction Failed.
```

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.

To check this out, let's take a look at the `Products` table again. `Products` has a column called `UnitsInStock`. Recently, there has been a rush on several products, and Northwind has been selling out of several things. Since Northwind needs more than just a few customers to stay in business in the long run, it has decided to institute a rationing system on their products. The Inventory department has requested that we prevent orders from being placed that try to sell more than half of the units in stock for any particular product.

Chapter 15

To implement this, we make use of both the `Inserted` and `Deleted` tables:

```
CREATE TRIGGER ProductIsRationed
ON Products
FOR UPDATE
AS
    IF EXISTS
    (
        SELECT 'True'
        FROM Inserted i
        JOIN Deleted d
            ON i.ProductID = d.ProductID
        WHERE (d.UnitsInStock - i.UnitsInStock) > d.UnitsInStock / 2
            AND d.UnitsInStock - i.UnitsInStock > 0
    )
BEGIN
    RAISERROR('Cannot reduce stock by more than 50%% at once.',16,1)
    ROLLBACK TRAN
END
```

Before we test this out, let's analyze what we're doing here.

First, we're making use of an `IF EXISTS` just as we have throughout this chapter. We only want to do the rollback if something exists that meets the evil, mean, and nasty criteria that we'll be testing for.

Then we join the `INSERTED` and `DELETED` tables together — this is what gives us the chance to compare the two.

Our `WHERE` clause is the point where things might become a bit confusing. The first line of it is pretty straightforward. It implements the nominal statement of our business requirement; updates to the `UnitsInStock` column that are more than half the units we previously had on hand will meet the criterion, and ready the transaction to be rejected.

The next line, though, is not quite so straightforward. As with all things in programming, we need to think beyond the nominal statement of the problem, and think about other ramifications. The requirement really only applies to reductions in orders — we certainly don't want to restrict how many units be put *in* stock — so we make sure that we only worry about updates where the number in stock after the update is less than before the update.

If both of these conditions have been met (over 50 percent, and a reduction rather than addition to the inventory), then we raise the error. Notice the use of two `%` signs, rather than one, in the `RAISERROR`. Remember that a `%` works as a placeholder for a parameter, so one `%` by itself won't show up when your error message comes out. By putting two in a row, `%%`, we let SQL Server know that we really did want to print out a percent sign.

OK — let's check out how it works. We'll just pick a record and try to do an update that reduces the stock by more than 50 percent:

```
UPDATE Products
SET UnitsInStock = 2
WHERE ProductID = 8
```

I just picked out “Northwoods Cranberry Sauce” as our victim, but you could have chosen any `ProductID` as long as you set the value to less than 50 percent of its previous value. If you do, you’ll get the expected error:

```
Msg 50000, Level 16, State 1, Line -1074284106
Cannot reduce stock by more than 50% at once.
```

Note that we could have also implemented this in the `Order Details` table by referencing the actual order quantity against the current `UnitInStock` amount, but we would have run into several problems:

- ❑ **Updates that change:** Is the process that’s creating the `Order Details` record updating `Products` before or after the `Order Details` record? That makes a difference in how we make use of the `UnitsInStock` value in the `Products` table to calculate the effect of the transaction.
- ❑ **The inventory external to the `Order Details` table updates would not be affected:** They could still reduce the inventory by more than half (this may actually be a good thing in many circumstances, but it’s something that has to be thought about).

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

As you can see, the possibilities are pretty far reaching — it’s really all about your particular situation and the needs of your particular system.

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 the 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 spdoc.

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.

Triggers Don't Prevent 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 had the triggers turned off.

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 only accept 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 15-3).

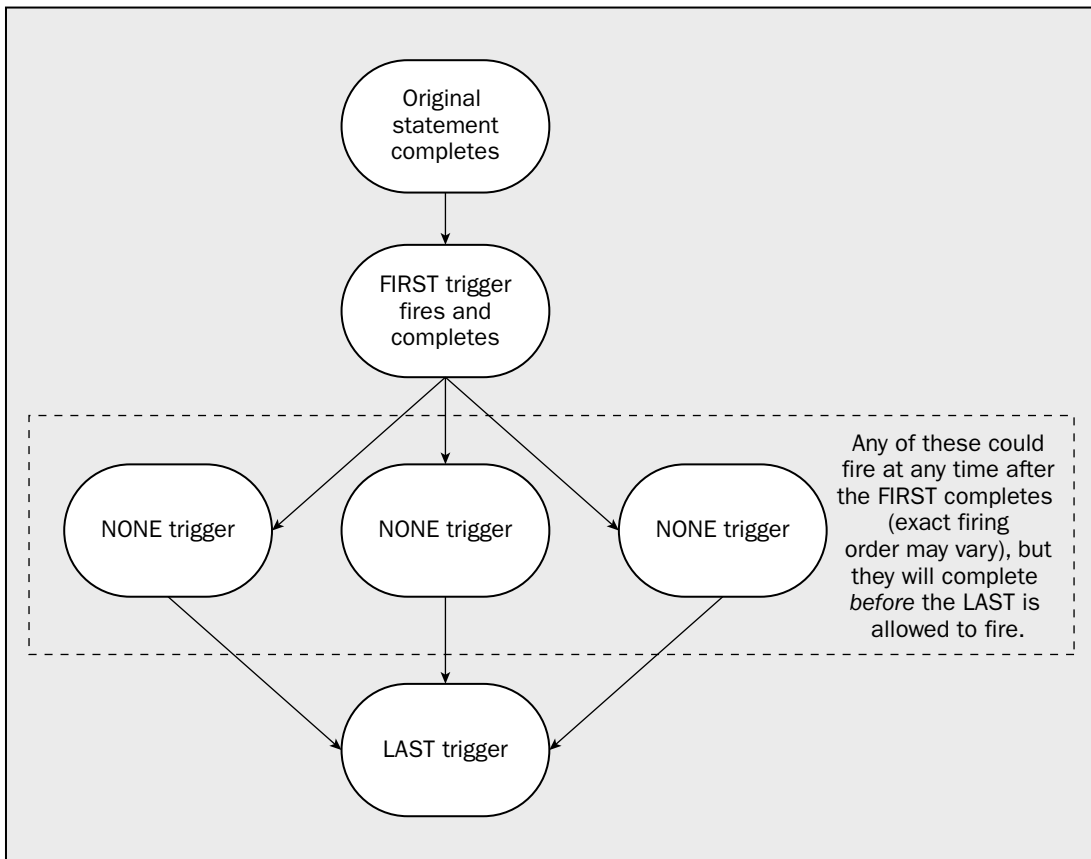


Figure 15-3

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 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.

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 also 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

`INSTEAD OF` triggers were added in SQL Server 2000, and remain one of the more complex features of SQL Server. While it is well outside the scope of a “beginning” concept, I’m still a big believer in even the beginner learning about what things are *available*, and so we’ll touch on what these are about here.

Essentially, an `INSTEAD OF` trigger is a block of code we can use as an interceptor for anything that anyone tries to do to our table or view. We can either elect to just go ahead and do whatever the user requests or, if we choose, we can go so far as doing something that is entirely different.

Like regular triggers, `INSTEAD OF` triggers come in three different flavors: `INSERT`, `UPDATE`, and `DELETE`. In each case, the most common use is the same — resolving ambiguity of what table(s) are to receive the actual changes when you’re dealing with a view based on multiple tables.

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 non-committal 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 detect things that will fail and will prevent them from happening earlier in the process. 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 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.

Using 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 ProductIsRationed
ON Products
FOR UPDATE
AS
IF UPDATE(UnitsInStock)
BEGIN
IF EXISTS
(
SELECT 'True'
FROM Inserted i
JOIN Deleted d
ON i.ProductID = d.ProductID
WHERE (d.UnitsInStock - i.UnitsInStock) > d.UnitsInStock / 2
AND d.UnitsInStock - i.UnitsInStock > 0
)
BEGIN
RAISERROR('Cannot reduce stock by more than 50%% at once.',16,1)
ROLLBACK TRAN
END
END
```

With this change, we will now limit the rest of the code to run only when the UnitsInStock 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 15-4.

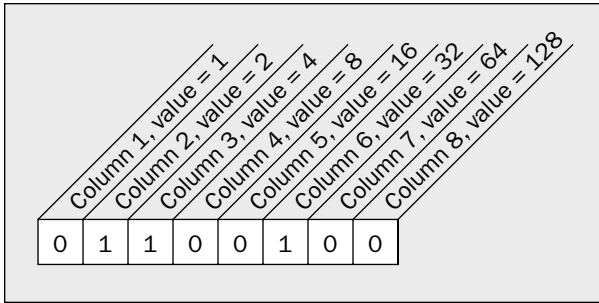


Figure 15-4

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 15-5).

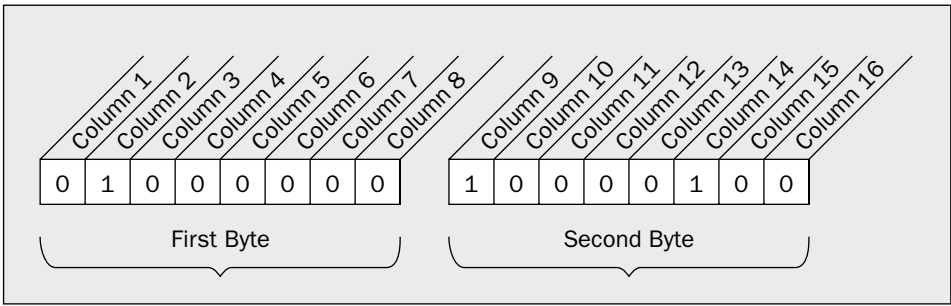


Figure 15-5

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 bitmask 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 bitmask 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!

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 12 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;
```

ENDNow I just navigate to this sproc in the Server Explorer, right-click on it, and select Step Into (see Figure 15-6).

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 15-7)!

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.

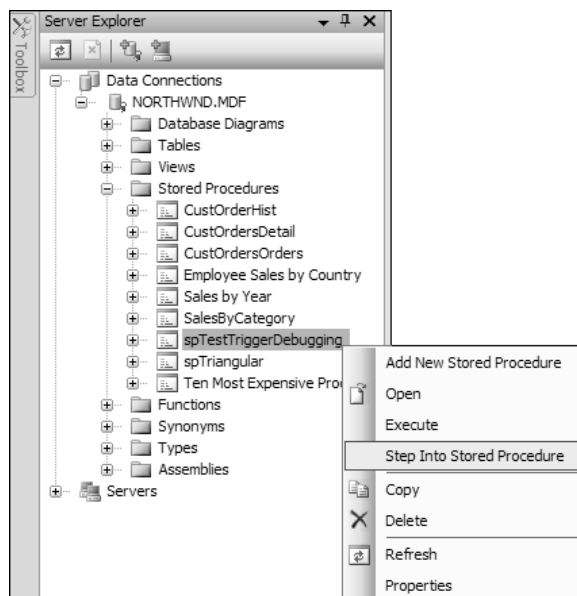


Figure 15-6

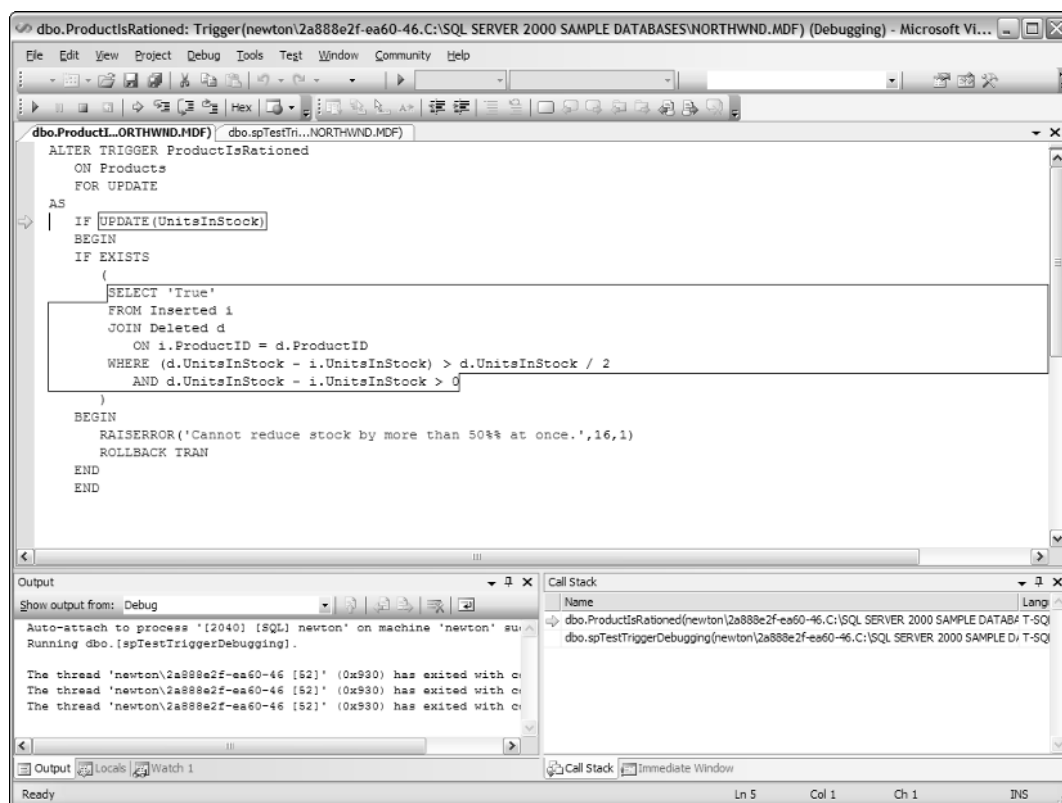


Figure 15-7

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.

Before you get too frustrated with triggers — or before you get too bored with the couple of trigger templates that fill about 90 percent of your trigger needs — keep in mind that there are a large number of tools out there that will auto-generate triggers for you that meet certain requirements. We will be looking at this a bit further in Appendix C of this book.

16

A Brief XML Primer

So, here we are — most of our structural stuff is done at this point, and we're ready to start moving on to the peripheral stuff. That is, we're ready to start looking at what I would consider to be the "extras" of SQL Server. It's not that some of the items we still have to cover aren't things that you would normally expect out of an RDBMS system — it's just that we don't really *need* these in order to have a functional SQL Server. Indeed, there are so many things included in SQL Server now, that it's difficult to squeeze everything into one book.

This chapter will start by presenting some background for the first of these "extras." We will then move on to looking at some of the many features SQL Server has to support XML. The catch here is that XML is really entirely its own animal — it's a completely different kind of thing than the relational system we've been working with up to this point. Why then does SQL Server include so much functionality to support it? The short answer is that XML is probably the most important thing to happen to data since the advent of data warehousing.

XML has actually been around for years now, but, while the talk was big, its actual usage was not what it could have been. Since the year 2000 or so, XML has gone into wider and wider use as a generic way to make data feeds and reasonable size data documents available. XML provides a means to make data self-describing — that is, you can define type and validation information that can go along with the XML document so that no matter who the consumer is (even if they don't know anything about how to connect to SQL Server), they can understand what the rules for that data are.

XML is usually not a very good place to *store* data, but it is a positively spectacular way of making data *useful* — As such, the ways of utilizing XML will likely continue to grow, and grow, and grow.

So, with all that said, in this chapter we'll look at:

- ☐ What XML is
- ☐ What other technologies are closely tied to XML

I mentioned a bit ago that XML is usually not a good way to store data, but there are exceptions. One way that XML is being utilized for data storage is for archival purposes. XML compresses very well, and it is in a very open kind of format that will be well understood for many years to come—if not forever. Compare that to, say, just taking a SQL Server 2005 backup. A decade from now when you need to restore some old data to review archival information, you may very well not have a SQL Server installation that can handle such an old backup file, but odds are very strong indeed that you'll have something around that can both decompress (assuming you used a mainstream compression library such as ZIP) and read your data. Very handy for such “deep” archives.

XML Basics

There are tons and tons of books out there on XML (for example, Wrox's *Professional XML*, by Birbeck et. al). Given how full this book already is, my first inclination was to shy away from adding too much information about XML itself, and assume that you already knew something about XML. I have, however, come to realize that even all these years after XML hit the mainstream, I continue to know an awful lot of database people who think that XML “is just some Web technology,” and, therefore, have spent zero time on it—they couldn't be more wrong.

XML is first and foremost an *information* technology. It is *not* a Web-specific technology at all. Instead, it just tends to be thought of that way (usually by people who don't understand XML) for several reasons—such as:

- ❑ XML is a *markup* language, and looks a heck of a lot like HTML to the untrained eye.
- ❑ XML is often easily *transformed* into HTML. As such, it has become a popular way to keep the information part of a page, with a final transformation into HTML only on request—a separate transformation can take place based on criteria (such as what browser is asking for the information).
- ❑ One of the first widely used products to support XML was Microsoft's Internet Explorer.
- ❑ The Internet is quite often used as a way to exchange information, and that's something that XML is ideally suited for.

Like HTML, XML is a text-based markup language. Indeed, they are both derived from the same original language, called SGML. SGML has been around for much longer than the Internet (at least what we think of as the Internet today), and is most often used in the printing industry or in government related documentation. Simply put, the “S” in SGML doesn't stand for simple—SGML is anything but intuitive and is actually a downright pain to learn. (I can only read about 35 percent of SGML documents that I've seen. I have, however, been able to achieve a full 100 percent nausea rate when reading any SGML.) XML, on the other hand, tends to be reasonably easy to decipher.

So, this might have you asking the question: “Great—Where can I get a listing of XML tags?” Well, you can’t—at least, not in the sense that you’re thinking when you ask the question. XML has very few tags that are actually part of the language. Instead, it provides for ways of defining your own tags and for utilizing tags as defined by others (such as the industry groups I mentioned earlier in the chapter). XML is largely about flexibility—which includes the ability for you to set your own rules for your XML through the use of either an XML schema document or the older Document Type Definition (DTD).

An XML document has very few rules placed on it just because it happens to be XML. The biggie is that it must be what is called *well formed*. We’ll look into what well formed means shortly. Now, just because an XML document meets the criteria of being well formed doesn’t mean that it would classify as being valid. Valid XML must not only be well formed, but must also live up to any restrictions placed on the XML document by XML schemas or DTDs that document references. We will briefly examine DTDs and XML schemas later on in the chapter.

XML can also be transformed. The short rendition of what this means is that it is relatively easy for you to turn XML into a completely different XML representation or even a non-XML format. One of the most common uses for this is to transform XML into HTML for rendering on the Web. The need for this transformation presents us with our first mini-opportunity to compare and contrast HTML vs. XML. In the simplest terms, XML is about information, and HTML is about presentation.

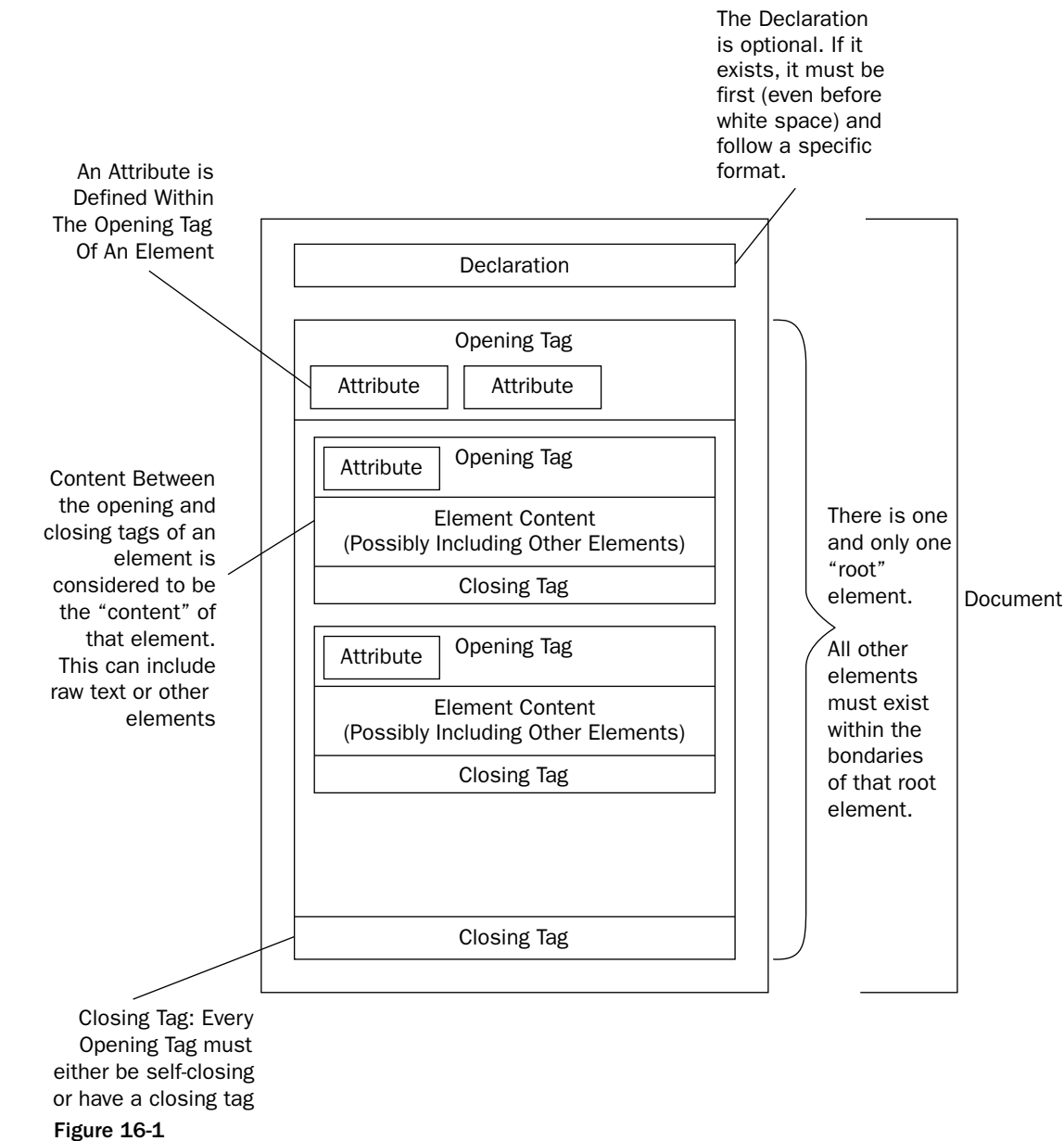
The information stored in XML is denoted through the use of what are called *elements* and *attributes*. Elements are usually created through the use of both an opening and a closing tag (there’s an exception, but we’ll see that later) and are identified with a case sensitive name (no spaces allowed). Attributes are items that further describe elements, and are embedded in the element’s start tag. Attribute values must be in matched single or double quotes.

Parts of an XML Document

Well, a few of the names have already flown by, but it makes sense, before we get too deep into things, to stop and create something of a glossary of terms that we’re going to be utilizing while talking about XML documents.

What we’re really going to be doing here is providing a listing of all the major parts of an XML document that you will run into, as shown in Figure 16-1. Many of the parts of the document are optional, though a few are not. In some cases, having one thing means that you have to have another. In other cases, the parts of the document are relatively independent of each other.

We will take things in something of a hierarchical approach (things that belong “inside” of something will be listed after whatever they belong inside of), and where it makes sense, in the order you’ll come across them in a given XML document.



The Document

The document encompasses everything from the very first character to the last. When we refer to an XML document, we are referring to both the structure and the content of that particular XML document.

Declaration

The *declaration* is technically optional, but, as a practical matter, should always be included. If it exists, it must be the very first thing in the document. *Nothing* can be before the declaration, not even white space (spaces, carriage returns, tabs, whatever) — nothing.

The declaration is made with a special tag that begins with a question mark (which indicates that this tag is a preprocessor directive) and the “xml” moniker:

```
<?xml version="1.0"?>
```

The declaration has one required attribute (something that further describe the element) — the *version*. In the preceding example, we’ve declared that this is an XML document and also that it is to comply with version 1.0 of the XML specification.

The declaration can optionally have one additional attribute — this one is called *encoding*, and it describes the nature of the character set this XML document utilizes. XML can handle a few different character sets, most notably UTF-16 and UTF-8. UTF-16 is essentially the Unicode specification, which is a 16-bit encoding specification that allows for most characters in use in the world today. The default encoding method is UTF-8, which is backward compatible to the older ASCII specification. A full declaration would look like this:

```
<?xml version='1.0' encoding='UTF-8'?>
```

Elements that start with the letters xml are strictly forbidden by the specification — instead, they are reserved for future expansion of the language.

Elements

Elements serve as a piece of glue to hold together descriptive information about something — it honestly could be anything. Elements define a clear start and end point for your descriptive information. Usually, elements exist in matched pairs of tags known as an opening tag and a closing tag. Optionally, however, the opening tag can be *self-closing* — essentially defining what is known as an *empty element*.

The structure for an XML element looks pretty much as HTML tags do. An opening tag will begin with an opening angle bracket (<), contain a name and possibly some attributes, and, then a closing angle bracket (>).

```
<ATagForANormalElement >
```

The exception to the rule is if the element is self-closing, in which case the closing angle bracket of the opening tag is preceded with a “/”.

```
<ASelfClosingElement/>
```

Closing tags will look exactly like the opening tag (case sensitive), but start with a slash (/) before the name of the element it’s closing.

```
<ATagForANormalElement >      <== Opening Tag
Some data or whatever can go in here.
We're still going strong with our data.
</ATagForANormalElement >    <== Closing Tag
```

Chapter 16

Elements can also contain attributes (which we'll look at shortly) as part of the opening (but not the closing) tag for the element. Finally, elements can contain other elements, but, if they do, the inner element must be closed before closing the outer element:

```
<OuterElement>
  <InnerElement>
  </InnerElement>
</OuterElement>
```

We will come back to elements shortly when we look at what it means to be well formed.

Nodes

When you map out the hierarchies that naturally form in an XML document, they wind up taking on the familiar tree model that you see in just about any hierarchical relationship, illustrated in Figure 16-2. Each intersection point in the tree is referred to as a *node*.

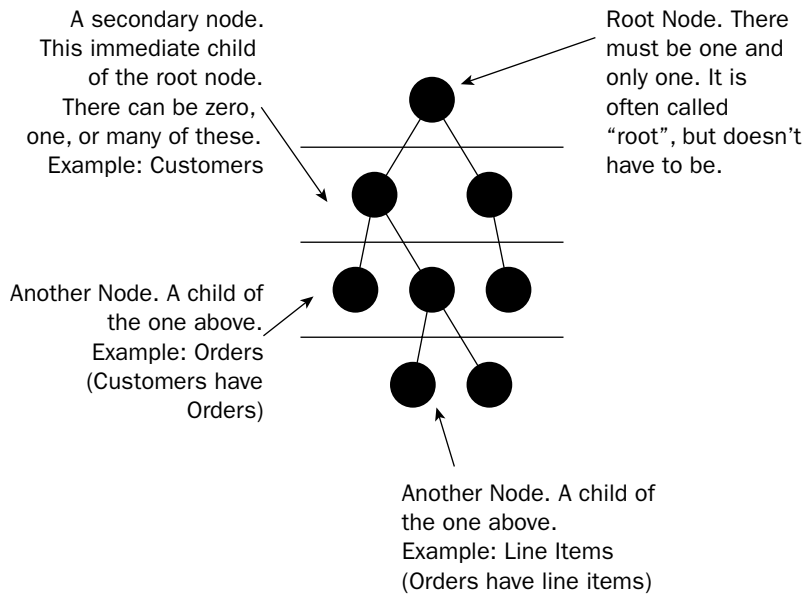


Figure 16-2

The contents of an XML document can be navigated based on node levels and the values in the attributes and elements of particular nodes.

The "Root" Node

Perhaps one of the most common points of confusion in XML documents is over what is called the *root node*. Every XML document must have exactly one (no more, no less) root node. The root node is an element that contains any other elements in the document (if there are any). You can think of the root node as being the unification point that ties all the nodes below it together and gives them structure within this scope of any particular XML document. So, what's all the confusion about? Well, it usually falls into

two camps: Those that don't know that they need to have a singular root node (which you now know), and those that don't understand how root nodes are named (which you will understand in a moment).

Because the general statement is usually "You must have a root node," people usually interpret that to mean that they must have a node that is called `root`. Indeed, you'll occasionally find XML documents that do indeed have a root node named `Root` (or `root` or `ROOT`). The reality, however, is that root nodes follow the exact same naming scheme as any other element with only one exception — the name must be unique throughout the document. That is, no other element in the entire document can have the same name as the root.

The existence of a root node is something of a key difference between an XML document and an XML fragment. Often, when extracting things from SQL Server, you'll be extracting little pieces of XML that belong to a large whole. We refer to these as XML fragments. Because an XML fragment is not supposed to be the whole document, we don't expect these to have a root node.

Attributes

Attributes exist only within the context of an element. They are implemented as a way of further describing an element, and are placed within the boundaries of the opening tag for the element:

```
<SomeElement MyFirstAttribute="Hi There" MySecondAttribute="25">
    Optionally, some other XML
</SomeElement>
```

Regardless of the datatype of the information in the value for the attribute, the value must be enclosed in either single or double quotes.

By default, XML documents have no concept of datatype. We will investigate ways of describing the rules of individual document applications later in this chapter. At that time, we'll see that there are some ways of ensuring datatype — it's just that you set the rules for it; XML does not do that by itself.

No Defects — Being Well Formed

The part of the rules that define how XML must look — that is, what elements are okay, how they are defined, what parts they have — is about whether an XML document is well formed or not.

Actually, all SGML-based languages have something of the concept of being well formed. Heck, even HTML has something of the concept of being well formed — it's just that it has been largely lost in the fact that HTML is naturally more forgiving and that the browsers ignore many errors.

If you're used to HTML at all, then you've seen some pretty sloppy stuff as far as a tag-based language goes. XML has much more strict rules about what is and isn't OK. The short rendition looks like this:

- ❑ Every XML document *must* have a unique "root" node.
- ❑ Every tag must have a matching (case sensitive) closing tag unless the opening tag is self-closing.

- ❑ Tags cannot straddle other tags.
- ❑ You can't use restricted characters for anything other than what they indicate to the XML parser. If you need to represent any of these special characters, then you need to use an escape sequence (which will be translated back to the character you requested).

The following is an example of a document that is well formed:

```
<?xml version="1.0" encoding="UTF-8"?>

<ThisCouldBeCalledAnything>
  <AnElement>
    <AnotherElement AnAttribute="Some Value">
      <AselfClosingElement AnAttributeThatNeedsASpecialCharacter="Fred&quot;s
flicks"/>
    </AnotherElement>
  </AnElement>
</ThisCouldBeCalledAnything>
```

Notice that we didn't need to have a closing tag at all for the declaration. That's because the declaration is a preprocessor directive — not an element. Essentially, it is telling the XML parser some things it needs to know before the parser can get down to the real business of dealing with our XML.

So, this has been an extremely abbreviated version of what's required for your XML document to be considered to be well formed, but it pretty much covers the basics for the limited scope of our XML coverage in this book.

Understanding these concepts is going to be absolutely vital to your survival (well, comprehension at least) in the next chapter. The example that is covered next should reinforce things for you, but, if after looking at the XML example, you find you're still confused, then read the above again or check out Professional XML or some other XML book. Your sanity depends on knowing this stuff before you move on to the styling and schema issues at the end of the chapter — let alone the next chapter.

An XML Example

Ok — if there's one continuing theme throughout this book, it's got to be that I don't like explaining things without tossing out an example or two. As I've said earlier, this isn't an XML book, so I'm not going to get carried away with my examples here, but let's at least take a look at what we're talking about.

Throughout the remainder of this chapter and the next, you're going to find that life is an awful lot easier if you have some sort of XML editing tool. Because XML is text based, you can easily open and edit XML documents in Notepad — the problem is that you're not going to get any error checking. How are you going to know that your document is well formed? Sure, you can just look it over if it's just a few lines, but get to a complete document or an XSL document, and life will quickly become very difficult.

As a side note, you can perform a rather quick and dirty check on whether your XML is well formed or not by opening the document in Microsoft's Internet Explorer — it will complain to you if the document is not well formed.

For this example, we're going to look at an XML representation of what some of our Northwind data might look like. In this case, I'm going to take a look at some order information. We're going to start with just a few things and grow from there.

First, we know that we need a root node for any XML document that we're going to have. The root node can be called anything we want, as long as it is unique within our document. A common way of dealing with this is to call the root `root`. Another common example would be to call it something representative of what the particular XML document is all about.

For our purposes, we'll start off with something hyper simple, and just use `root`:

```
<root>
</root>
```

Just that quick, we've created our first well-formed XML document. Notice that it didn't include the `<?xml>` tag that we saw in the earlier illustration. We could have put that in, but it's actually an optional item. The only restriction related to it is that, if you include it, it must be first. For best practice reasons as well as clearness, we'll go ahead and add it:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
</root>
```

Actually, by the rules of XML, any tag starting with `<?xml>` is considered to be a reserved tag — that is, you shouldn't name your tags that, as they are reserved for current or future use of the W3C as XML goes into future versions.

So, moving on, we have our first well-formed XML document. Unfortunately, this document is about as plain as it can get — it doesn't really tell us anything. Well, for our example, we're working on describing order information, so we might want to start putting in some information that is descriptive of an order. Let's start with an `Order` tag:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <Order/>
</root>
```

OK — so this is getting monotonous — isn't it? We now know that we have one order in our XML document, but we still don't know anything about it. Let's expand on it some by adding a few attributes:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <Order CustomerID="ALFKI" OrderID="10643" OrderDate="1997-08-25T00:00:00" />
</root>
```

Well, it doesn't really take a rocket scientist at this point to be able to discern the basics about our order at this point:

- ❑ The customer's ID number is ALFKI.
- ❑ The order ID number was 10643.
- ❑ The order was placed on August 25, 1997.

Basically, as we have things, it equates out to a row in the `Orders` table in Northwind in SQL Server. If the customer had several orders, it might look something like:


```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <Order CustomerID="ALFKI" OrderID="10643" OrderDate="1997-08-25T00:00:00" />
  <Order CustomerID="ALFKI" OrderID="10692" OrderDate="1997-10-03T00:00:00" />
  <Order CustomerID="ALFKI" OrderID="10702" OrderDate="1997-10-13T00:00:00" />
  <Order CustomerID="ALFKI" OrderID="10835" OrderDate="1998-01-15T00:00:00" />
</root>
```

While this is a perfectly legal—and even well formed—example of XML, it doesn’t really represent the hierarchy of the data as we might wish. We might, for example, wish to build our XML a little differently, and represent the notion that customers are usually considered to be higher in the hierarchical chain (they are the “parent” to orders if you will). We could represent this by changing the way we express customers. Instead of an attribute, we could make it an element in its own right—including having its own attributes—and nest that particular customer’s orders inside the customer element:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
    <Order OrderID="10643" OrderDate="1997-08-25T00:00:00" />
    <Order OrderID="10692" OrderDate="1997-10-03T00:00:00" />
    <Order OrderID="10702" OrderDate="1997-10-13T00:00:00" />
    <Order OrderID="10835" OrderDate="1998-01-15T00:00:00" />
  </Customer>
</root>
```

If we have more than one customer, that’s not a problem—we just add another customer node:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
    <Order OrderID="10643" OrderDate="1997-08-25T00:00:00" />
    <Order OrderID="10692" OrderDate="1997-10-03T00:00:00" />
    <Order OrderID="10702" OrderDate="1997-10-13T00:00:00" />
    <Order OrderID="10835" OrderDate="1998-01-15T00:00:00" />
  </Customer>
  <Customer CustomerID="ANTON" CompanyName="Antonio Moreno Taquería">
    <Order OrderID="10365" OrderDate="1996-11-27T00:00:00" />
    <Order OrderID="10507" OrderDate="1997-04-15T00:00:00" />
    <Order OrderID="10535" OrderDate="1997-05-13T00:00:00" />
    <Order OrderID="10573" OrderDate="1997-06-19T00:00:00" />
  </Customer>
</root>
```

Indeed, this can go to unlimited levels of hierarchy (subject, of course, to whatever your parser can handle). We could, for example, add a level for individual line items in the order.

Determining Elements vs. Attributes

The first thing to understand here is that there is no hard and fast rule for determining what should be an element vs. an attribute. An attribute describes something of the properties of the element that it is an attribute of. Child elements—or child “nodes”—of an element do much the same thing. So how, then, do we decide which should be? Why are attributes even necessary? Well, like most things in life, it’s something of a balancing act.

Attributes make a lot of sense in situations where the value is a one to one relationship with, and is inherently part of, the element. In Northwind, for example, we have only one company name per customer ID — this is ideal for an attribute. As we are transforming our relational data to XML, the columns of a table will often make good attributes to an element directly related to individual rows of a table.

Elements tend to make more sense if there is more of a one to many relationship between the element and what's describing it. In our example earlier in the chapter, there are many orders for each customer. Technically speaking, we could have had each order be an attribute of a customer element, but then we would have needed to repeat much of the customer element information over and over again. Similarly, if our Northwind database allowed for the notion of customers having aliases (multiple names), then we may have wanted to have a "name" element under the customer, and have its attribute describe individual instances of names.

Whichever way you go here, stick to one rule I've emphasized many times throughout the book — be consistent. Once something of a given nature is defined as being an attribute in one place, lean toward keeping it an attribute in other places you use it unless its nature is somehow different in the new place you're using it. One more time: Be consistent.

Namespaces

With all this freedom to create your own tags, to mix and match data from other sources, and just otherwise do your own thing, there are bound to be a few collisions here and there about what things are going to use what names in which places. For example, an element with a name of "letter" might have entirely different structure, rules, and meaning to an application built for libraries (for whom a letter is probably a document written from one person to another person) than it would to another application, say one describing fonts (which might describe the association between a character set and a series of glyphs).

To take this example a little further, the nature of XML is such that industry organizations around the world are slowly agreeing on naming and structure conventions to describe various information in their industry. Library organizations may have agreed on element formats describing books, plays, movies, letters, essays, and so on. At the same time, the operating systems and/or graphics industries may have agreed on element formats describing pictures, fonts and document layouts.

Now, imagine that we, the poor hapless developers that we are, have been asked to write an application that needs to render library content. Obviously, library content makes frequent use of things like fonts — so, when you refer to something called "letter" in our XML, are you referring to something to do with the font or is it a letter from a person to another person (say, from Thomas Jefferson to George Washington)? We have a conflict, and we need a way to resolve it.

That's where *namespaces* come in. Namespaces describe a domain of elements and attributes and what their structure is. The structure that supports letters in libraries would be described in a libraries namespace. Likewise, the graphics industry would likely have their own namespace(s) that would describe letters as they relate to that industry. The information for a namespace is stored in a reference document, and can be found using a Uniform Resource Identifier (URI) — a special name, not dissimilar from a URL — that will eventually resolve to our reference document.

Chapter 16

When we build our XML documents that refer to both library and graphics constructs, then we simply reference the namespaces for those industries. In addition, we qualify elements and attributes whose nature we want described by the namespace. By qualifying our names using namespaces, we make sure that, even if a document has elements that are structurally different but have the same name, we can refer to the parts of our document with complete confidence that we are not referring to the wrong kind of element or attribute.

To reference a namespace to the entire document, we simply add the reference as a special attribute (called `xmlns`) to our root. The reference we all will provide both a local name (how we want to refer to the namespace) and the URI that will eventually resolve to our reference document. We can also add namespace references (again, using `xmlns`) to other nodes in the document if we want to apply only that particular namespace within the scope of the node we assign the namespace, too.

What follows is an example of an XML document (technically, this is what we call a schema") that we will be utilizing in our next chapter. Notice several things about it as relates to namespaces:

The document references three namespaces — one each for XDR (this happens to be an XDR document), a Microsoft datatype namespace (this one builds a list about the number and nature of different datatypes), and, last, but not least, a special SQL namespace used for working with SQL Server XML integration.

Some attributes (including one in the root) are qualified with namespace information (see the `sql:relation` attribute, for example).

```
<?xml version="1.0" encoding="UTF-8"?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:dt="urn:schemas-microsoft-com:datatypes"
        xmlns:sql="urn:schemas-microsoft-com:xml-sql"
        sql:xsl='../Customers.xsl'>
  <ElementType name="Root" content="empty" />
  <ElementType name="Customers" sql:relation="Customers">
    <AttributeType name="CustomerID"/>
    <AttributeType name="CompanyName"/>
    <AttributeType name="Address"/>
    <AttributeType name="City"/>
    <AttributeType name="Region"/>
    <AttributeType name="PostalCode"/>
    <attribute type="CustomerID" sql:field="CustomerID"/>
    <attribute type="CompanyName" sql:field="CompanyName"/>
    <attribute type="Address" sql:field="Address"/>
    <attribute type="City" sql:field="City"/>
    <attribute type="Region" sql:field="Region"/>
    <attribute type="PostalCode" sql:field="PostalCode"/>
  </ElementType>
</Schema>
```

The `sql` datatype references a couple of special attributes. We do not have to worry about whether the Microsoft datatypes namespace also has a field or relation datatype because we are fully qualifying our attribute names. Even if the datatypes namespace does have an attribute called `field`, our XML parser will still know to treat this element by the rules of the `sql` namespace.

Element Content

Another notion of XML and elements that deserves mention (which is close to all it will get here) is the concept of element content.

Elements can contain data beyond the attribute level and nested elements. While nested elements are certainly one form of element content (one element contains the other), XML also allows for row text information to be contained in an element. For example, we could have an XML document that looked something like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
    <Note Date="1997-08-25T00:00:00">
      The customer called in today and placed another order. Says they really like our
      work and would like it if we would consider establishing a location closer to their
      base of operations.
    </Note>
    <Note Date="1997-08-26T00:00:00">
      Followed up with the customer on new location. Customer agrees to guarantee us
      $5,000 per month in business to help support a new office.
    </Note>
  </Customer>
</root>
```

The contents of the Note elements, such as “The customer called . . .” are neither an element nor an attribute, yet they are valid XML data.

Be aware that such data exists in XML, but SQL Server will not output data in this format natively. The row/column approach of RDBMS systems lends itself far stronger to elements and attributes. To output data such as our notes above, you would need to transform the data into the new format. We will look at Transformations as the last item in this chapter.

Being Valid vs. Being Well Formed — Schemas and DTDs

Just because an XML document is well formed does not mean that it is *valid XML*. Now, while that is sinking in on you I’ll tell you that XML, *no* XML, is considered “valid” unless it has been validated against some form of specification document. Currently, there are only two recognized types of specification documents — a Document Type Definition, or DTD, and an XML schema.

The basic premise behind both varieties of validation documents is much the same. While XML hopes to define the most basic tenants of what a XML document will look like, DTDs and XML schemas seek to define what the rules are for a particular *class* of XML document. The two approaches are implemented somewhat differently and each offers distinct advantages over the other:

- ❑ **DTDs:** This is the old tried and true. DTDs are utilized in SGML (XML is an SGML application — you can think of SGML being a superset of XML, but incredibly painful to learn), and have the advantage of being a very well-known and accepted way of doing things. There are tons of DTDs already out there that are just waiting for you to utilize them.

The downside (you knew there had to be one — right?) is that the “old” is operative in my “old tried and true” statement above. Not that being old is a bad thing, but in this case, DTDs are definitely not up to speed with what else has happened in document technology. DTDs do not really allow for such seemingly rudimentary things as restricting datatypes.

- ❑ **XML schemas:** XML schemas have the distinct advantage of being strongly typed. What’s cool about them is that you can effectively establish your own complex datatypes — types that are made up based on combinations of one or more other datatypes (including other complex datatypes) or require specialized pattern matching (for example, a Social Security number is just a number, but it has special formatting that you could easily enforce via an XML schema). XML schemas also have the advantage, as their name suggests, of being an XML document themselves. This means that a lot of the skills in writing your XML documents also apply to writing schemas (though there’s still plenty to learn) and that schemas can, themselves, be self describing — right down to validating themselves against yet another schema.

What SQL Server Brings to the Party

So, now we have all the basics of what XML *is* down. What we need is to understand the relevance in SQL Server.

XML functionality was a relatively late addition to SQL Server. Indeed, it first appeared as a downloadable add-on to SQL Server 7.0. What’s more, a significant part of the functionality was more an addition to Internet Information Server (IIS) than to SQL Server.

With SQL Server 2000, the XML side of thing moved to what Microsoft called a “Web Release” model, and was updated several times. Now, with SQL Server 2005, XML finishes moving into the core product. While all the old functionality remains supported, SQL Server 2005 adds more core feature set that makes XML an integral part of things rather than the afterthought that XML sometimes seemed to be in prior releases.

What functionality? Well, SQL Server 2005 has several places where XML comes to the forefront:

- ❑ Support for multiple methods of selecting data out of normal columns and receiving them in XML format
- ❑ Support for storing XML data natively within SQL Server
- ❑ Support for querying data that is stored in its original XML format using XQuery
- ❑ Support for enforcing data integrity in the data being stored in XML format using XML schemas
- ❑ Support for indexing XML data

And this is just the mainstream stuff.

The support for each of the above often makes use of several functional areas of XML support, so let’s look at XML support one piece at a time.

Retrieving Relational Data in XML Format

This is the area that SQL Server already had largely figured out prior to the 2005 release. We had a 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 resultset. 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[, XMLDATA][, ELEMENTS][, BINARY base64]|PATH]
[OPTION (<query hint>, [, ...n))]
```

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 three 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 in the end. 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 to SQL Server 2005 to try and 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 or have SQL Server do it for you (using the ROOT option described below). 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 four major formatting options, there are five other optional parameters that further modify the output that SQL Server provides in an XML query:

- ❑ **XMLDATA:** 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 datatypes) and rules of the XML data that follows. Keep in mind that this schema, while similar, is not an exact match to the W3C spec as it is proposed at the time of this writing.
- ❑ **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 datatype 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 ALFKI and ANTON. After cruising through just the first five or so chapters of the book, you would probably say “No problem!” and supply something like:

```
USE Northwind

SELECT Customers.CustomerID,
       Customers.CompanyName,
       Orders.OrderID,
       Orders.OrderDate
FROM Customers
JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
```

So, you go hand your boss the results:

ANTON	Antonio Moreno Taquería	10365	1996-11-27 00:00:00.000
ANTON	Antonio Moreno Taquería	10507	1997-04-15 00:00:00.000
...			
...			
ALFKI	Alfreds Futterkiste	11081	2000-07-22 00:00:00.000
ALFKI	Alfreds Futterkiste	11087	2000-08-05 17:37:52.520

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:

```
USE Northwind

SELECT Customers.CustomerID,
       Customers.CompanyName,
       Orders.OrderID,
       Orders.OrderDate
FROM Customers
JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
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 resultset had we run just a standard SQL query:

```
<row CustomerID="ANTON" CompanyName="Antonio Moreno Taquería" OrderID="10365"
OrderDate="1996-11-27T00:00:00" />
<row CustomerID="ANTON" CompanyName="Antonio Moreno Taquería" OrderID="10507"
OrderDate="1997-04-15T00:00:00" />
<row CustomerID="ANTON" CompanyName="Antonio Moreno Taquería" OrderID="10535"
OrderDate="1997-05-13T00:00:00" />
<row CustomerID="ANTON" CompanyName="Antonio Moreno Taquería" OrderID="10573"
OrderDate="1997-06-19T00:00:00" />
<row CustomerID="ALFKI" CompanyName="Alfreds Futterkiste" OrderID="10643"
OrderDate="1997-08-25T00:00:00" />
<row CustomerID="ANTON" CompanyName="Antonio Moreno Taquería" OrderID="10677"
OrderDate="1997-09-22T00:00:00" />
<row CustomerID="ANTON" CompanyName="Antonio Moreno Taquería" OrderID="10682"
OrderDate="1997-09-25T00:00:00" />
<row CustomerID="ALFKI" CompanyName="Alfreds Futterkiste" OrderID="10692"
OrderDate="1997-10-03T00:00:00" />
```



```
<row CustomerID="ALFKI" CompanyName="Alfreds Futterkiste" OrderID="10702"
OrderDate="1997-10-13T00:00:00" />
<row CustomerID="ALFKI" CompanyName="Alfreds Futterkiste" OrderID="10835"
OrderDate="1998-01-15T00:00:00" />
<row CustomerID="ANTON" CompanyName="Antonio Moreno Taquería" OrderID="10856"
OrderDate="1998-01-28T00:00:00" />
<row CustomerID="ALFKI" CompanyName="Alfreds Futterkiste" OrderID="10952"
OrderDate="1998-03-16T00:00:00" />
<row CustomerID="ALFKI" CompanyName="Alfreds Futterkiste" OrderID="11011"
OrderDate="1998-04-09T00:00:00" />
```

Be aware that Management Studio will truncate any column where the length exceeds the number set in the Options menu in the Results to Text tab (maximum is 8192). This issue exists in the results window (grid or text) 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 for example), you shouldn't encounter an issue with this.

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 our data — orders are only placed 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 us — 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.

```
USE Northwind

SELECT Customers.CustomerID,
Customers.CompanyName,
    Orders.OrderID,
    Orders.OrderDate
FROM Customers
JOIN Orders
    ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
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, but, another even more significant difference appears when we look at the XML more thoroughly (I have again cleaned up the output a bit for clarity):

```
<Customers CustomerID="ANTON" CompanyName="Antonio Moreno Taquería">
  <Orders OrderID="10365" OrderDate="1996-11-27T00:00:00" />
  <Orders OrderID="10507" OrderDate="1997-04-15T00:00:00" />
```

```

    <Orders OrderID="10535" OrderDate="1997-05-13T00:00:00" />
    <Orders OrderID="10573" OrderDate="1997-06-19T00:00:00" />
  </Customers>
  <Customers CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
    <Orders OrderID="10643" OrderDate="1997-08-25T00:00:00" />
  </Customers>
  <Customers CustomerID="ANTON" CompanyName="Antonio Moreno Taquería">
    <Orders OrderID="10677" OrderDate="1997-09-22T00:00:00" />
    <Orders OrderID="10682" OrderDate="1997-09-25T00:00:00" />
  </Customers>
  <Customers CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
    <Orders OrderID="10692" OrderDate="1997-10-03T00:00:00" />
    <Orders OrderID="10702" OrderDate="1997-10-13T00:00:00" />
    <Orders OrderID="10835" OrderDate="1998-01-15T00:00:00" />
  </Customers>
  <Customers CustomerID="ANTON" CompanyName="Antonio Moreno Taquería">
    <Orders OrderID="10856" OrderDate="1998-01-28T00:00:00" />
  </Customers>
  <Customers CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
    <Orders OrderID="10952" OrderDate="1998-03-16T00:00:00" />
    <Orders OrderID="11011" OrderDate="1998-04-09T00:00:00" />
  </Customers>

```

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 `Orders` elements are nested inside our `Customers` elements. If a column from the `Orders` table were listed first in our select list, then `Customers` would be nested inside `Orders`.

Pay attention to this business of the ordering of your `SELECT` list! In our last chapter, I showed examples of XML with `Customers` being the parent of `Products`. I also showed you how we could style that same data into a different hierarchy — `Products` being the parent of `Customers`. Think about the primary question your XML query is meant to solve. 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 (e.g. `Orders` sorted within `Customers`).

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

Chapter 16

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 resultset is that you must provide sufficient metadata right within your resultset such that SQL Server can then transform that resultset 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	Customer!1! CustomerID	Customer!1! CompanyName	Order!2! OrderID	Order!2!OrderDate
1	NULL	ALFKI	Alfreds Futterkiste	NULL	NULL
2	1	ALFKI	Alfreds Futterkiste	10643	1997-08-25 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	10692	1997-10-03 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	10702	1997-10-13 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	10835	1998-01-15 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	10952	1998-03-16 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	11011	1998-04-09 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	11078	1999-05-01 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	11079	NULL
2	1	ALFKI	Alfreds Futterkiste	11080	2000-07-22 16:48:00.000
2	1	ALFKI	Alfreds Futterkiste	11081	2000-07-22 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	11087	2000-08-05 17:37:52.520
1	NULL	ANTON	Antonio Moreno Taquería	NULL	NULL
2	1	ANTON	Antonio Moreno Taquería	10365	1996-11-27 00:00:00.000
2	1	ANTON	Antonio Moreno Taquería	10507	1997-04-15 00:00:00.000
2	1	ANTON	Antonio Moreno Taquería	10535	1997-05-13 00:00:00.000

Tag	Parent	Customer!1! CustomerID	Customer!1! CompanyName	Order!2! OrderID	Order!2!OrderDate
2	1	ANTON	Antonio Moreno Taquería	10573	1997-06-19 00:00:00.000
2	1	ANTON	Antonio Moreno Taquería	10677	1997-09-22 00:00:00.000
2	1	ANTON	Antonio Moreno Taquería	10682	1997-09-25 00:00:00.000
2	1	ANTON	Antonio Moreno Taquería	10856	1998-01-28 00:00:00.000

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 with 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

We saw in our last chapter that elements can contain other elements, but that they cannot overlap. This means that 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 that we separate them by an exclamation mark (!).

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 we 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	Customer!1! CustomerID	Customer!1! CompanyName	Order!2! OrderID	Order!2! OrderDate
1	NULL	ALFKI	Alfreds Futterkiste	NULL	NULL

it can look at the tag number, see that it is 1, and know that it should process Customer!1!CustomerID and Customer!1!CompanyName, but that it doesn't have to process Order!2!OrderID, 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 a 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 the following (yuck):

```
USE Northwind

SELECT 1
        NULL
        Customers.CustomerID
        Customers.CompanyName
        NULL
        NULL
FROM Customers
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'

UNION ALL

SELECT 2,
        1,
        Customers.CustomerID,
        Customers.CompanyName,
        Orders.OrderID,
        Orders.OrderDate
FROM Customers
JOIN Orders
ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
ORDER BY [Customer!1!CustomerID], [Order!2!OrderID]
FOR XML EXPLICIT
```

Notice that we use the `FOR XML` clause only once — after the last query in the `UNION`.

Chapter 16

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 `CompanyName` information to be an attribute of the `Order` rather than (or, if we wished, in addition to) the `Customer` element? With `AUTO`, we would need some trickery in order to get this (for every row, we would need to look up the `Company` 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 `CompanyName` in our `SELECT` list again, but associate the new instance of it with `Orders` instead of `Customers`:

```
USE Northwind

SELECT 1
      NULL
      Customers.CustomerID
      Customers.CompanyName
      NULL
      NULL
      NULL
      as Tag,
      as Parent,
      as [Customer!1!CustomerID],
      as [Customer!1!CompanyName],
      as [Order!2!OrderID],
      as [Order!2!OrderDate],
      as [Order!2!CompanyName]
FROM Customers
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'

UNION ALL

SELECT 2,
      1,
      Customers.CustomerID,
      Customers.CompanyName,
      Orders.OrderID,
      Orders.OrderDate,
      Customers.CompanyName
FROM Customers
JOIN Orders
ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
ORDER BY [Customer!1!CustomerID], [Order!2!OrderID]
FOR XML EXPLICIT
```

Execute this, and we get pretty much the same results as before, only this time we received the additional attribute we were looking for in our `Order` element.

```
<Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
  <Order OrderID="10643" OrderDate="1997-08-25T00:00:00" CompanyName="Alfreds
Futterkiste" />
  <Order OrderID="10692" OrderDate="1997-10-03T00:00:00" CompanyName="Alfreds
Futterkiste" />
  <Order OrderID="10702" OrderDate="1997-10-13T00:00:00" CompanyName="Alfreds
Futterkiste" />
```

```

    <Order OrderID="10835" OrderDate="1998-01-15T00:00:00" CompanyName="Alfreds
Futterkiste" />
    <Order OrderID="10952" OrderDate="1998-03-16T00:00:00" CompanyName="Alfreds
Futterkiste" />
    <Order OrderID="11011" OrderDate="1998-04-09T00:00:00" CompanyName="Alfreds
Futterkiste" />
    <Order OrderID="11078" OrderDate="1999-05-01T00:00:00" CompanyName="Alfreds
Futterkiste" />
    <Order OrderID="11079" CompanyName="Alfreds Futterkiste" />
    <Order OrderID="11080" OrderDate="2000-07-22T16:48:00" CompanyName="Alfreds
Futterkiste" />
    <Order OrderID="11081" OrderDate="2000-07-22T00:00:00" CompanyName="Alfreds
Futterkiste" />
    <Order OrderID="11087" OrderDate="2000-08-05T17:37:52.520" CompanyName="Alfreds
Futterkiste" />
  </Customer>
<Customer CustomerID="ANTON" CompanyName="Antonio Moreno Taquería">
  <Order OrderID="10365" OrderDate="1996-11-27T00:00:00" CompanyName="Antonio
Moreno Taquería" />
  <Order OrderID="10507" OrderDate="1997-04-15T00:00:00" CompanyName="Antonio
Moreno Taquería" />
  <Order OrderID="10535" OrderDate="1997-05-13T00:00:00" CompanyName="Antonio
Moreno Taquería" />
  <Order OrderID="10573" OrderDate="1997-06-19T00:00:00" CompanyName="Antonio
Moreno Taquería" />
  <Order OrderID="10677" OrderDate="1997-09-22T00:00:00" CompanyName="Antonio
Moreno Taquería" />
  <Order OrderID="10682" OrderDate="1997-09-25T00:00:00" CompanyName="Antonio
Moreno Taquería" />
  <Order OrderID="10856" OrderDate="1998-01-28T00:00:00" CompanyName="Antonio
Moreno Taquería" />
</Customer>

```

Notice the fact that our `OrderDate` attribute is completely missing—you'll see that on any row where the value was `NULL`. XML represents null values by the non-existence of those values.

This example is really just for starters. We can utilize *directives* to achieve far more flexibility — shaping and controlling both our data and our schema output (if we 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, although they can still be confusing at times (some of them work pretty counter-intuitively 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.

Chapter 16

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 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:

```
SELECT 1, NULL, NULL, Customers.CustomerID, Customers.CompanyName, NULL, NULL
        as Tag,
        as Parent,
        as [Customer!1!CustomerID],
        as [Customer!1!CompanyName],
        as [Order!2!OrderID],
        as [Order!2!OrderDate!element]
FROM Customers
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'

UNION ALL

SELECT 2,
       1,
       Customers.CustomerID,
       Customers.CompanyName,
       Orders.OrderID,
       Orders.OrderDate
FROM Customers
JOIN Orders
ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
ORDER BY [Customer!1!CustomerID], [Order!2!OrderID]
FOR XML EXPLICIT
```

Suddenly, we have an extra element instead of an attribute:

```
<Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
  <Order OrderID="10643">
    <OrderDate>1997-08-25T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10692">
    <OrderDate>1997-10-03T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10702">
    <OrderDate>1997-10-13T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10835">
    <OrderDate>1998-01-15T00:00:00</OrderDate>
```

```

    </Order>
    <Order OrderID="10952">
      <OrderDate>1998-03-16T00:00:00</OrderDate>
    </Order>
    <Order OrderID="11011">
      <OrderDate>1998-04-09T00:00:00</OrderDate>
    </Order>
  </Customer>
  <Customer CustomerID="ANTON" CompanyName="Antonio Moreno Taquería">
    <Order OrderID="10365">
      <OrderDate>1996-11-27T00:00:00</OrderDate>
    </Order>
    <Order OrderID="10507">
      <OrderDate>1997-04-15T00:00:00</OrderDate>
    </Order>
    <Order OrderID="10535">
      <OrderDate>1997-05-13T00:00:00</OrderDate>
    </Order>
    <Order OrderID="10573">
      <OrderDate>1997-06-19T00:00:00</OrderDate>
    </Order>
    <Order OrderID="10677">
      <OrderDate>1997-09-22T00:00:00</OrderDate>
    </Order>
    <Order OrderID="10682">
      <OrderDate>1997-09-25T00:00:00</OrderDate>
    </Order>
    <Order OrderID="10856">
      <OrderDate>1998-01-28T00:00:00</OrderDate>
    </Order>
  </Customer>

```

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 only be seen 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.

Chapter 16

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 OrderIDs 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 CategoryID, but we do not want the CategoryID 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
        ProductID
        CategoryID
        NULL
        NULL
FROM Products
        as Tag,
        as Parent,
        as [Product!1!ProductID],
        as [Product!1!CategoryID],
        as [Order!2!OrderID],
        as [Order!2!OrderDate]

UNION ALL

SELECT 2,
        1,
        p.ProductID,
        p.CategoryID,
        od.OrderID,
        o.OrderDate
FROM Products AS p
JOIN [Order Details] AS od
    ON p.ProductID = od.ProductID
JOIN Orders AS o
    ON od.OrderID = o.OrderID
WHERE o.OrderDate BETWEEN '1998-01-01' AND '1998-01-07'
ORDER BY [Product!1!CategoryID], [Product!1!ProductID], [Order!2!OrderID]
FOR XML EXPLICIT
```

Be sure and check out the way we dealt with the OrderDate on this one. Even though I needed to fetch that information out of the Orders table, it was easy (since we're using EXPLICIT anyway) to combine that information with the OrderID from the Order Details table. As it happens, I could have also just grabbed the OrderID from the Orders 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="1" CategoryID="1" />
<Product ProductID="2" CategoryID="1">
<Order OrderID="10813" OrderDate="1998-01-05T00:00:00" />
</Product>
<Product ProductID="24" CategoryID="1" />
<Product ProductID="34" CategoryID="1" />
<Product ProductID="35" CategoryID="1" />
<Product ProductID="38" CategoryID="1">
<Order OrderID="10816" OrderDate="1998-01-06T00:00:00" />
<Order OrderID="10817" OrderDate="1998-01-06T00:00:00" />
</Product>
```

```

<Product ProductID="39" CategoryID="1" />
<Product ProductID="43" CategoryID="1">
<Order OrderID="10814" OrderDate="1998-01-05T00:00:00" />
<Order OrderID="10819" OrderDate="1998-01-07T00:00:00" />
</Product>
<Product ProductID="67" CategoryID="1" />
<Product ProductID="70" CategoryID="1">
<Order OrderID="10810" OrderDate="1998-01-01T00:00:00" />
</Product>
<Product ProductID="75" CategoryID="1">
<Order OrderID="10819" OrderDate="1998-01-07T00:00:00" />
</Product>
<Product ProductID="76" CategoryID="1">
<Order OrderID="10808" OrderDate="1998-01-01T00:00:00" />
</Product>
<Product ProductID="3" CategoryID="2" />
<Product ProductID="4" CategoryID="2" />
...

```

Note that I've trimmed my results a bit for brevity.

Now we'll add our `hide` directive and get rid of the category information:

```

SELECT 1                                as Tag,
      NULL                                as Parent,
      ProductID                          as [Product!1!ProductID],
      CategoryID                         as [Product!1!CategoryID!hide],
      NULL                                as [Order!2!OrderID],
      NULL                                as [Order!2!OrderDate]
FROM Products

UNION ALL

SELECT 2,
      1,
      p.ProductID,
      p.CategoryID,
      od.OrderID,
      o.OrderDate
FROM Products AS p
JOIN [Order Details] AS od
  ON p.ProductID = od.ProductID
JOIN Orders AS o
  ON od.OrderID = o.OrderID
WHERE o.OrderDate BETWEEN '1998-01-01' AND '1998-01-07'
ORDER BY [Product!1!CategoryID!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="1" />
<Product ProductID="2">
<Order OrderID="10813" OrderDate="1998-01-05T00:00:00" />

```

```
</Product>
<Product ProductID="24" />
<Product ProductID="34" />
<Product ProductID="35" />
<Product ProductID="38">
<Order OrderID="10816" OrderDate="1998-01-06T00:00:00" />
<Order OrderID="10817" OrderDate="1998-01-06T00:00:00" />
</Product>
<Product ProductID="39" />
<Product ProductID="43">
<Order OrderID="10814" OrderDate="1998-01-05T00:00:00" />
<Order OrderID="10819" OrderDate="1998-01-07T00:00:00" />
</Product>
<Product ProductID="67" />
<Product ProductID="70">
<Order OrderID="10810" OrderDate="1998-01-01T00:00:00" />
</Product>
<Product ProductID="75">
<Order OrderID="10819" OrderDate="1998-01-07T00:00:00" />
</Product>
<Product ProductID="76">
<Order OrderID="10808" OrderDate="1998-01-01T00:00:00" />
</Product>
<Product ProductID="3" />
<Product ProductID="4" />
...
```

id, idref, and idrefs

None of these three has any effect 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.

Note, however, that unlike Primary Keys in SQL, you cannot have multiple attributes make up your `id` in XML.

Because 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 multi-valued, white space 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
        NULL
        ProductID
        CategoryID
        NULL
        NULL
        NULL
FROM Products
        as Tag,
        as Parent,
        as [Product!1!ProductID!ID],
        as [Product!1!CategoryID!hide],
        as [Order!2!OrderID],
        as [Order!2!ProductID!idref],
        as [Order!2!OrderDate]

UNION ALL

SELECT 2,
        1,
        p.ProductID,
        p.CategoryID,
        od.OrderID,
        od.ProductID,
        o.OrderDate
FROM Products AS p
JOIN [Order Details] AS od
ON p.ProductID = od.ProductID
JOIN Orders AS o
ON od.OrderID = o.OrderID
WHERE o.OrderDate BETWEEN '1998-01-01' AND '1998-01-07'
ORDER BY [Product!1!CategoryID!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="Schema2" 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" />
  </ElementType>
</Schema>

```

```
<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:#Schema2" ProductID="2">
  <Order OrderID="10813" ProductID="2" OrderDate="1998-01-05T00:00:00"/>
</Product>
```

In this case, notice that SQL Server has referenced our in-line 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`. In addition, 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
        ProductID
        CategoryID
        NULL
        NULL
        NULL
        NULL
        as Tag,
        as Parent,
        as [Product!1!ProductID],
        as [Product!1!CategoryID!hide],
        as [Product!1!OrderList!idrefs],
        as [Order!2!OrderID!id],
        as [Order!2!OrderDate]
FROM Products

UNION ALL

SELECT 1,
        NULL,
        p.ProductID,
        p.CategoryID,
        'id' + CAST(od.OrderID AS varchar),
        NULL,
        NULL
FROM Products AS p
JOIN [Order Details] AS od
    ON p.ProductID = od.ProductID
JOIN Orders AS o
    ON od.OrderID = o.OrderID
WHERE o.OrderDate BETWEEN '1998-01-01' AND '1998-01-07'
```

```

UNION ALL

SELECT 2,
    1,
    p.ProductID,
    p.CategoryID,
    NULL,
    'id' + CAST(od.OrderID AS varchar),
    o.OrderDate
FROM Products AS p
JOIN [Order Details] AS od
    ON p.ProductID = od.ProductID
JOIN Orders AS o
    ON od.OrderID = o.OrderID
WHERE o.OrderDate BETWEEN '1998-01-01' AND '1998-01-07'
ORDER BY [Product!1!CategoryID!hide], [Order!2!OrderID!id],
[Product!1!OrderList!idrefs]
FOR XML EXPLICIT, XMLDATA

```

The schema winds up looking an awful lot like the one we got for idref:

```

<Schema name="Schema9" 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:#Schema9" ProductID="76" OrderList="id10808 id10810
id10813 id10814 id10816 id10817 id10819 id10819">
  <Order OrderID="id10808" OrderDate="1998-01-01T00:00:00"/>
  <Order OrderID="id10810" OrderDate="1998-01-01T00:00:00"/>
  <Order OrderID="id10813" OrderDate="1998-01-05T00:00:00"/>
  <Order OrderID="id10814" OrderDate="1998-01-05T00:00:00"/>
  <Order OrderID="id10816" OrderDate="1998-01-06T00:00:00"/>
  <Order OrderID="id10817" OrderDate="1998-01-06T00:00:00"/>
  <Order OrderID="id10819" OrderDate="1998-01-07T00:00:00"/>
  <Order OrderID="id10819" OrderDate="1998-01-07T00:00:00"/>
</Product>

```

Using `id`, `idref`, and `idrefs` is very complex. Still, they allow us to make our 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 (“OK, 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, then 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. Basically, it 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 for xml-specific purposes. You would use `cdata` any time 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). Last, but not least, you can consider moving binary information in your XML document by character encoding it (using MIME, Base64, or the encoding scheme of your choice) and wrapping the results in a CDATA.

For this one, we’ll just take a simple example—the `Northwind Employees` table. This table has a field that has a text datatype. The contents are basically unknown. A query to generate the notes on employees into XML might look something like the following:

```
SELECT 1                                as Tag,
        NULL                             as Parent,
        Employees.EmployeeID             as [Employee!1!EmployeeID],
        Employees.Notes                   as [Employee!1!!cdata]
FROM Employees
ORDER BY [Employee!1!EmployeeID]
FOR XML EXPLICIT
```

The output is pretty straightforward:

```
<Employee EmployeeID="1">
<![CDATA[
Education includes a BA in psychology from Colorado State University in 1970. She
also completed "The Art of the Cold Call." Nancy is a member of Toastmasters
International.
]]>
</Employee>
<Employee EmployeeID="2">
<![CDATA[
Andrew received his BTS commercial in 1974 and a Ph.D. in international marketing
from the University of Dallas in 1981. He is fluent in French and Italian and
reads German. He joined the company as a sales representative, was promoted to
sales manager in January 1992 and to vice president of sales in March 1993. Andrew
is a member of the Sales Management Roundtable, the Seattle Chamber of Commerce,
and the Pacific Rim Importers Association.
]]>
</Employee>
<Employee EmployeeID="3">
<![CDATA[
Janet has a BS degree in chemistry from Boston College (1984). She has also
completed a certificate program in food retailing management. Janet was hired as a
sales associate in 1991 and promoted to sales representative in February 1992.
]]>
</Employee>
```

If you look at `EmployeeID="1"`, you'll see right away that there is indeed no encoding going on (otherwise the `"` symbols would be encoded to `"`).

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 any 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're needing 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 pointed 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.

XPath is its own thing, and there are entire books dedicated to just that topic. PATH utilizes a wide variety of what's available in XPath, and so there really is too much to cover here for a single chapter in a beginning text. That said, we're going to touch on the basics here, and give you a taste of the more advanced stuff in the next section. From there, it's really up to you whether you want to learn XPath more fully, and from there, what pieces of it are understood by PATH. More advanced coverage of this is also supplied in the next book in this series: Professional SQL Server 2005 Programming.

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:

```
USE Northwind

SELECT Customers.CustomerID,
       COUNT(Orders.OrderID)
FROM Customers
JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
GROUP BY Customers.CustomerID
FOR XML PATH
```

Check the output from this:

```
<row><CustomerID>ALFKI</CustomerID>7</row>
<row><CustomerID>ANTON</CustomerID>7</row>
```

What it created is a row element for each row in the query — much as we had with RAW — but notice the difference in how it treated our column data.

Since the `CustomerID` column was named, it was placed in its own element (we'll explore this more in our next section) — notice, however, the number 7 in my results. This is just loose embedded text for the row element — it isn't even associated directly with the `CustomerID` since it is outside the `CustomerID` element.

I feel like I'm repeating myself for the five thousandth time by saying this, but, again, remember that the exact counts (7s in my case) that come back may vary on your system depending on how much you have been playing with the data in the Orders table. The key thing is to see how the counts are not associated with the CustomerID, but are instead just raw text associated with the row.

My personal slant on this is that the 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 their 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><CustomerID>ALFKI</CustomerID>7</row>
```

Our `CustomerID` 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 `CustomerID` to be an attribute of the top element for the row:

```
SELECT Customers.CustomerID AS '@CustomerID',
       COUNT(Orders.OrderID)
FROM Customers
JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
GROUP BY Customers.CustomerID
FOR XML PATH
```

This yields:

```
<row CustomerID="ALFKI">7</row>
<row CustomerID="ANTON">7</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 Customers.CustomerID AS '@CustomerID',
       COUNT(Orders.OrderID) AS '@OrderCount'
FROM Customers
JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID
```

Chapter 16

```
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'  
GROUP BY Customers.CustomerID  
FOR XML PATH
```

With this, we no longer have our loose text for the element:

```
<row CustomerID="ALFKI" OrderCount="7" />  
<row CustomerID="ANTON" OrderCount="7" />
```

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 Customers.CustomerID,  
       COUNT(Orders.OrderID) AS '@OrderCount'  
FROM Customers  
JOIN Orders  
    ON Customers.CustomerID = Orders.CustomerID  
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'  
GROUP BY Customers.CustomerID  
FOR XML PATH
```

What I’ve done here is to go back to CustomerID as its own element. What, at first glance, you would expect to happen is to get a CustomerID element with OrderCount as an attribute, but it doesn’t quite work that way:

```
Msg 6852, Level 16, State 1, Line 3  
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 CustomerID?

/

Yes, a forward slash.

Must like @, this special character indicates special things you want done. Essentially, you use it to defined 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 it 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 Customers.CustomerID,  
       COUNT(Orders.OrderID) AS 'CustomerID/OrderCount'  
FROM Customers
```

```

JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
GROUP BY Customers.CustomerID
FOR XML PATH

```

By adding the `"/`, and then placing `CustomerID` before the slash, we are telling SQL Server that `OrderCount` is below `CustomerID` in a hierarchy. Now, there are many ways XML hierarchy can be structured, so let's see what SQL Server does with this:

```

<row><CustomerID>ALFKI<OrderCount>7</OrderCount></CustomerID>
</row><row><CustomerID>ANTON<OrderCount>7</OrderCount></CustomerID></row>

```

Now, if you recall, we wanted to make `OrderCount` an attribute of `CustomerID`, so, while we have `OrderCount` below `CustomerID` 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 Customers.CustomerID,
       COUNT(Orders.OrderID) AS 'CustomerID/@OrderCount'
FROM Customers
JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
GROUP BY Customers.CustomerID
FOR XML PATH

```

Error time:

```

Msg 6852, Level 16, State 1, Line 1
Attribute-centric column 'CustomerID/@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 are wanting 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 (sub elements or raw text).

In our case, we are putting the `CustomerID` as being raw text, but the `OrderCount` as being an attribute (OK, backwards of what would be likely in real life, but hang with me here). This means we are telling SQL Server things backwards. By the time it sees the `OrderCount` information it is already done with attributes for `CustomerID` and can't go back.

So, to fix things for us, we simply need to tell it about the attributes before we tell it about any more elements or raw text:

```

SELECT COUNT(Orders.OrderID) AS 'CustomerID/@OrderCount',
       Customers.CustomerID AS 'CustomerID'
FROM Customers

```

```
JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
GROUP BY Customers.CustomerID
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 right the lower-level information for the CustomerID element. Run it, and you'll see we get what we were after:

```
<row><CustomerID OrderCount="7">ALFKI</CustomerID></row>
<row><CustomerID OrderCount="7">ANTON</CustomerID></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 can 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 the preceding text 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 beginning 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 datatype 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. Such functionality has indeed been added to this release as a match to the FOR XML clause that we've already seen.

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. The syntax is pretty simple:

```
sp_xml_preparedocument @hdoc = <integer variable> OUTPUT,
[, @xmltext = <character data>]
[, @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 (e.g. <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.

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 a path 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 below, 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 above, 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 above. Consumed data should not be copied to the overflow property <code>@mp:xmltext</code> (you would have to use the <code>MetaProperty</code> schema item to retrieve this). If you're not going to use the <code>MetaProperties</code> — and most of the time you won't be — I recommend this option. It cuts a small (OK, <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).

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 so on.
- ❑ The datatype is any valid SQL Server datatype. Because XML can have datatypes 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 above, only for the parent.
- ❑ `@mp:localname`: Provides the non-fully qualified name of the node. It is used with 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 above, 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 above, only for the parent.
- ❑ `@mp:prefix`: Stores the namespace prefix of the current element name.
- ❑ `@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 datatypes) 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!

OK, 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>
```

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!

OK, I'm sure you've been bored 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 `shippers` that they have and our company (`Northwind`) doesn't. A sample of what our script might look like to import these from an XML document might be:

```
USE Northwind

DECLARE @idoc      int
DECLARE @xmldoc    nvarchar(4000)

-- define the XML document
SET @xmldoc = '
<ROOT>
<Shipper ShipperID="100" CompanyName="Billy Bob's Pretty Good Shipping"/>
<Shipper ShipperID="101" CompanyName="Fred's Freight"/>
</ROOT>
'

--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 Shippers

-- ShipperID is an IDENTITY column, so we need to allow direct updates
SET IDENTITY_INSERT Shippers ON

--See our XML data in a tabular format
SELECT * FROM OPENXML (@idoc, '/ROOT/Shipper', 0) WITH (
    ShipperID      int,
    CompanyName     nvarchar(40))

--Perform and insert based on that data
INSERT INTO Shippers
(ShipperID, CompanyName)
SELECT * FROM OPENXML (@idoc, '/ROOT/Shipper', 0) WITH (
    ShipperID      int,
    CompanyName     nvarchar(40))

--Set things back to normal
SET IDENTITY_INSERT Shippers OFF

--Now look at the Shippers table after our insert
SELECT * FROM Shippers

--Now clear the XML document from memory
EXEC sp_xml_removedocument @idoc
```

The final result set from this looks just like what we wanted:

ShipperID	CompanyName	Phone
1	Speedy Express	(503) 555-9831
2	United Package	(503) 555-3199
3	Federal Shipping	(503) 555-9931
4	Speedy Shippers, Inc.	(503) 555-5566
5	Speedy Shippers, Inc	NULL
100	Billy Bob's Pretty Good Shipping	NULL
101	Fred's Freight	NULL
102	Readyship	(503) 555-1234
103	MyShipper	(503) 555-3443

Well, that's all well and good for such an easy example, but let's get a few more of OPENXML's options going.

This time, we'll make use of a subset of an XML block we generated with our FOR XML EXPLICIT clause earlier in the chapter. This time, we want to retrieve the ProductID, CategoryID, OrderID, and OrderDate. To make things a little more interesting, try retrieving the @mp:previous property, too.

Before you get into the code here, it seems like a good time to remind you — remember your root element! SQL Server doesn't put it in for you on FOR XML queries in order to allow you to put multiple queries in one document. You need to manually add it before you try and send it to sp_xml_preparedocument.

```
DECLARE @idoc int
DECLARE @doc varchar(4000)
-- XML that came from our FOR XML EXPLICIT
SET @doc = '
<root>
<Product ProductID="1" CategoryID="1" />
<Product ProductID="2" CategoryID="1">
  <Order OrderID="10813" OrderDate="1998-01-05T00:00:00" />
</Product>
<Product ProductID="24" CategoryID="1" />
<Product ProductID="34" CategoryID="1" />
<Product ProductID="35" CategoryID="1" />
<Product ProductID="38" CategoryID="1">
  <Order OrderID="10816" OrderDate="1998-01-06T00:00:00" />
  <Order OrderID="10817" OrderDate="1998-01-06T00:00:00" />
</Product>
<Product ProductID="39" CategoryID="1" />
<Product ProductID="43" CategoryID="1">
  <Order OrderID="10814" OrderDate="1998-01-05T00:00:00" />
  <Order OrderID="10819" OrderDate="1998-01-07T00:00:00" />
</Product>
<Product ProductID="67" CategoryID="1" />
<Product ProductID="70" CategoryID="1">
  <Order OrderID="10810" OrderDate="1998-01-01T00:00:00" />
</Product>
</root>
'

-- Create an internal representation of the XML document.
EXEC sp_xml_preparedocument @idoc OUTPUT, @doc

-- Execute a SELECT statement using OPENXML rowset provider.
SELECT *
FROM OPENXML (@idoc, '/root/Product/Order', 1)
    WITH (ProductID int ' ../@ProductID',
          Category int ' ../@CategoryID',
          OrderID int '@OrderID',
          OrderDate varchar(19) '@OrderDate',
          Previous varchar(10) '@mp:prev')
EXEC sp_xml_removedocument @idoc
```

Really pay attention to the relative pathing in our XPath references when we build the metadata. Compare it to the hierarchy in the XML. If you think about it, it works an awful lot like a DOS/Windows directory structure.

So let’s take a look at what our hard work has wrought:

ProductID	Category	OrderID	OrderDate	Previous
2	1	10813	1998-01-05T00:00:00	NULL
38	1	10816	1998-01-06T00:00:00	NULL
38	1	10817	1998-01-06T00:00:00	23
43	1	10814	1998-01-05T00:00:00	NULL
43	1	10819	1998-01-07T00:00:00	35
70	1	10810	1998-01-01T00:00:00	NULL

As you can see, we were able to grab XML data from different points in the hierarchy. We can use XPath to navigate us to any point in the XML tree that we need to retrieve our data from.

So, now that we have a way to get XML query results, and a way to get XML turned into relational data, it’s time to start looking at even more cool stuff — ways to interact with SQL Server’s XML features.

A Brief Word on XSLT

Well now, this takes us to the last, but far and away the most complex of the things we’re dealing with in this chapter.

The Extensible Stylesheet Language Transformations side of things is something of a little extra toss into the XML world that increases the power of XML multifold. You see, using XSLT, we can transform our XML document into other forms.

To get us going on a quick start here, let’s take a look at an XML document that I’ve produced from the Northwind database.

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
    <Products ProductID="28" ProductName="Rössle Sauerkraut"/>
    <Products ProductID="39" ProductName="Chartreuse verte"/>
    <Products ProductID="46" ProductName="Spegesild"/>
  </Customer>
  <Customer CustomerID="BLONP" CompanyName="Blondesddsl père et fils">
    <Products ProductID="28" ProductName="Rössle Sauerkraut"/>
    <Products ProductID="29" ProductName="Thüringer Rostbratwurst"/>
    <Products ProductID="31" ProductName="Gorgonzola Telino"/>
    <Products ProductID="38" ProductName="Côte de Blaye"/>
    <Products ProductID="39" ProductName="Chartreuse verte"/>
    <Products ProductID="41" ProductName="Jack&apos;s New England Clam
Chowder"/>
    <Products ProductID="46" ProductName="Spegesild"/>
    <Products ProductID="49" ProductName="Maxilaku"/>
  </Customer>
</root>
```

What we have here is something XML does very well — keep hierarchies. In this case, we have a situation where customers have ordered different products. What our XML document is laid out to tell us is what products our customers have purchased. It seems like a reasonable question — doesn't it?

Now, time for me to twist things on you a bit — what if I change the question to be more along the lines of “Which customers have ordered each product?” Now our perspective has changed mightily. At this point, a much better hierarchy would be one that had the products on the outside and the customers on the inside. Under that scenario, it would be our customers (rather than our products) that were repeated multiple times (once for each product), but it would get more to the root of our question.

With XML coupled with XSL transformations, this is no big deal. You see, I don't want to change the data that I'm looking at all — I just need to *transform* my XML document so that I can *look* at it differently.

Don't confuse my saying “The way I look at the data” to mean how I visually look at it — what I'm talking about is more of how the data is perceived. Part of that is just how ready the data is to be used in a particular fashion. With our customers at the top of the hierarchy, our data doesn't seem very ready for use answer questions that are product focused. What I need is to change the data to be product focused — just like my questions.

So let's look back at the same exact data, but transformed into another look:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <Products ProductID="28" ProductName="Rössle Sauerkraut">
    <Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste"/>
    <Customer CustomerID="BLONP" CompanyName="Blondesddsl père et fils"/>
  </Products>
  <Products ProductID="29" ProductName="Thüringer Rostbratwurst">
    <Customer CustomerID="BLONP" CompanyName="Blondesddsl père et fils"/>
  </Products>
  <Products ProductID="31" ProductName="Gorgonzola Telino">
    <Customer CustomerID="BLONP" CompanyName="Blondesddsl père et fils"/>
  </Products>
  <Products ProductID="38" ProductName="Côte de Blaye">
    <Customer CustomerID="BLONP" CompanyName="Blondesddsl père et fils"/>
  </Products>
  <Products ProductID="39" ProductName="Chartreuse verte">
    <Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste"/>
    <Customer CustomerID="BLONP" CompanyName="Blondesddsl père et fils"/>
  </Products>
  <Products ProductID="41" ProductName="Jack's New England Clam Chowder">
    <Customer CustomerID="BLONP" CompanyName="Blondesddsl père et fils"/>
  </Products>
  <Products ProductID="46" ProductName="Spegesild">
    <Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste"/>
    <Customer CustomerID="BLONP" CompanyName="Blondesddsl père et fils"/>
  </Products>
  <Products ProductID="49" ProductName="Maxilaku">
    <Customer CustomerID="BLONP" CompanyName="Blondesddsl père et fils"/>
  </Products>
</root>
```

Again, this is the same data — just a different perspective.

But hang on a sec — why limit ourselves to just XML? While going over every type of transformation you could do is well outside the scope of this book, it is important to understand that XSL transforms are possible to and from a wide variety of formats. You can transform to other XML layouts, but you can also transform to other formats entirely such as CSV files or even a Word document.

Summary

Well, there you have an extreme whirlwind tour of XML, DTDs, schemas, and a bit of XPath to boot. It is impossible in one chapter to address a topic as broad as XML and its related technologies; however, I hope this chapter has at least provided insight into what XML is all about and what is involved in making use of it at the most basic level.

XML is one of the most important technologies to hit the industry in the last 10 or more years. It provides a flexible, very transportable way of describing data. This ease of description will help to facilitate not only the Website development that you hear so much about, but also the kind of behind-the-scenes information exchange that businesses have longed for a very long time indeed.

17

Reporting for Duty, Sir!: A Look At Reporting Services

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 seems incredibly simple, but turns 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 few 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 of 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 such as *Professional SQL Server 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 email.

Building Simple Report Models

To create a simple model, start by opening the Business Intelligence Studio.

Note that this is entirely different from the SQL Server Management Studio that we've been working with thus far. The 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 the Business Intelligence Studio in this chapter, we will also visit it some to work with Integration Services in Chapter 18.

Choose File⇨New Project, and you should bring up the dialog shown in Figure 17-1.

Choose the name of your choice (I've gone with the rather descript 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.

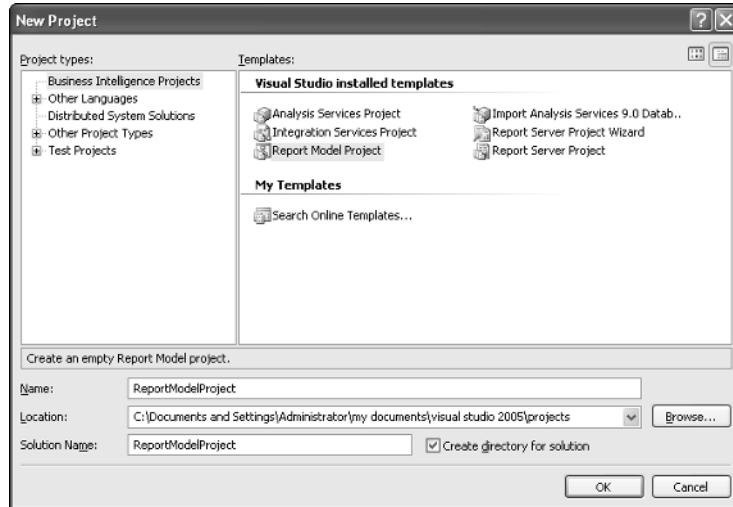


Figure 17-1

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 show in Figure 17-2.

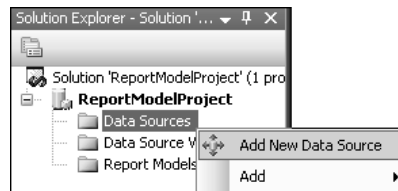


Figure 17-2

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 data source selection page of the Data Source Wizard. There is, however, one problem—we don't have any data sources as yet (as we see in Figure 17-3).

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 the 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).



Figure 17-3

This brings us up to the Connection Manager shown in Figure 17-4.

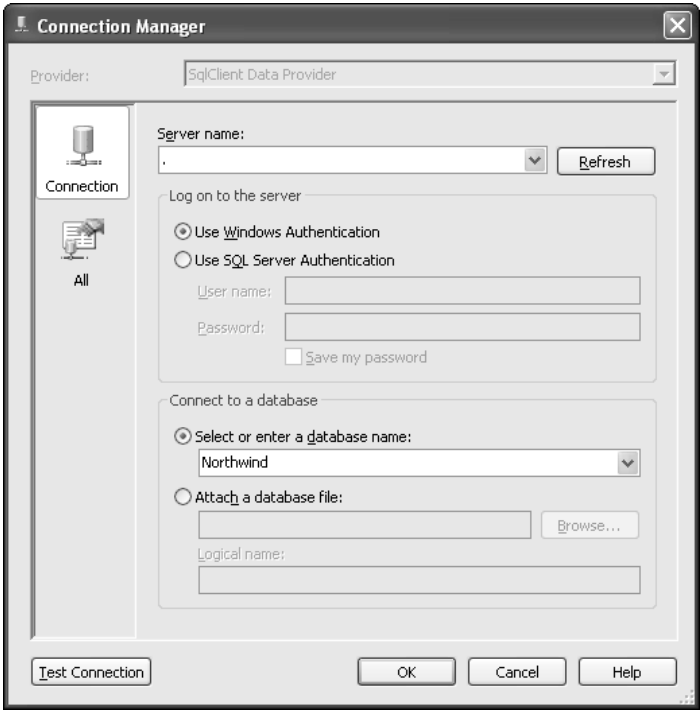


Figure 17-4

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 to 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 Northwind database.

Go ahead and click OK, and we see a different Data Source Wizard dialog than we did the first time (see Figure 17-5).

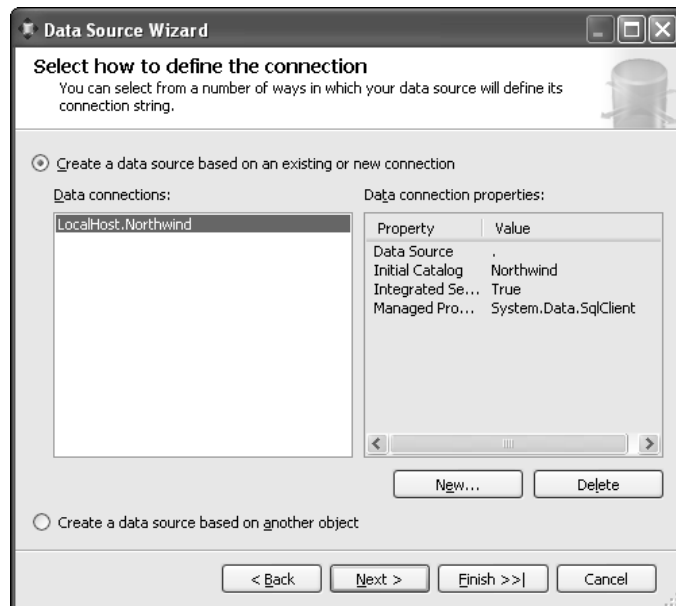


Figure 17-5

Our example only had the connection we just created, but we could, if desired, actually create several connections, and then choose between them. Note also the summary of the data connection properties on the right-hand side of the dialog.

Click Next, and it brings you to the final dialog of the wizard, as shown in Figure 17-6.

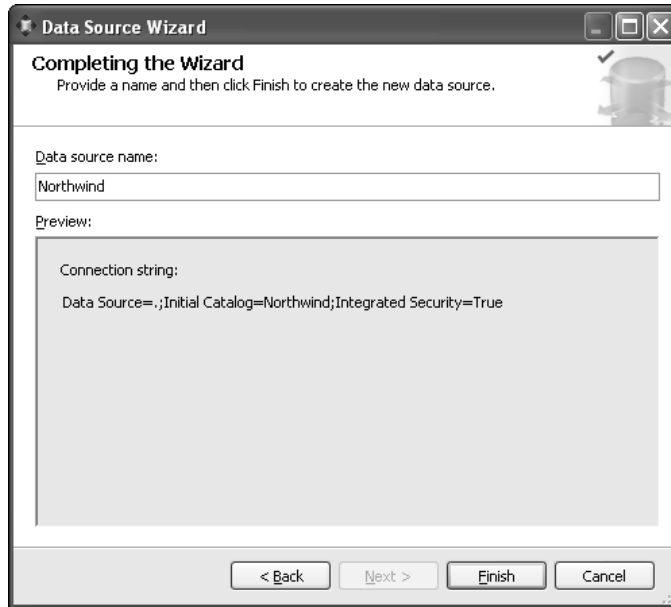


Figure 17-6

Note that the default name happens to be the database we chose, but this is truly just a name for the data source—we can call this data source whatever we want and it would still be connecting to the Northwind 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 data source for our project, as shown in Figure 17-7.

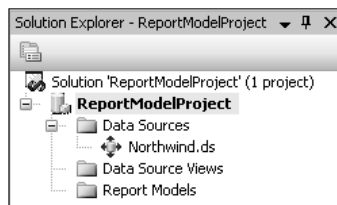


Figure 17-7

If you ponder Figure 17-7 for a moment, you'll notice that there is a bit more to our Report Model project than just data sources — 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 10. 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 data source, then they are able to see all data available to that data source. Data Source Views allow us to filter the data source 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 17-8.

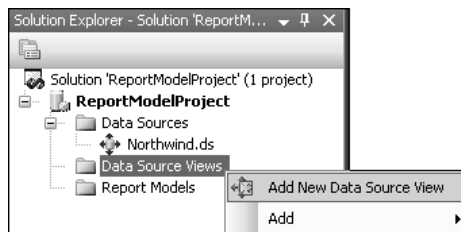


Figure 17-8

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 (only one or two very subtle cosmetic differences to one we had partway through creating our data source — Figure 17-5.

Accept the default, and then click through to the Select Tables and Views dialog shown in Figure 17-9. 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 Order Details and then use the > button to move that into the Selected Objects column. Next, click the Add Related Tables button and see what happens — SQL Server will use the rules that it knows about the relationships between the tables (based on defined Foreign Keys) to figure out what tables are related to the one or more tables we had already selected. In this case, it chose Orders (the parent of our Order Details table) and Products (which Order Details also has a foreign key to). Add the Customers table to finish catching up with the tables I had selected in Figure 17-9.

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 the Completing the Wizard dialog shown in Figure 17-10. This dialog is pretty much just synopsisizing 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!”

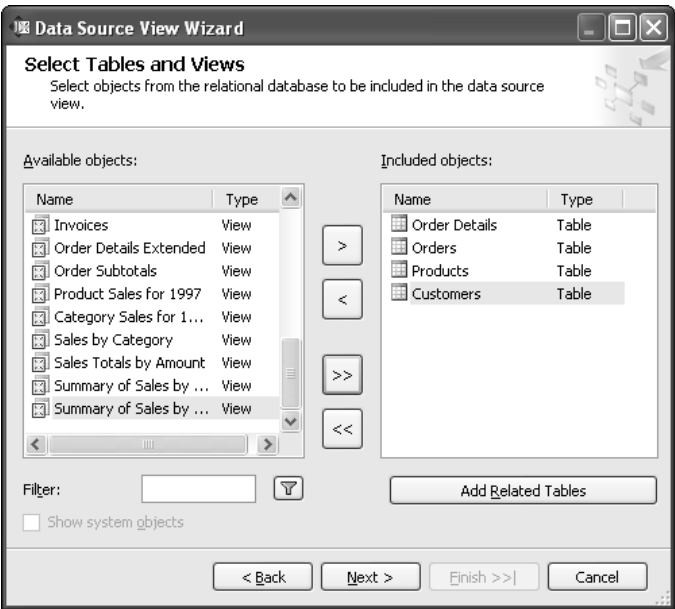


Figure 17-9

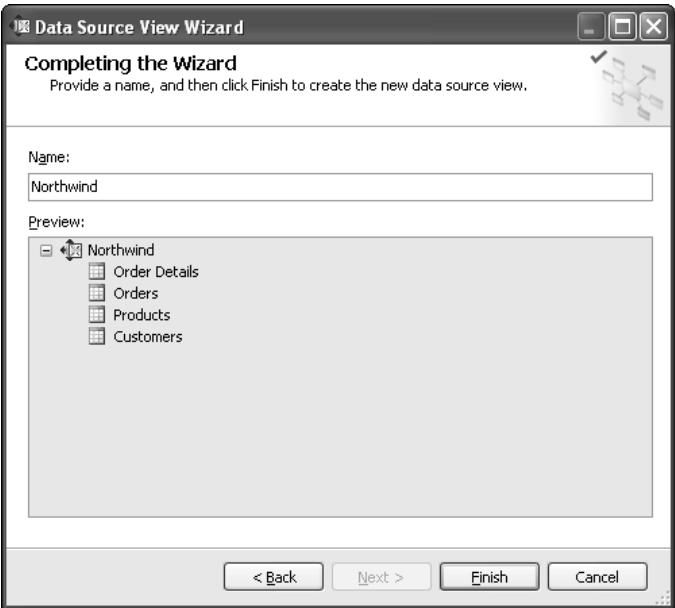


Figure 17-10

Now click Finish and the data source view is actually constructed and added to our Report Model Project.

At this point, we're ready to create the actual Report Model. As we did with data source and Data Source View, simply right-click the Report Model node of the Solution Explorer tree and select Add New, as shown in Figure 17-11.

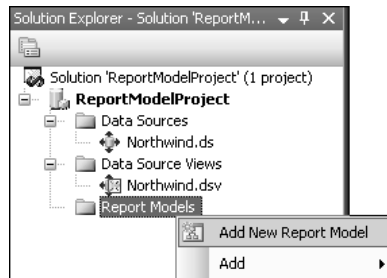


Figure 17-11

At the start of defining a Report Model, we get yet another one of those “welcome” dialogs. Click through that and you’ll be at the relatively meaty Select Data Source View dialog shown in Figure 17-12. Not surprisingly, the only one that shows is the one just created (still called Northwind).

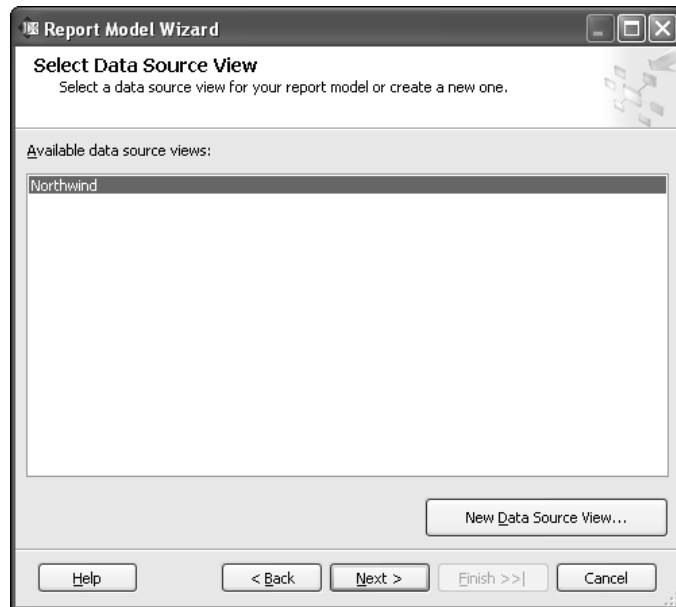


Figure 17-12

Go ahead and select our one Data Source View and click Next to move on to the next dialog (shown in Figure 17-13), the report model generation rules.

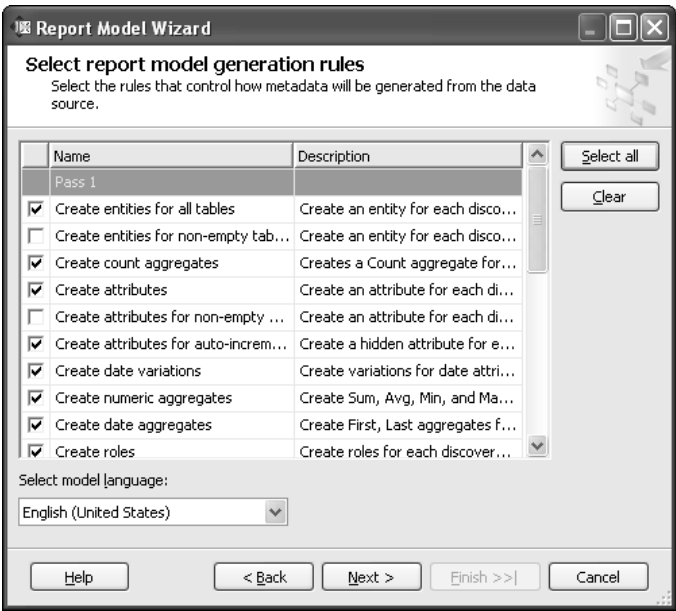


Figure 17-13

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 9 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 makeup of your data.

Whether you rely on existing statistics vs. going ahead and updating your statistics is a somewhat advanced concept. As such, 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 subtly 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 Northwind Report Model—original, eh?) and click Run, and your report model will be generated as shown in Figure 17-14.

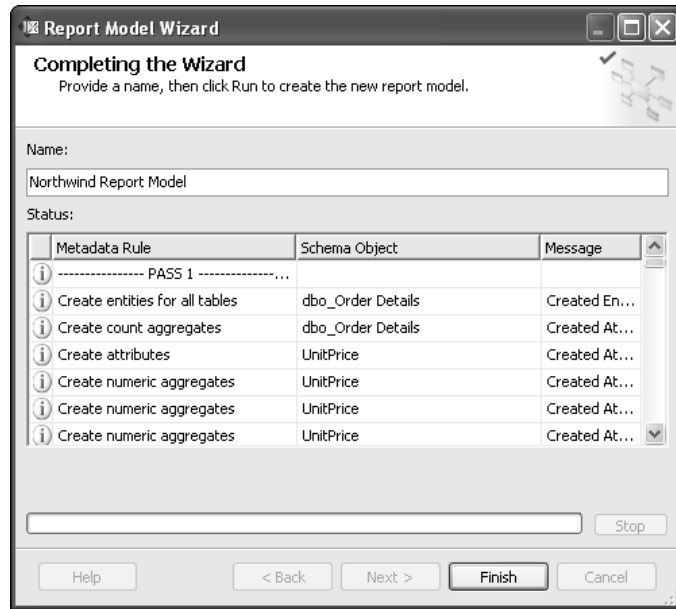


Figure 17-14

Click Finish, and we can begin to see the results of what we just created, which should look something like Figure 17-15.

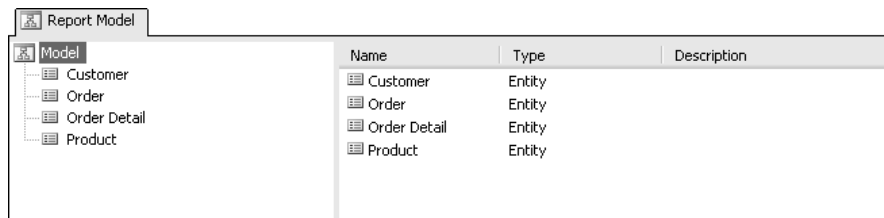


Figure 17-15

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 datatype.

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 Order table. Notice how the Order column is grayed out. If you click on it and check the property window shown in Figure 17-16, you should notice that this rather important field has been defined as being hidden.



Figure 17-16

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 OrderID 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 this in the report.

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 in a white space area of the solution explorer (or on our actual report) and choose Deploy. You can watch the progress of the deploy in the output window of Visual Studio.

```
----- Deploy started: Project: ReportModelProject, Configuration: Production -----  
-  
Deploying to http://localhost/ReportServer?%2f  
Deploying data source '/Data Sources/Northwind'.  
Deploying model 'Northwind 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 the Business Intelligence Studio and actually visit the reporting user interface — which is basically just a Website. 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 17-17.

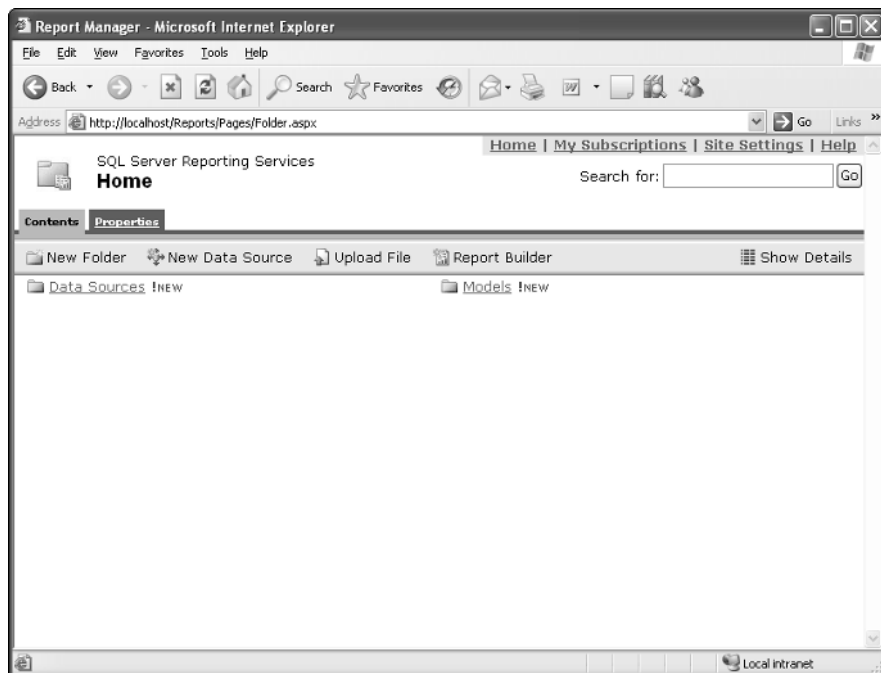


Figure 17-17

Chapter 17

This is essentially the home page for user reports and such. Notice right away the “New!” icons next to our data sources and models (a strong hint that our deploy actually worked — amazing!). Go ahead and click on Report Builder.

The Report Builder relies on a small applet that 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 the Report Builder.

The 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 between 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 each table and dragging the relevant columns into the table:

Orders.OrderID

Customers.CompanyName

Products.ProductName

Order Details.Quantity

Orders.OrderDate

Orders.RequiredDate

Also go ahead and click in the title area and add the “Outstanding Orders” title. When you’re done, you should wind up with something that looks like Figure 17-18.

Notice also how the fields you have used are bolded in the Report Data Explorer windows.

When you have this done, click Run Report, and you’ll see the first fruits of our labor, as shown in Figure 17-19.

Order ID	Company Name	Product Name	Quantity	Order Date	Required Date
0	XXXXXXXXXXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXXXXXXXXXX	0	1/1/2005	1/1/2005

Total Order Details: 0

All Order Details

Figure 17-18

Ord	Company Name	Product Name	Quantity	Order Dat	Required
11076	Bon app'	Grandma's Boysenberry Spread	20	5/6/1998	6/3/1998
		Teatime Chocolate Biscuits	10	5/6/1998	6/3/1998
		Tofu	20	5/6/1998	6/3/1998
11077	Rattlesnake Canyon Grocery	Aniseed Syrup	4	5/6/1998	6/3/1998
		Camembert Pierrot	2	5/6/1998	6/3/1998
		Chang	24	5/6/1998	6/3/1998
		Chartreuse verte	2	5/6/1998	6/3/1998
		Chef Anton's Cajun Seasoning	1	5/6/1998	6/3/1998

Figure 17-19

We, do, however, have a problem — the report has 49 pages. Unless Northwind has suddenly gotten a lot bigger than it has been throughout the book thus far, we must be including way too much data. Indeed, this is showing all orders — not just those that are outstanding. The 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 17-20.

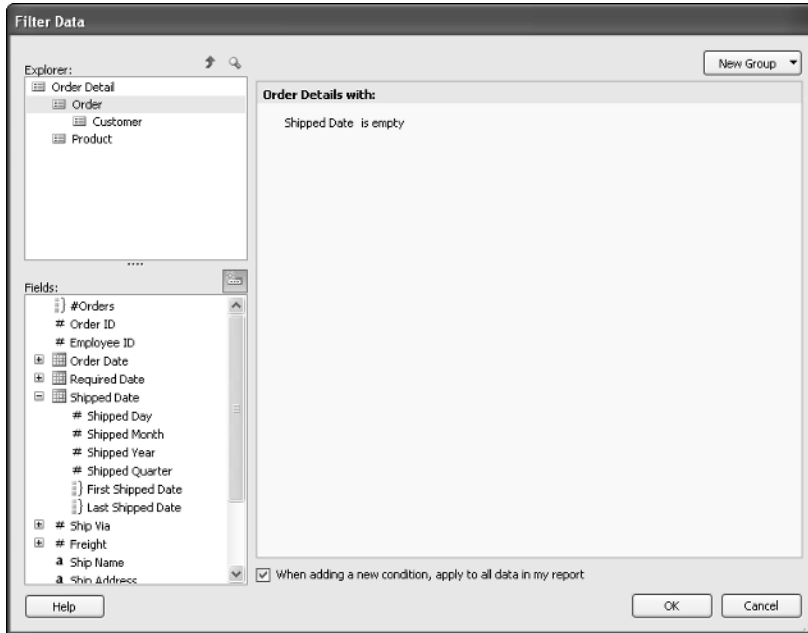


Figure 17-20

To create this, I started by dragging the Shipped date into the “filter with” area. I think I clicked the default comparison of “equals” and changed it to the “Is Empty” option (which is equivalent to an `IS NULL` in standard SQL). Click OK, rerun the report, and you should be down to just 3 pages or so (at total of 73 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).

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 data source, and access to the data source is secured such that you can control which users see which data sources), 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.

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 will be 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, the Business Intelligence Development Studio will allow you to create Report Server Projects.

As I mentioned before, there will likely be entire books around just this subject, so the approach we're going to take here is 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 the Business Intelligence Development Studio.
2. Create a new data source against our Northwind 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).
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 data source you created in Step 2, and click Next.

This should bring you to the query dialog shown in Figure 17-21.

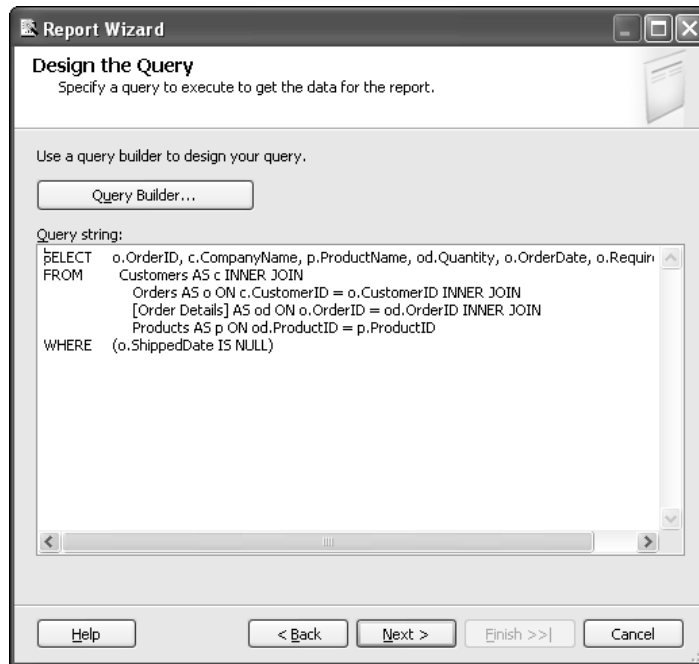


Figure 17-21

Chapter 17

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 unshipped 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 17-22.

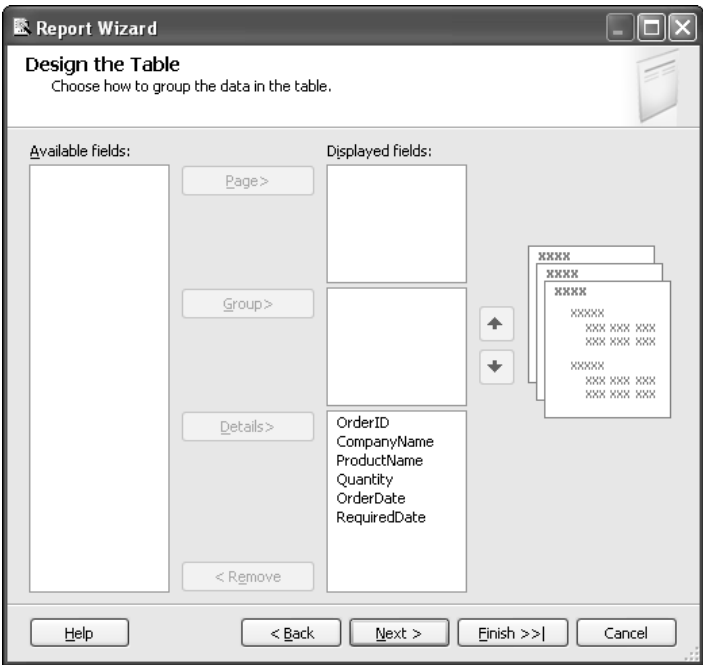


Figure 17-22

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 Company 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.

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 17-23 (yours may look a tad different if you chose a different style than I did).

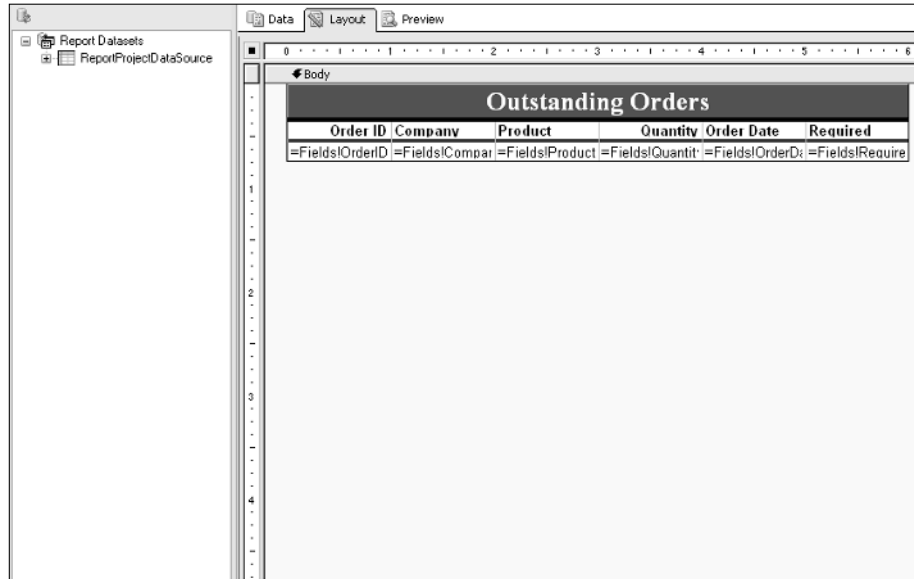


Figure 17-23

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.

Try It Out Altering Report Projects

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 first date field (Order Date) and select Properties.
2. Click on the fx 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 17-24.
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.

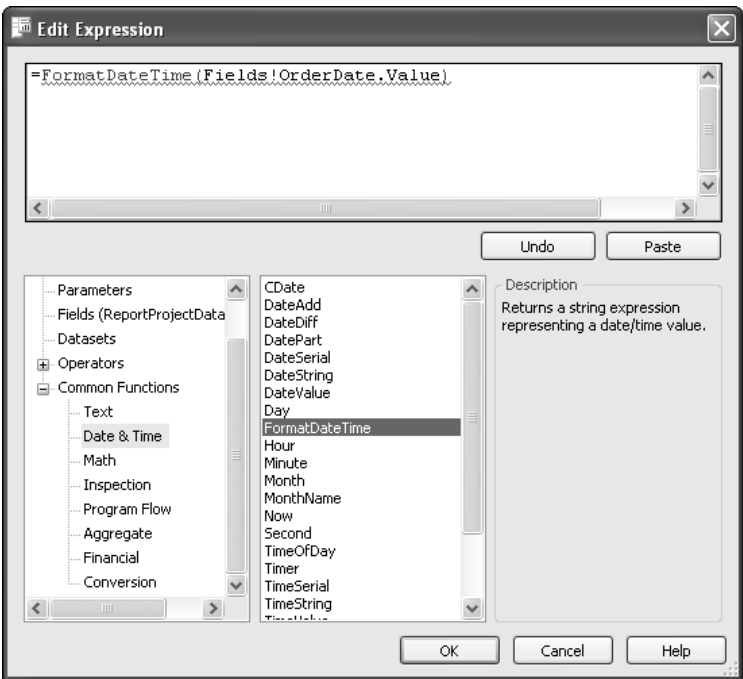


Figure 17-24

- 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 17-25.

Outstanding Orders						
Order ID	Company Name	Product Name	Quantity	Order Date	Required Date	
11008	Ernst Handel	Rössle Sauerkraut	70	4/8/1998	5/6/1998	
11008	Ernst Handel	Sasquatch Ale	90	4/8/1998	5/6/1998	
11008	Ernst Handel	Flotemysost	21	4/8/1998	5/6/1998	
11019	Rancho grande	Spegesild	3	4/13/1998	5/11/1998	
11019	Rancho grande	Maxilaku	2	4/13/1998	5/11/1998	
11039	LINO-Delicatesses	Rössle Sauerkraut	20	4/21/1998	5/19/1998	
11039	LINO-Delicatesses	Stocking St...	34	4/21/1998	5/19/1998	

Figure 17-25

Note that I’ve played around with column sizes a bit (try it!) to use the page efficiently.

How It Works

Under the covers, there is what is called Report Definition Language — or RDL. RDL is actually an XML implementation. 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 the 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="OrderDate">
      <rd:DefaultName>OrderDate</rd:DefaultName>
      <ZIndex>1</ZIndex>
      <Style>
        <PaddingLeft>2pt</PaddingLeft>
        <PaddingTop>2pt</PaddingTop>
        <PaddingBottom>2pt</PaddingBottom>
        <Direction>RTL</Direction>
        <PaddingRight>2pt</PaddingRight>
      </Style>
      <CanGrow>true</CanGrow>
      <Value>=FormatDateTime(Fields!OrderDate.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 the 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 17-26.

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.

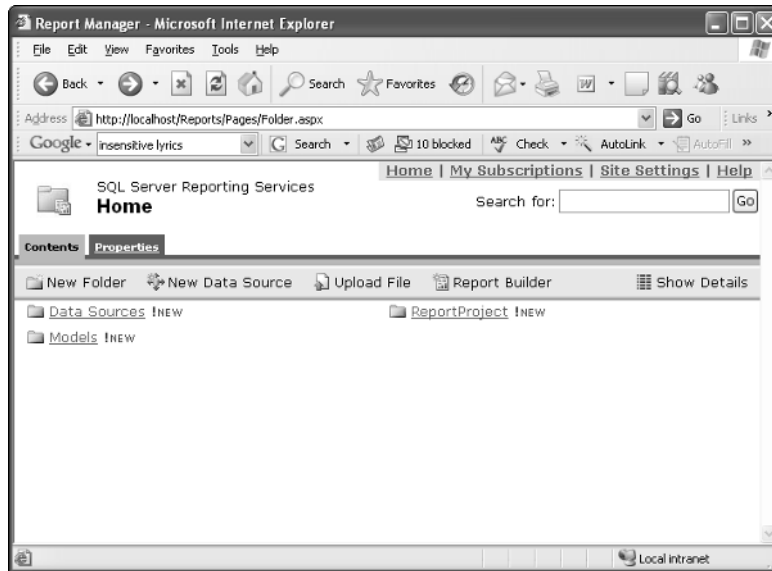


Figure 17-26

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

Getting Integrated With Integration Services

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 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, 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 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!).

Using the Import/Export Wizard to Generate Basic Packages

An SSIS package is essentially the SSIS equivalent to a program. It bundles up a set of instructions (potentially including limited conditional branch logic) such that they can be moved around, cloned, edited, and so on. The Import/Export Wizard is a tool to automate building such a package for a relatively simple Import or Export.

To use the Import/Export Wizard, you need to start the SQL Server Business Intelligence Suite tool from the programs menu on your system, and start a new Integration Services project.

To be honest, I consider the move of this to the Intelligence Suite 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 OLTP database and transforming it for using in a 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 the Business Intelligence Suite—not in the Management Studio that you’ve become so accustomed to using so far.

On the far right (assuming you haven’t moved any of the dockable windows around), you should find a node in the solution explorer called “SSIS Packages.” Right-click this, and choose the SSIS Import and Export Wizard.

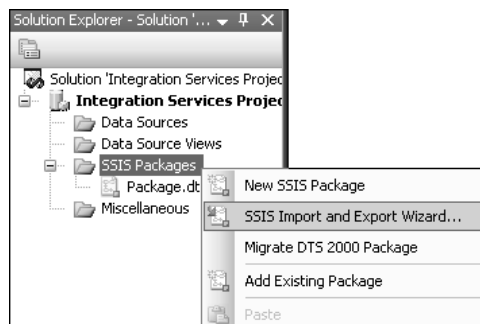


Figure 18-1

This brings up an introduction dialog that is pretty useless, but click Next and we move on to a far more fundamental dialog to help us set up a connection to a Data Source (see Figure 18-2).

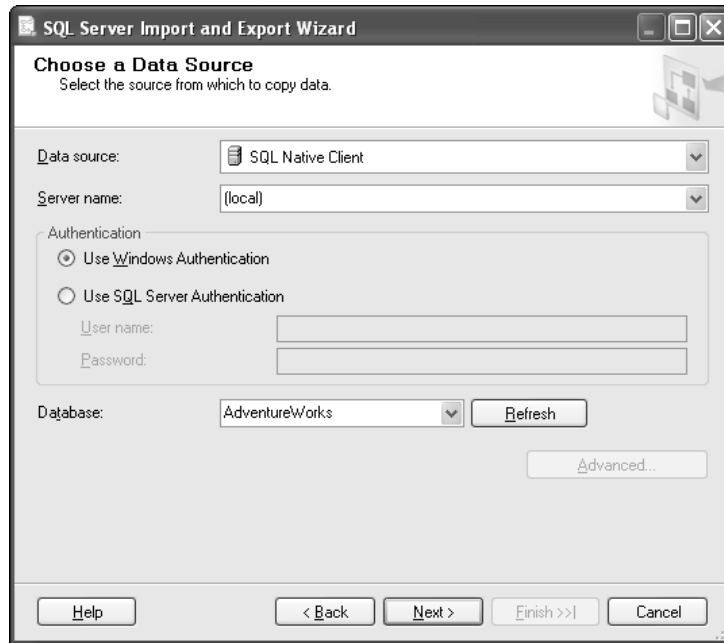


Figure 18-2

Microsoft makes the mistake of using the same term — `dataSource` — in two different ways in this dialog. The first is the overarching concept of a source of data — complete with all of the information that goes into making a connection string (server, authentication, database and the like). The second way is a subelement of the first — the data source as an OLEDB driver type.

For now, leave the actual data source drop-down box as SQL Native Client. From there, set up your authentication information (I've used Windows Authentication in the preceding example) and choose AdventureWorks as your Database.

You're now ready to click Next, where you'll see essentially the same dialog to start. This time, however, we're going to make a bit of a change in things. Change the Destination field to be Flat File Destination, as shown in Figure 18-3.

Notice that the rest of the options change to something more suitable to a file system-based destination instead of a table in our SQL Server. Click Browse, and a standard file dialog box comes up. Let's name our file "TextExport.csv" and click OK. At this point, several other options become available for us to edit, as shown in Figure 18-4.

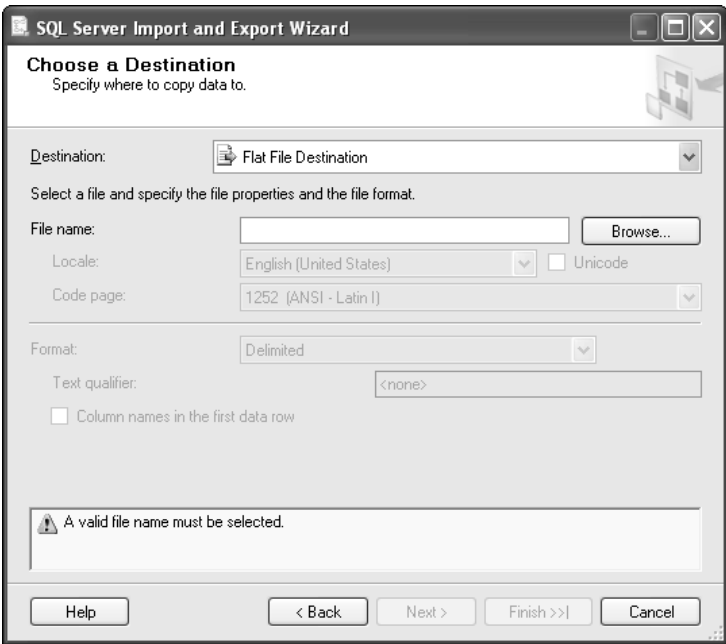


Figure 18-3

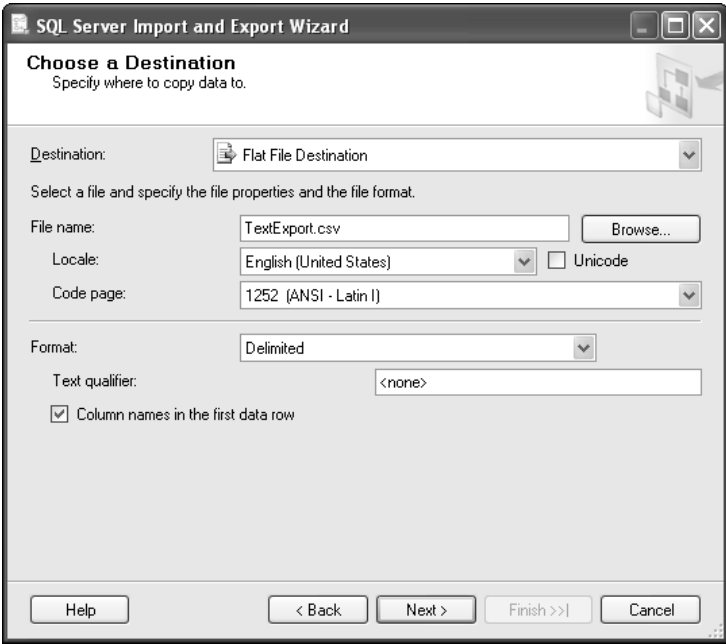


Figure 18-4

We're not going to go into real detail on this, because what options there are will vary widely by the particular OLEDB data source you've selected. Indeed, the real point of the moment is just that—SSIS will alter the dialog to give you option choices that are contextual to the particular OLEDB datasource you're using. In the preceding example, I've simply clicked the `Column names in first data row` option, and it's time to click `Next` again. This takes us to choices on how to select what data we want out of our source database. We have two choices here:

- ☐ **Copy:** This has us do a straight copy out of a table or view.
- ☐ **Query:** This allows us to select out of pretty much any operation that's going to yield a result set.

Let's start by taking a look at the `Query` option. If you choose this option and click `Next`, you get yet another dialog—this one pretty boring in the sense that all it does is give you a box where you can enter in the text of your query (see Figure 18-5).

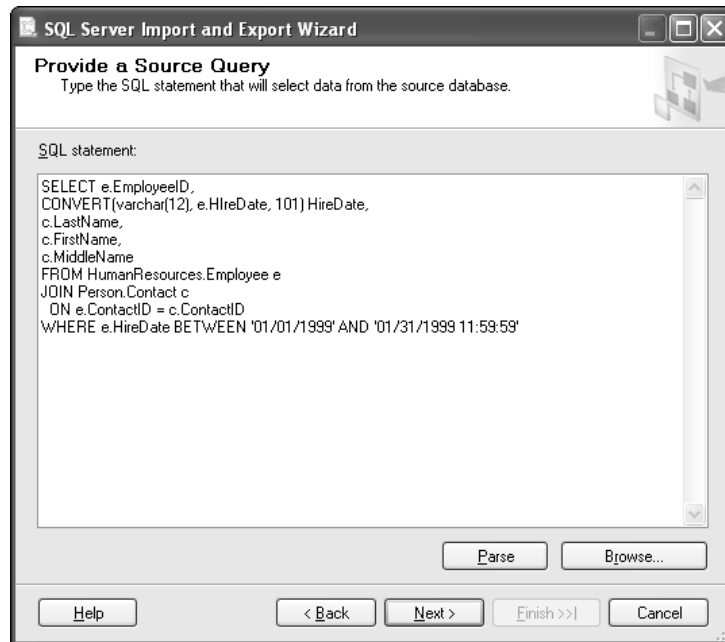


Figure 18-5

In my example here, I've shown where we could be setting up to export a list of employees that we hired in our first month of business. If you want to test this out, you can type out the query, and then try testing it with the `Parse` button to make sure that the query you want to run is syntactically valid. Then click `Next` to get to the dialog box shown in Figure 18-6.

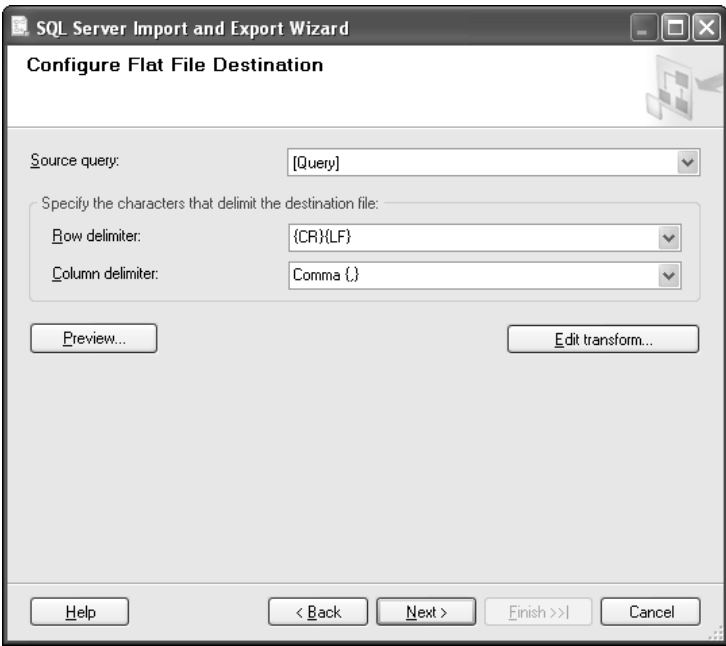


Figure 18-6

For starters, this sets us up to decide what we want our flat file to look like. You can click preview to see whether your query looks like it’s going to return the data that you expect (see Figure 18-7).

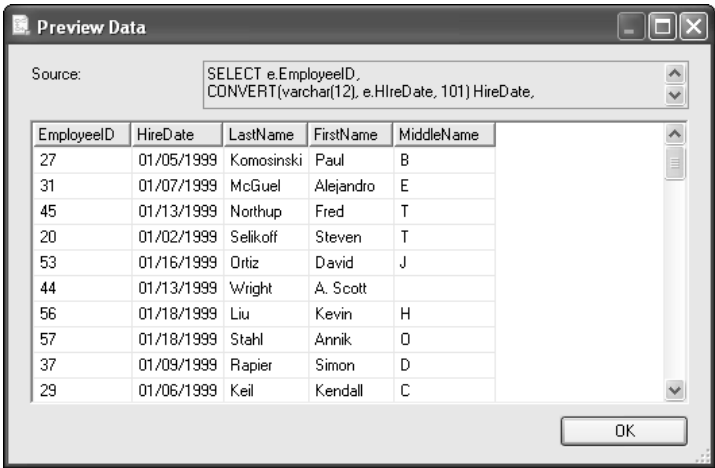


Figure 18-7

Click OK, and you’re back to the Configure Flat File dialog.

Next, try clicking on the Edit Transform button which takes you to the dialog box shown in Figure 18-8.

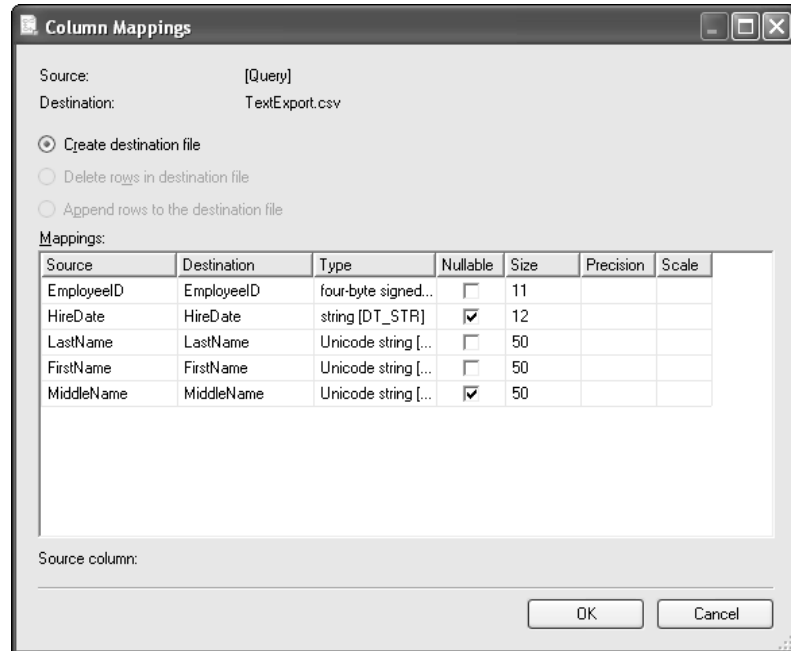


Figure 18-8

At first, things will probably appear pretty mundane here, but, actually, things are getting a lot more interesting.

Try clicking on any of the Destination column values. You'll see a drop-down option appear, and that will include the option to ignore that column — essentially omitting a source column from the destination when you output the file.

This should bring up the question of “Why would I have even put the column in the original query if I was just going to ignore it at output?” The reasons are multifold. First, you may have wanted the column there primarily for preview purposes to verify that the data is what you expected — for example, including the names to verify that it seems like the right people when you only intend to output the EmployeeID. Also, you may be using a query that is copied from some other source and want to avoid risking editing it (for example, if it is particularly complex).

One other point there — this dialog is shared with the direct table choice (back when we chose Query, we had the option of a table — remember?), so the option is even more applicable there — where there may be many columns in the table that you don't want in the end output.

Next, try clicking on the Type column values, and you'll see a number of choices.

I highly recommend resizing the columns in this dialog so you can see what the Type column is really trying to show you. Just hover your mouse over the right side of the column header much as you would if you were using Excel, and then click and drag the column divider to the right to enlarge the column.

Most of the time you'll stick with whatever the default conversion was, based on the source datatype, but sometimes you may want to change the output type in some way — for example, treating integer data as being a string to deal with some expectation of the end destination of your data (not all systems treat what is seemingly the same data the same way).

Finally, you can mess with the nullability and scale/precision of your data. It's pretty rare that you would want to do this, and it's something of an advanced concept, but suffice to say that if you were to use more advanced transformations (instead of the default Import/Export Wizard), then this might be an interesting option to facilitate error trapping of data that isn't going to successfully go into your destination system. In most cases, however, you would want to do that with `WHERE` clauses in your original query.

Now, let's click Cancel to go back to our Configure Flat File dialog, and then click Next to move onto the next dialog:

At this point, you'll get a simple confirmation dialog. It will synopsise all the things that you've asked the wizard to do — in this case:

- ☐ Copy rows from [Query] to C:\Documents and Settings\xxx\My Documents\TestExport.txt.
- ☐ The new target table will be created.
- ☐ The package will be saved to the package file "C:\Documents and Settings\xxxx\My Documents\Visual Studio 2005\Projects\Integration Services Project1\Integration Services Project1\Package1.dtsx".
- ☐ The package will not be run immediately.

Most of this is self-explanatory, but I want to stress a couple of things.

First, it will create the file for you with the name specified; if, when we designed the package, the file had been detected as already existing, then we would have been given some options on whether to overwrite the existing file or just append our data onto the end of it.

Perhaps more importantly, notice that a *package* is being created for you. This package is what amounts to an SSIS program. You can have SSIS execute that package over and over again (including scheduling it), and it will perform the defined export for you each time.

Finally, note that it says that your package will *not* be run immediately — we need to either manually execute this package or schedule it to run.

Click Next, and your package is created.

Let me stress again that this is just something of a preparation step — your package has not run, and your data is not yet exported. You still need to execute or schedule the package to get your actual exported file. That side of things — the execution of the package — is the same for all SSIS packages though, and isn't Import/Export Wizard-specific, so we'll hold that side of things until the next section, and move on to a very quick discussion of turning things around into an import.

Executing Packages

There are a few different ways to execute an SSIS package:

- ❑ **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:** We'll talk more about the SQL Server Agent in our next chapter, 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 to run it in, 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 original export package, and you should get the execute dialog shown in Figure 18-9.

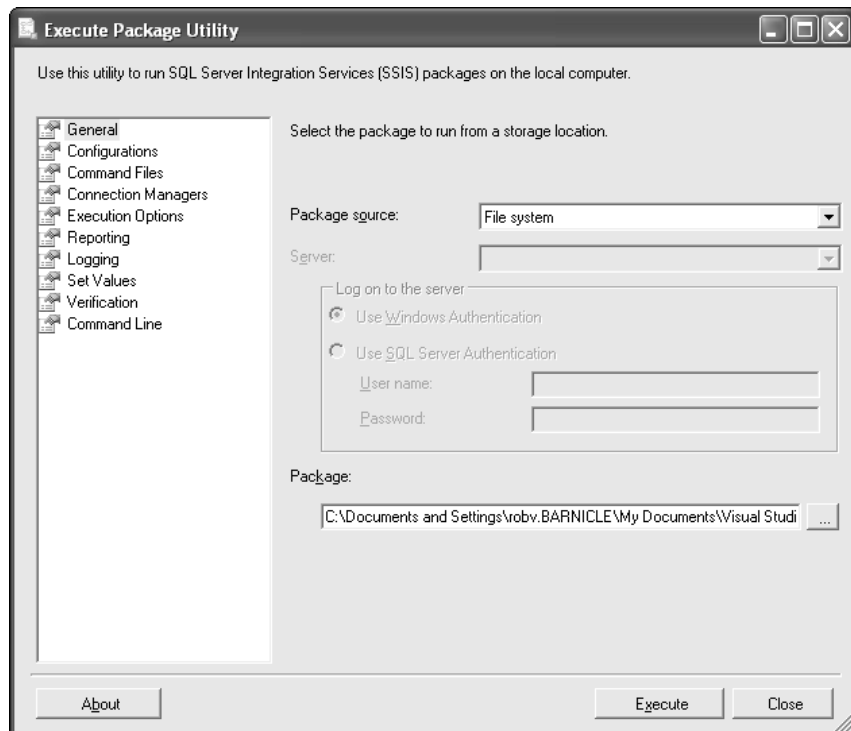


Figure 18-9

Chapter 18

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 we did on our 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.

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 of 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 file name (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 the Business Intelligence Development Studio

You can edit the package within the development studio also. Either execute your entire solution (running it with the `execute` button as you would any Visual Studio solution), or bring the package up in the edition by double clicking within the solution explorer (you can then right-click any particular item you want to execute).

Executing Within the Management Studio

While the Management Studio doesn’t give you a package editor, it does give you the ability to run your packages.

Chapter 18

In the Registered Servers pane of the 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 the Management Studio). This should create a connection to the Integration Services on that server, and add an Integration Services node in your Object Explorer.

In order to execute a package in this fashion (using the 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.)

Editing the Package

This is going to take us onto something of the home stretch of our journey through SSIS, and I feel the need to preface it a bit by saying that we will have only scratched the surface of what *can* be done. So, as we make this last part of our journey, *consider the possibilities*.

What we're going to learn about here is editing a package. The same basic notion applies to creating new packages also, but, being as we're all beginners here, we're going to show off the features on a package that already has a few features. The goal here isn't so much to make an SSIS expert out of you, but to give you a concept of how you can construct relatively complex packages.

If you don't still have it up, fire up the Business Intelligence Development Studio. Go to File⇨New Project, and start a new Integration Services Project (see Figure 18-10).

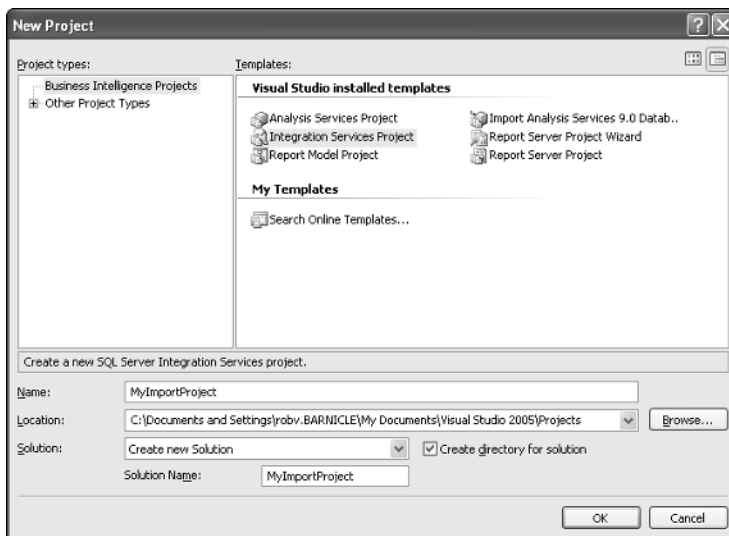


Figure 18-10

I've called mine MyImportProject.

After you click OK, it will create the solution workspace — when it's done, notice that it's already created an empty package for you named "Package.dtsx". This isn't the package we're interested in, so right-click and select Delete to remove it. Now right-click the SSIS Packages node and select Add Existing Package. This brings up a dialog to ask us where we want to get the package from — remember that there are multiple places you can store your SSIS packages. We happened to have chosen the file system, so we'll fill out our dialog accordingly (see Figure 18-11).

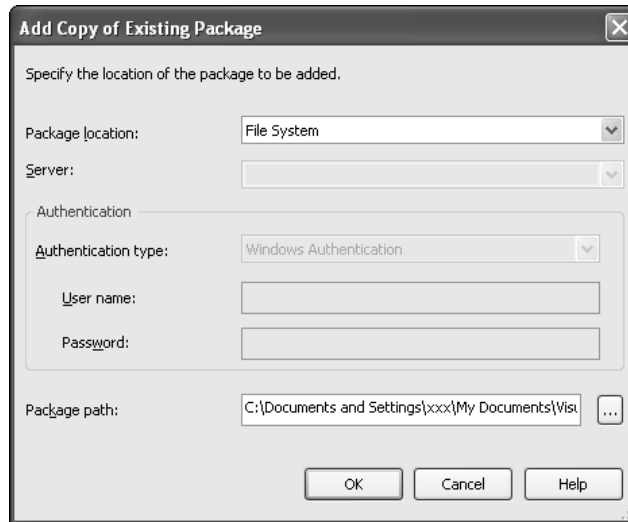


Figure 18-11

Notice that I selected the File System as being our source, and then filled in a file location at the bottom. You'll need to put in the appropriate path to wherever you created your export project.

Your package should have been added to the SSIS Packages node of your solution; right click the name of your package and select Open. This brings up a relatively simple package for editing, but it has only one task in it, highlighted in Figure 18-12.



Figure 18-12

This is yet another container for the work our package is doing. In this case, our export really has only one task, but double-click the task to open it up, and let's take a look at some of the pieces that go into an SSIS package (see Figure 18-13).

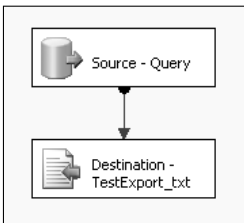


Figure 18-13

Here are the two steps of our export task. We run a query to get source data, and we pump it into the text file. Notice the arrow and the way it indicates the direction of dataflow. Now, double-click the source query node, and you'll see our original query (see Figure 18-14).

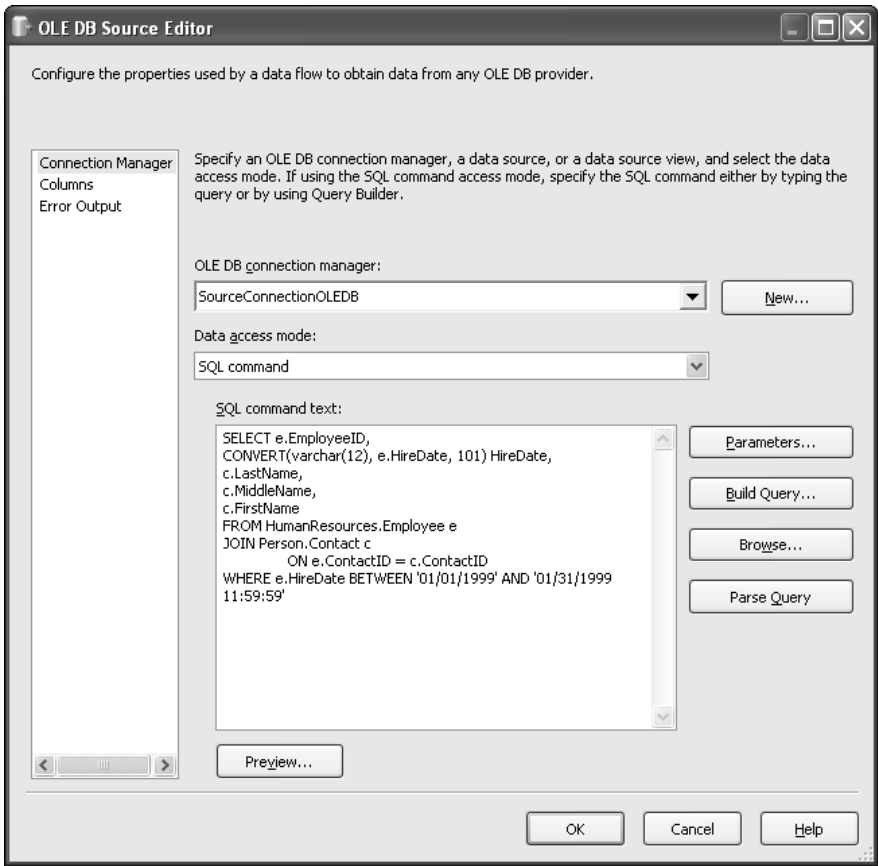


Figure 18-14

All the wizard did for us is hide things such that we didn't have to do this piece by piece—we would have built our own package right here in the designer with all the same settings and objects. It hid (for better or worse), much of the complexity of building a package.

We can build these packages from scratch. We can also edit wizard-driven packages just like this one.

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.

19

Playing Administrator

And so, here we are. The relative “end of the road,” and yet, it’s really just the beginning. 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 stand-alone applications. Most stand-alone 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 restoring
- ☐ Performing basic defragmentation and index rebuilding
- ☐ Using 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 application.

For purposes of this book, we’re going to stay with the GUI tools that are included in SQL Server Management Studio. Be aware, however, that there are more robust options available for the more advanced user that can allow you to programmatically control many of the administration features via the SQL Server Management Objects (SMO) object model.

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 the following criteria:

- ❑ 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 fire automatically when another finishes, 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 perform the following:

- ❑ 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 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 Management Studio has a little red square in it, 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, Try It Out, and Exercises 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 19-1. (Mine is partially filled in.)

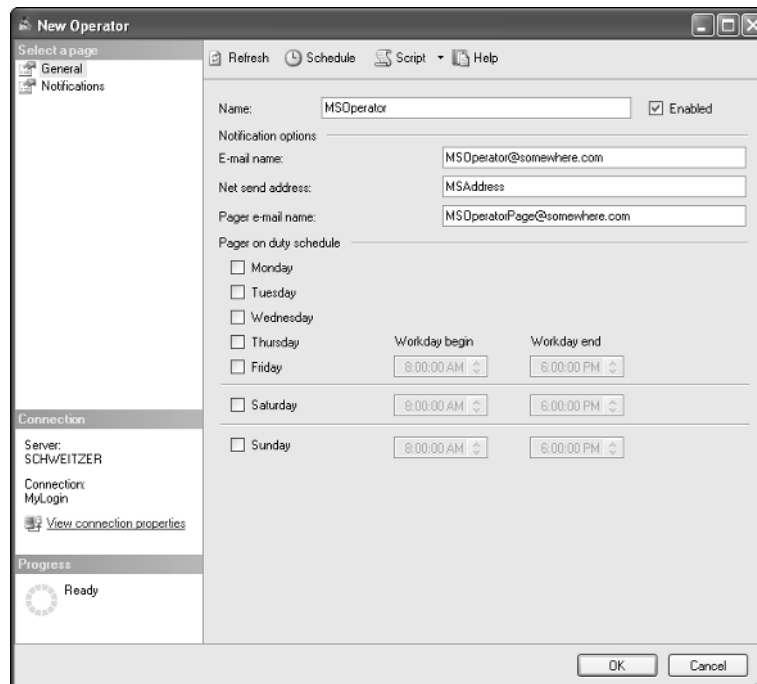


Figure 19-1

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.

Speaking of that Notifications tab, go ahead and click over to that tab. It should appear as in Figure 19-2.

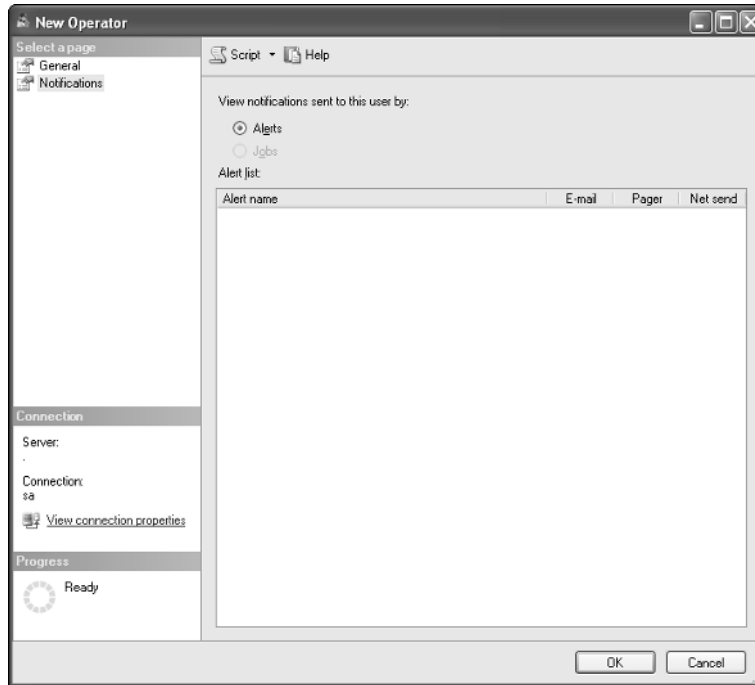


Figure 19-2

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 alters are triggered when certain things happen in your database, and this page defines which alters this particular operator receives.

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 to be 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 through programmatic constructs. For purposes of this book, we'll stick to the Management Studio method. (Even among highly advanced SQL Server developers, programmatic construction of jobs and tasks is very rare indeed.)

Creating Jobs and Tasks Using Management Studio

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 the Jobs member and select New Job. You should get a multimode dialog box, shown in Figure 19-3, that will help you build the job one step at a time:

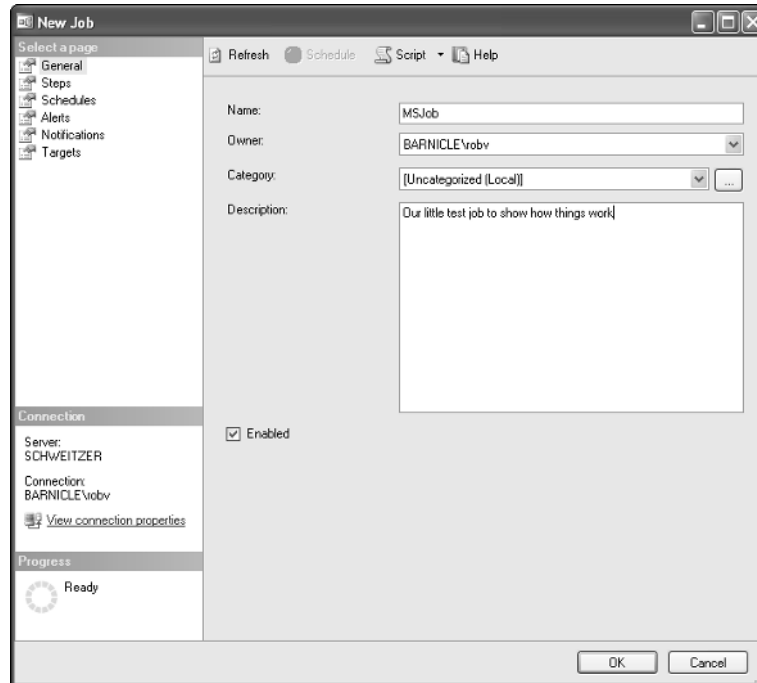


Figure 19-3

The name can be whatever you like as long as it adheres to the SQL Server rules for naming as discussed early in the book.

Most of the rest of the information is, again, self-explanatory with the exception of Category, which is just one way of grouping 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 19-4. This is the place where we tell SQL Server to start creating our new tasks that will be part of this job.

To add a new step to our job, we just click the New button and fill in the new dialog box, shown in Figure 19-5. 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.

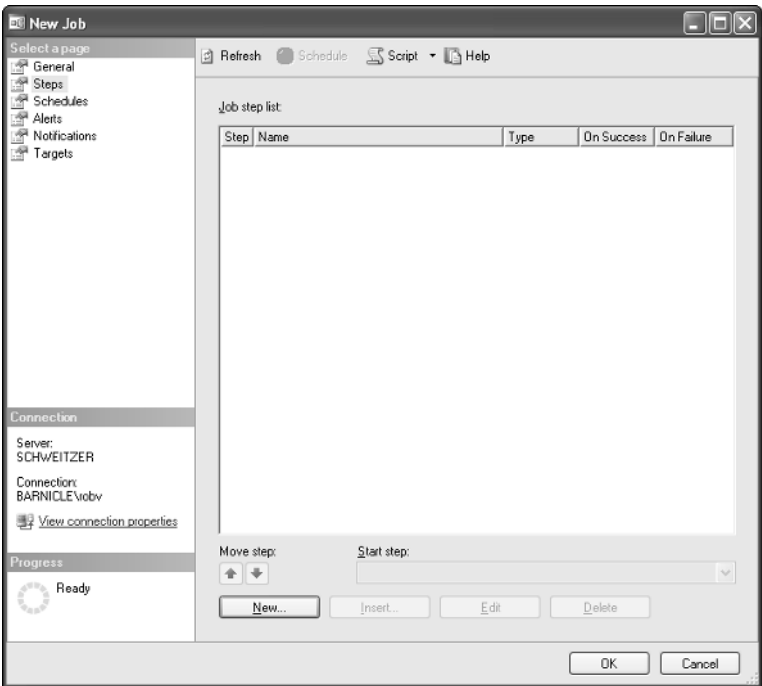


Figure 19-4

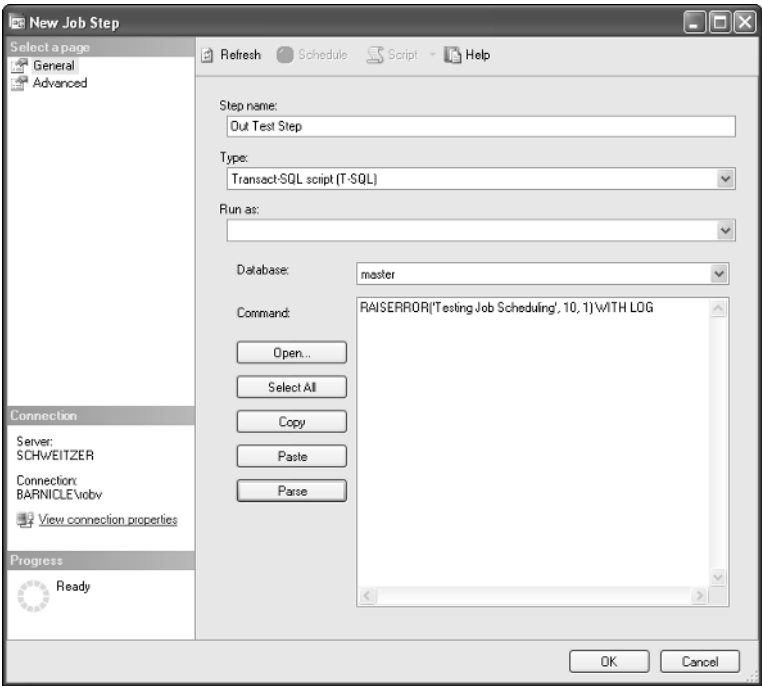


Figure 19-5

Let's go ahead and move on to the Advanced tab for this dialog box, shown in Figure 19-6; it's here that we really start to see some of the cool functionality that our job scheduler offers.

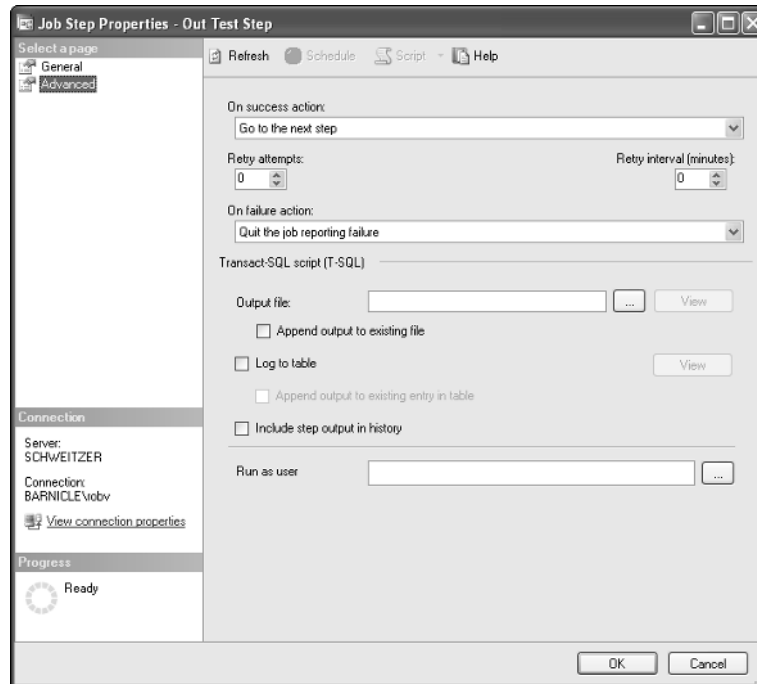


Figure 19-6

Notice several things in this dialog box:

- ☐ 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. Since 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 their own jobs this way anyway. (Let your client application provide that functionality.)

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.)

Chapter 19

That moves us back to the main New Job dialog box, and we're now ready to move on to the Schedules node, shown in Figure 19-7.

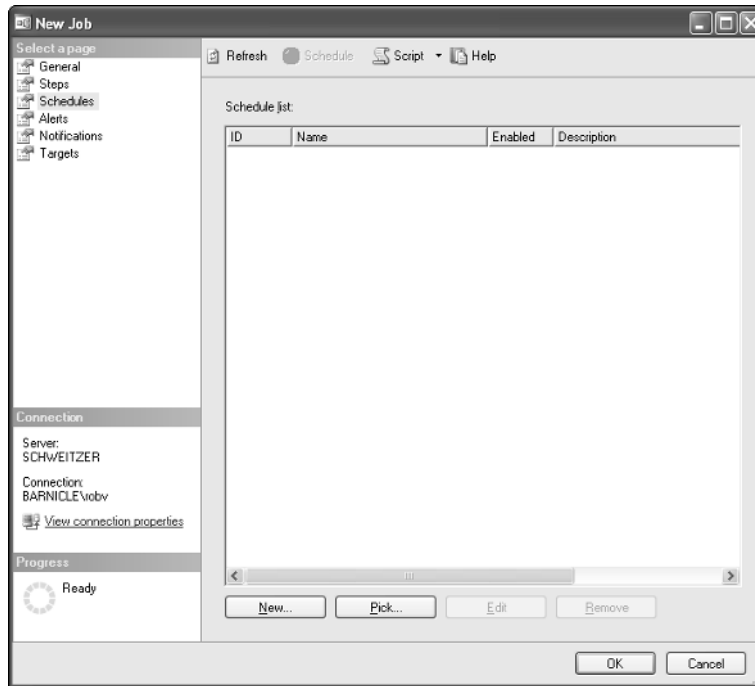


Figure 19-7

In this dialog box, 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 the New button. That brings us up yet another dialog box, shown in Figure 19-8.

I've largely filled this one out already (lest you get buried in a sea of screenshots), but it is from this dialog box 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 at 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 19-9.

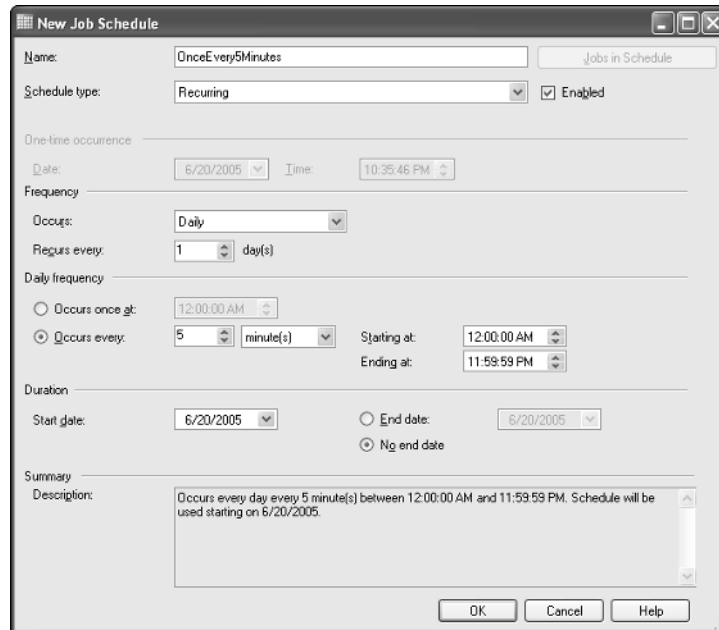


Figure 19-8

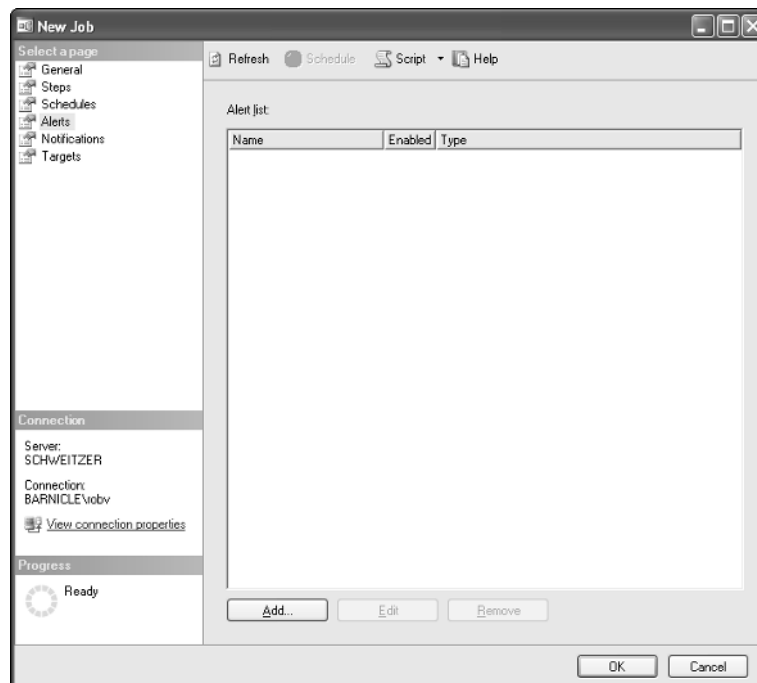


Figure 19-9

Chapter 19

From here, we can select which alerts we want to make depending on what happens. Choose Add and we get yet another rich dialog box, shown in Figure 19-10.

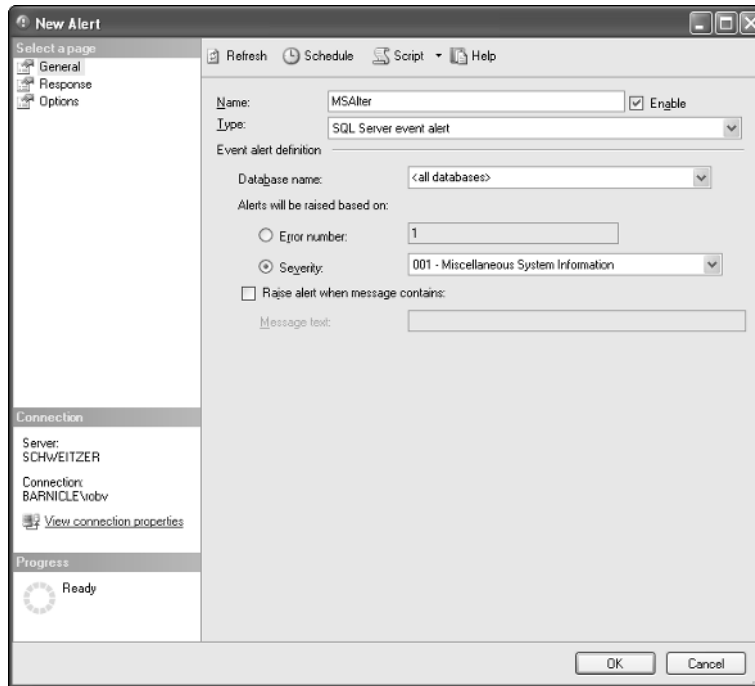


Figure 19-10

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, shown in Figure 19-11.

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 net send address to make the notification to. Also notice that we have control, on the right side, over how our operator is notified.

Last, but not least, we have the options node, shown in Figure 19-12.

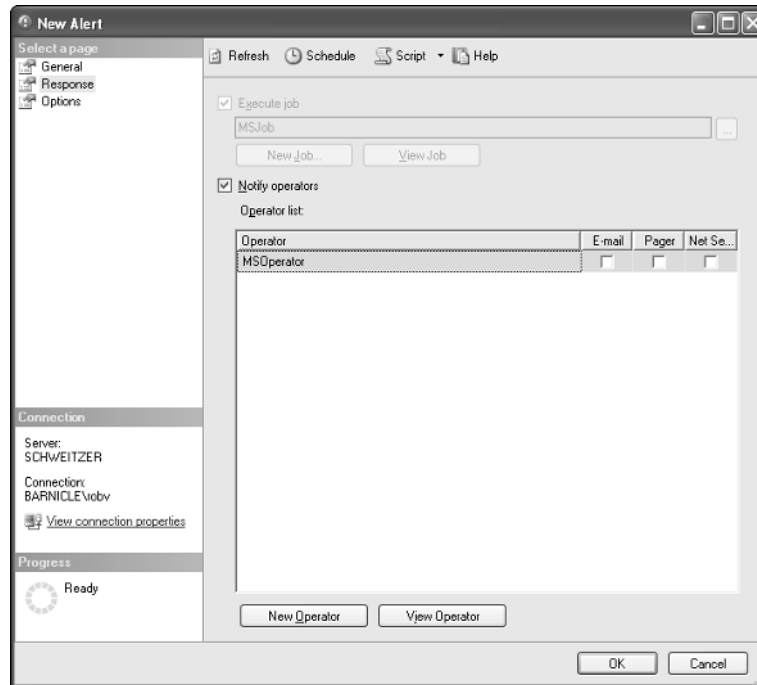


Figure 19-11

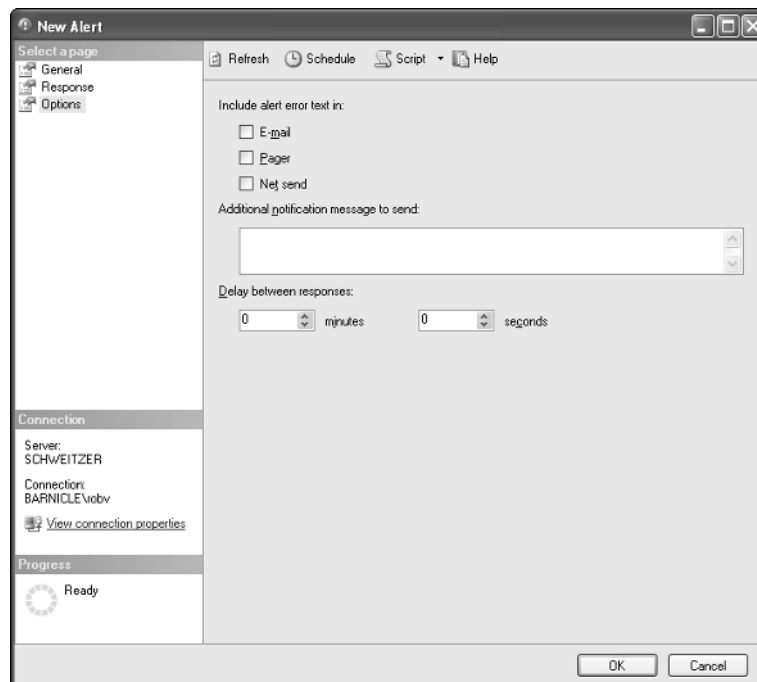


Figure 19-12

Finally, we can go back to the Notifications node of the main New Job dialog box, shown in Figure 19-13.

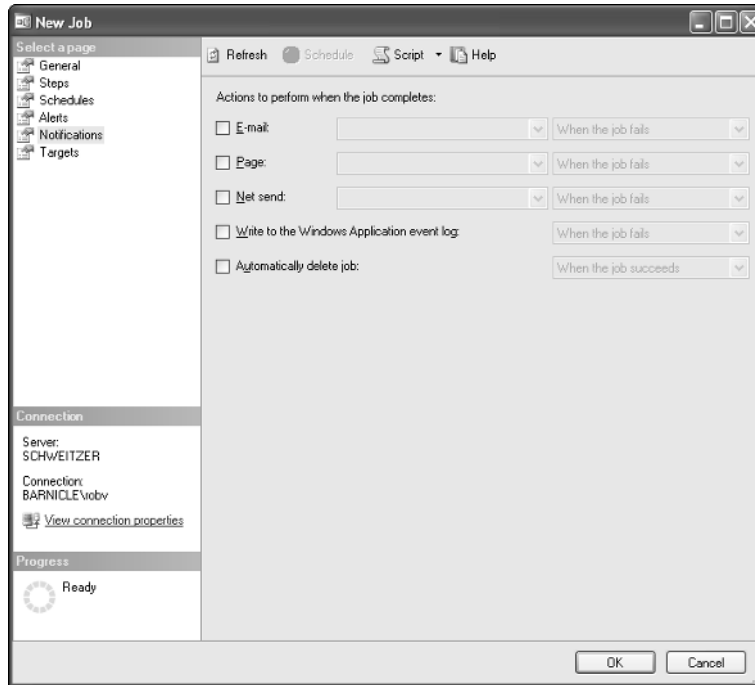


Figure 19-13

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 box.

At this point, you are ready to say OK and exit the dialog box. 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 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 navigate to the SQL Server Agent node of the Object Explorer in Management Studio.

Also, don't forget to disable this job. (Right-click 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.)

Backup and Recovery

No database-driven application 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: EEEEEEEEEK!

There is one simple rule to follow regarding backups; do it early and often. The follow up to this 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 applications being done by the relative beginner, you're probably going to stick with referring the customer or onsite 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 their 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. Simple navigate in Object Explorer to the database you're interested in, and right-click.

Now choose Tasks and Back up, as shown in Figure 19-14.

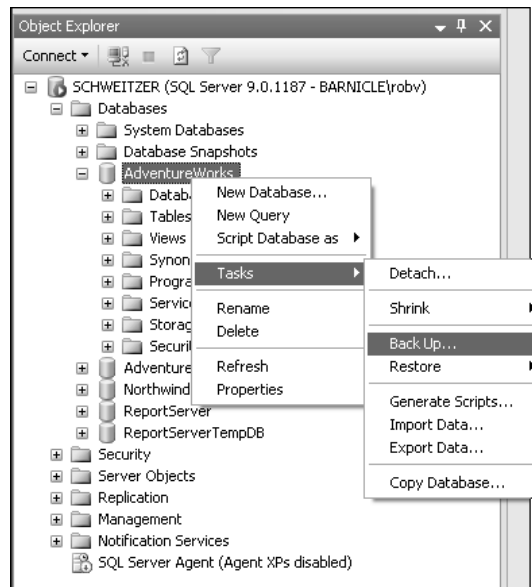


Figure 19-14

And you'll get a dialog box that lets you define pretty much all of the backup process as in Figure 19-15.

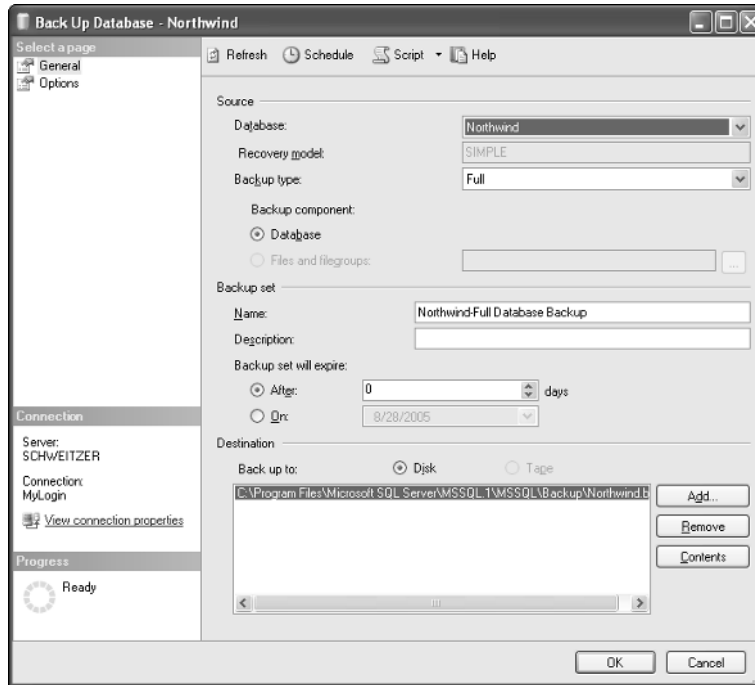


Figure 19-15

The first setting here is pretty self-explanatory — what database you want to back up. From there, however, things get a bit trickier.

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 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 9 if you’ve forgotten) that have changed since you did the last full backup. These typically run much faster than a full backup does 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 show up only 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 only applies 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 our chapters on database objects and creating database, 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. The option is 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 the devices you used as a destination available 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 box, 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 setup, 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 box is meant to facilitate you doing just that. Click it, and it will bring up the Job Schedule dialog box we saw earlier in the chapter. You can then define a regular schedule to run the backup you just defined.

Note that there are several more options for backups that are only addressable via issuing actual T-SQL commands or using the .NET programmability object model SMO. These are fairly advanced use, and further coverage of them can be found in *Profession SQL Server 2005 Programming*.

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 our chapter on transactions (see Chapter 14), 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 you 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 track of all the money that 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 type 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 box, as shown in Figure 19-16.

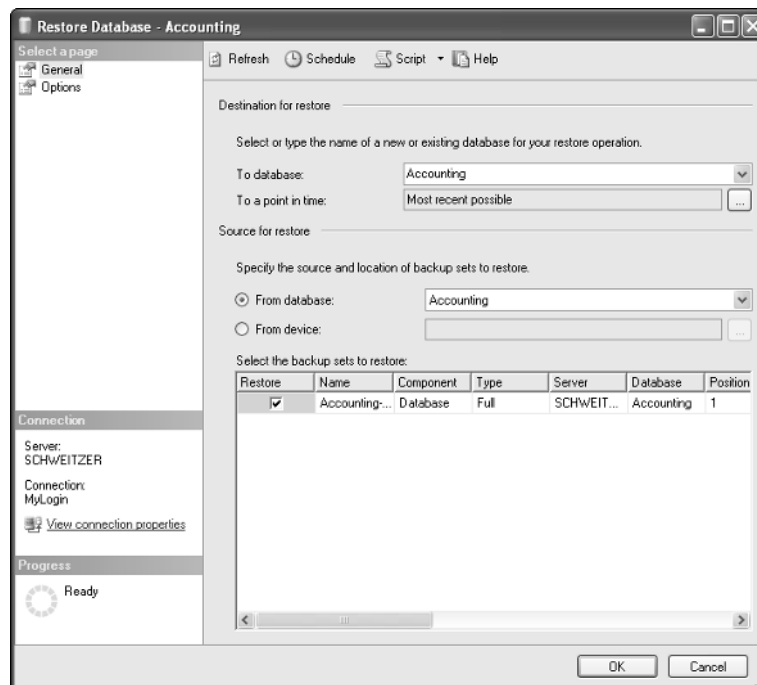


Figure 19-16

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 straight forward; 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 that was backed up, and, perhaps more importantly, 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 which physical files (the .MDF and .LDF files) it's going to try and store to. To deal with this, go to the Options node of the Restore dialog box.

Again, most of the options here are self-describing, but, in particular, notice the Restore As column. In this part of the dialog box, 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 be 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), the database would 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 from 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 be 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 what 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 and 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.

Index Maintenance

Back in Chapter 9, 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 a 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 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 makeup of your index, you still need to either DROP and CREATE it, or you need to CREATE and use the with the 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 fair amount on 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 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 a 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 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 a happy medium in life. When you reorganize your index, it 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 your real cost of fragmentation is, although 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 a index reorganization command.

Try It Out Index Reorganization

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 indexes on the table in one simple command:

```
ALTER INDEX ALL  
ON Production.TransactionHistory  
REORGANIZE;
```

You should get back essentially nothing from the database — just a simple “Command(s) completed successfully.”

How It Works

The `ALTER INDEX` command sees that `ALL` was supplied instead of a specific index name and looks up which 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).

Archiving Data

Ooh — here’s a tricky one. There are as many ways of archiving data as there are database engineers. If you’re building an 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 to, as a developer, think about many administrative tasks and establish what the increasingly 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 non-profit group that operates in the Northwestern 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!).

Exercises

1. Take the command that we made to reorganize our Production.TransactionHistory table, and schedule it to run as a weekend job Sunday mornings at 2AM.
2. Perform a full backup of the Pubs database. Save the backup to C:\MyBackup.bak (or alter slightly if you don't have enough disk space on that volume).
3. Restore the backup you created in Exercise 2 to a new database name — NewPubs.

A

Exercise Solutions

Chapter 3

1. Write a query that outputs all of the columns and all of the rows from the authors table of the pubs database.

With pubs as the current database, `SELECT * FROM authors`

2. Modify the query in Exercise 1 so it filters down the result to just the authors from the state of Utah. (Hint: There are 2 two.)

```
SELECT * FROM authors WHERE state = 'UT'
```

3. Add a new row into the authors table in the pubs database.

As an example of how to do this, you can perform the following: `INSERT INTO authors VALUES ('999-99-9999', 'Spade', 'Sam', '333 555-1212', '123 Main St.', 'Seattle', 'WA', '99999', 1)`

4. Remove the row you just added.

To follow along with the example in Exercise 3, you can perform the following: `DELETE authors WHERE au_id = '999-99-9999'`

Chapter 4

1. Write a query against the Northwind database that returns one column called “SupplierName” and contains the name of the supplier of the product called Chai.

```
SELECT CompanyName AS SupplierName FROM Products p JOIN Suppliers s ON  
p.SupplierID = s.SupplierID WHERE p.ProductName = 'Chai')
```

2. Using a JOIN, write a query that will list the name of every territory in the Northwind database that does not have an employee assigned to it. (Hint: Use an outer join!)

```
SELECT TerritoryDescription FROM Territories t LEFT JOIN EmployeeTerritories  
et ON t.TerritoryID = et.TerritoryID WHERE et.TerritoryID IS NULL)
```

Chapter 5

1. Using Management Studio's script generator, generate SQL for both the Customers and the Employees tables.

```
USE [Northwind]
GO
/***** Object: Table [dbo].[Customers]    Script Date: 08/26/2005
07:12:46 *****/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Customers](
    [CustomerID] [nchar](5) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
    [CompanyName] [nvarchar](40) COLLATE SQL_Latin1_General_CP1_CI_AS
NOT NULL,
    [ContactName] [nvarchar](30) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [ContactTitle] [nvarchar](30) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [Address] [nvarchar](60) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [City] [nvarchar](15) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [Region] [nvarchar](15) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [PostalCode] [nvarchar](10) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [Country] [nvarchar](15) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [Phone] [nvarchar](24) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [Fax] [nvarchar](24) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    CONSTRAINT [PK_Customers] PRIMARY KEY CLUSTERED
(
    [CustomerID] ASC
) ON [PRIMARY]
) ON [PRIMARY]

GO
SET ANSI_NULLS OFF
GO
SET QUOTED_IDENTIFIER OFF

USE [Northwind]
GO
/***** Object: Table [dbo].[Employees]    Script Date: 08/26/2005
07:13:21 *****/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Employees](
    [EmployeeID] [int] IDENTITY(1,1) NOT NULL,
    [LastName] [nvarchar](20) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
    [FirstName] [nvarchar](10) COLLATE SQL_Latin1_General_CP1_CI_AS NOT
NULL,
    [Title] [nvarchar](30) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [TitleOfCourtesy] [nvarchar](25) COLLATE
SQL_Latin1_General_CP1_CI_AS NULL,
    [BirthDate] [datetime] NULL,
    [HireDate] [datetime] NULL,
```

```

[Address] [nvarchar](60) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
[City] [nvarchar](15) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
[Region] [nvarchar](15) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
[PostalCode] [nvarchar](10) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
[Country] [nvarchar](15) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
[HomePhone] [nvarchar](24) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
[Extension] [nvarchar](4) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
[Photo] [image] NULL,
[Notes] [ntext] COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
[ReportsTo] [int] NULL,
[PhotoPath] [nvarchar](255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
CONSTRAINT [PK_Employees] PRIMARY KEY CLUSTERED
(
    [EmployeeID] ASC
) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]

GO
SET ANSI_NULLS OFF
GO
SET QUOTED_IDENTIFIER OFF
GO
USE [Northwind]
GO
ALTER TABLE [dbo].[Employees] WITH NOCHECK ADD CONSTRAINT
[FK_Employees_Employees] FOREIGN KEY(    [ReportsTo])
REFERENCES [dbo].[Employees] (    [EmployeeID])
GO
ALTER TABLE [dbo].[Employees] CHECK CONSTRAINT [FK_Employees_Employees]
GO
ALTER TABLE [dbo].[Employees] WITH NOCHECK ADD CONSTRAINT
[CK_Birthdate] CHECK (([BirthDate]<getdate()))
GO
ALTER TABLE [dbo].[Employees] CHECK CONSTRAINT [CK_Birthdate]

```

2. Without using Management Studio, script a database called “MyDB” with a starting database size of 17MB and a starting log size of 5MB; set both the log and the database to grow in 5MB increments.

One of several possible answers: CREATE DATABASE MyDB ON PRIMARY(NAME='MyDB', FILENAME='C:\Myfile.mdf', SIZE = 17MB, FILEGROWTH = 5MB) LOG ON (NAME='MyDBLog', FILENAME='C:\MyLog.ldf', SIZE = 5MB, FILEGROWTH = 5MB)

3. CREATE a table called Foo with a single variable length character field called “Col1”. Limit the size of Col1 to 50 characters.

```
CREATE TABLE Foo (Col1    varchar(50))
```

Chapter 7

1. Write a query that returns the start date of all Northwind employees in MM/DD/YY format.
SELECT CONVERT(varchar(12), HireDate, 1) FROM Employees

Appendix A

2. Write separate queries using a JOIN, a subquery, and then an EXISTS to list all Northwind customers who have not placed an order.

```
SELECT CompanyName
FROM Customers c
LEFT JOIN Orders o
  ON c.CustomerID = o.CustomerID
WHERE o.CustomerID IS NULL
SELECT CompanyName
FROM Customers C
WHERE c.CustomerID NOT IN (SELECT CustomerID FROM Orders o)
SELECT CompanyName
FROM Customers c
WHERE NOT EXISTS (SELECT CustomerID FROM Orders o WHERE o.CustomerID =
c.CustomerID)
```

3. Show the five most recent orders that were purchased from a customer who has spent more than \$25,000 with Northwind.

```
SELECT TOP 5 *
FROM Orders oo
WHERE CustomerID IN
(
  SELECT c.CustomerID
  FROM Customers c
  JOIN Orders o
    ON c.CustomerID = o.CustomerID
  JOIN [Order Details] od
    ON o.OrderID = od.OrderID
  GROUP BY c.CustomerID
  HAVING SUM(UnitPrice * (1-Discunt) * Quantity) > 25000
)
ORDER BY OrderDate
```

This could also be written using a JOIN and a derived table.

Chapter 8

1. Normalize the following data into Third normal form.

Patient	SSN	Physician	Hospital	Treatment	AdmitDate	ReleaseDate
Sam Spade	555-55-5555	Albert Schweitzer	Mayo Clinic	Labotomy	10/01/2005	11/07/2005
Sally Nally	333-33-3333	Albert Schweitzer	NULL	Cortezone Injection	10/10/2005	10/10/2005
Peter Piper	222-22-2222	Mo Betta	Mustard Clinic	Pickle Extraction	11/07/2005	11/07/2005
Nicki Doohickey	123-45-6789	Sheeze Sheila	Mustard Clinic	Cortezone Injection	11/07/2005	11/07/2005

Patients

PatientID	PatientName	SSN
1	Sam Spade	555-55-5555
2	Sally Nally	333-33-3333
3	Peter Piper	222-22-2222
4	Nicki Doohickey	123-45-6789

Physicians

PhysicianID	PhysicianName	Address
1	Albert Schweitzer	1234 anywhere
2	Mo Betta	567 Main
3	Sheeze Sheila	89 1 st

Hospitals

HospitalID	HospitalName	Address
1	Mayo Clinic	1234 Anywhere
2	Mustard Clinic	55 French's Avenue

Treatments

TreatmentID	TreatmentDescription
1	Labotomy
2	Cortezone Injection
3	Pickle Extraction
4	Cortezone Injection

Visits

VisitID	PatientID	PhysicianID	HospitalID	TreatmentID	AdmitDate	ReleaseDate
1	1	1	1	1	10/01/2005	11/07/2005
2	2	1	NULL	2	10/10/2005	10/10/2005
3	3	2	2	3	11/07/2005	11/07/2005
4	4	3	2	4	11/07/2005	11/07/2005

Chapter 9

1. Name at least two ways of determining what indexes can be found on the HumanResources.Employee table in the AdventureWorks database.

At least two ways of determining what indexes can be found on the HumanResources.Employee table in the AdventureWorks database are:

- a. Navigate to the Indexes node under the HumanResources.Employee table under Tables in the AdventureWorks database in Management Studio and expand the node; all indexes are listed there and can be double-clicked for more information.
 - b. Use `sp_help 'HumanResources.Employee'` or `sp_helpindex 'HumanResources.Employee'`. Both are about the same since the `sp_help` one has the `sp_helpindex` results embedded in it.)
2. Create a non-clustered index on the ModifiedDate column of the Production.ProductModel table in the AdventureWorks database:

```
CREATE NONCLUSTERED INDEX ieProductModel ON Production.ProductModel (ModifiedDate)
```

3. Delete the index you created in Exercise2.

```
DROP INDEX Production.ProductModel.ieProductModel
```

Chapter 10

1. Add a view called Managers in the Northwind database that shows only employees that supervise other employees:

```
CREATE VIEW Managers
AS
SELECT * FROM Employees o
WHERE EXISTS (SELECT 'true' FROM Employees i WHERE i.ReportsTo = o.EmployeeID)
```

2. Change the view you just created to be encrypted.

```
ALTER VIEW Managers
WITH ENCRYPTION
AS
SELECT * FROM Employees o
WHERE EXISTS (SELECT 'true' FROM Employees i WHERE i.ReportsTo = o.EmployeeID)
```

3. Create and index the existing Northwind view called “Products by Category” based on the columns CategoryName and ProductName.

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO

ALTER view [dbo].[Products by Category]
```

```

WITH SCHEMABINDING
AS
SELECT c.CategoryName, p.ProductName, p.QuantityPerUnit, p.UnitsInStock,
p.Discontinued
FROM dbo.Categories c
JOIN dbo.Products p
    ON c.CategoryID = p.CategoryID
WHERE p.Discontinued <> 1
GO

CREATE UNIQUE CLUSTERED INDEX ivProductsByCategory
ON [Products by Category](CategoryName, ProductName)

SET ANSI_NULLS OFF
GO
SET QUOTED_IDENTIFIER OFF
GO

```

Chapter 11

1. Write a simple script that creates two integer variables (one called Var1 and one called Var2), places the values 2 and 4 in them respectively and then outputs the value of the two variables added together.

```

DECLARE @Var1 int
DECLARE @Var2 int
SET @Var1 = 2
SET @Var2 = 4
SELECT @Var1 + @Var2

```

2. Create a variable called MinOrder and populate it with the smallest line item amount after discount for the Northwind CustomerNo 'ALFKI' (Note: Careful, we're dealing with currency here, so don't just assume you're going to use an int.) Output the final value of MinOrder:

```

DECLARE @MinOrder money
SELECT @MinOrder = MIN(Quantity * UnitPrice * (1-Discount))
FROM [Order Details] od
JOIN Orders o
    ON od.OrderID = o.OrderID
WHERE CustomerID = 'ALFKI'
SELECT @MinOrder

```

3. Use SQLCMD to output the results of the query `SELECT COUNT(*) FROM Customers` to the console window.

```

C:\SQLCMD -UMyLogin -PMyPassword -q"SELECT COUNT(*)_ FROM Customers"

```

Chapter 12

1. Write a simple stored procedure that returns the Customer record from the Northwind database given a parameter of the CustomerID:

```
CREATE spListCustomer @CustomerID
SELECT * FROM Customers
WHERE CustomerID = @CustomerID
```

2. Write a stored procedure that accepts a Territory ID, Territory Description, and Region ID and inserts them as new row in the Territories table in Northwind.

```
CREATE PROC spAddTerritory @TerritoryID nvarchar(20), @TerritoryDescription
nchar(50), @RegionID int
AS
INSERT INTO Territories (TerritoryID, TerritoryDescription, RegionID)
VALUES (@TerritoryID, @TerritoryDescription, @RegionID)
```

3. Alter the procedure you created in #2 to pre-check for the existence of the foreign key (RegionID) before attempting the insert. If the RegionID doesn't exist, throw an error with the error text "RegionID is not valid. Please check your RegionID and try again."

```
ALTER PROC spAddTerritory @TerritoryID nvarchar(20), @TerritoryDescription
nchar(50), @RegionID int
AS
BEGIN
    DECLARE @Count INT

    SELECT @Count = COUNT(*)
    FROM Region
    WHERE RegionID = @RegionID

    IF @Count < 1
        RAISERROR ('RegionID is not valid. Please check your RegionID and try
again.', 11, 1)
    ELSE
        INSERT INTO Territories (TerritoryID, TerritoryDescription, RegionID)
        VALUES (@TerritoryID, @TerritoryDescription, @RegionID)
END
```

4. Alter the procedure you created in #2 to handle the exception after the fact when a RegionID doesn't exist. Trap all other errors and provide the generic error message: "An unhandled exception has occurred. Contact your system administrator"

```
ALTER PROC spAddTerritory @TerritoryID nvarchar(20), @TerritoryDescription
nchar(50), @RegionID int
AS
BEGIN
    BEGIN TRY
        INSERT INTO Territories (TerritoryID, TerritoryDescription, RegionID)
        VALUES (@TerritoryID, @TerritoryDescription, @RegionID)
    END TRY
    BEGIN CATCH
        IF ERROR_NUMBER() = 547
            RAISERROR ('RegionID is not valid. Please check your RegionID and try
again.', 11, 1)
        ELSE
```

```

        RAISERROR ('An unhandled exception has occurred. Contact your system
administrator', 11, 1)
    END CATCH
END

```

Chapter 13

1. Reimplement the `spTriangular` function from Chapter 12 as a function instead of a stored procedure.

```

CREATE FUNCTION dbo.fnTriangular
(@ValueIn AS int)
RETURNS int
AS
BEGIN
    DECLARE @ValueOut int
    DECLARE @InWorking int
    IF @ValueIn != 1
    BEGIN
        SELECT @InWorking = @ValueIn - 1

        SELECT @ValueOut = @ValueIn + dbo.fnTriangular(@InWorking)
    END
    ELSE
    BEGIN
        SELECT @ValueOut = 1
    END
    RETURN (@ValueOut)
END

```

Chapter 19

1. Take the command that we made to reorganize our `Production.TransactionHistory` table and schedule it to run as a weekend job Sunday mornings at 2AM.

Schedule the command as follows:

- a. Navigate to the SQL Server Agent node and right-click the Jobs folder and select New Job.
- b. Name your new job and, optionally, set the category to Database Maintenance; then select the Steps page.
- c. Select New and name your new step. Leave the type option as T-SQL, and change your database in the database drop-down list to AdventureWorks.
- d. In the command box, type your command (the `ALTER INDEX` command we used on this table earlier in the chapter) and click OK.
- e. Select the Schedules page and click New.
- f. Name your new schedule, select Recurring for the schedule type, Weekly for the occurs box, and check Sunday. Set the Daily frequency to Once at 2AM, and click OK.
- g. Click OK a the main dialog and you're done.

2. Perform a full backup of the Pubs database. Save the backup to `C:\MyBackup.bak` (or alter slightly if you don't have enough disk space on that volume).

Perform the backup as follows:

- a. Navigate to the Pubs database node, right-click, and select Tasks ➔ Backup.
- b. Click Add in the Destination area at the bottom of the dialog box and type **C:\MyBackup.bak** in the pop-up box.
- c. Select the default data location (the one other than the one you just created) and click Remove. Click OK.

3. Restore the backup you created in Exercise 2 to a new database name: NewPubs.

Restore the back up as follows:

- a. Right-click the Databases node and select Restore Database.
- b. Type **NewPubs** into the To Database box.
- c. Select From Device and click the ellipsis (...).
- d. Click Add in the Specify Backup dialog box and navigate to `C:\MyBackup.bak`; click OK.
- e. Click OK again in the Specify Backup dialog box to return to the main Restore dialog box.
- f. Click the backup we just made in the Select backups to restore area, and then select the Options page of the dialog box (in the top-left of the dialog box).
- g. Change the filenames of `pubs.mdf` to `newpubs.mdf` and `pubs_log.ldf` to `newpubs_log.ldf`.
- h. Click OK and wait for restore to complete.

B

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 rarer in use and more cryptic in nature.

In this appendix, we’ll try and 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 strived 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 (aka, 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 things 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. 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

The number of rows 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. 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 created so far in the cursor. If however, the value is `-1`, the cursor is a dynamic cursor, in that the number of rows are constantly changing. A returned value of `0` informs you that either no cursor 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 to be 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 B

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, 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, and, if the `FETCH` failed, why. The constants are:

- ❑ 0 — Success
- ❑ -1 — Failed, usually because you are beyond the beginning or end of the cursorset.
- ❑ -2 — Failed. The row you were fetching wasn't found, usually because it was deleted between when the cursorset was created and when you navigated to the current row. Should only occur in scrollable, nondynamic 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 which 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 whether 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 on 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 cancelled.

Appendix B

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 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.

@@NESTLEVEL

Returns the current nesting level for nested stored procedures.

The first stored procedure (sproc) to run has an `@@NESTLEVEL` of 0. If that sproc calls another, then the second sproc is said to be nested in the first sproc (and `@@NESTLEVEL` 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 which values are set. 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_CLOSE_ ON_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 returns the number of input packets read/written from/to the network by SQL Server since it was last started.

Primarily, these are network troubleshooting tools.

@@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 non-runtime 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 would be 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 that 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>
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 returns 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.

@@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 when 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:

```
COUNT
(
  [ALL | DISTINCT] <expression> | *
)
```

The *expression* cannot be of the `uniqueidentifier`, `text`, `image`, or `ntext` datatypes. 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 previously, 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 which 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 whether that procedure has returned a cursor and result set. The syntax is as follows:

```
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 resultset 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 resultset has zero or more rows. If the cursor is not dynamic, it has one or more rows.
- ❑ 0 — The resultset 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` datatypes or which 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, and so on) 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>)
```

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>)
```

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 datatype.

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 datatype.

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 datatype.

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 datatype.

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 datatype.

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 datatype.

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 datatype.

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 datatype.

LOG10

The LOG10 function returns the base 10 logarithm of the value specified in *expression*. The syntax is as follows:

```
LOG10(<expression>)
```

The *expression* must be of the float datatype.

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 datatype. 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` datatype.

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` datatype.

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` datatype.

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` datatype.

Metadata Functions

The metadata functions provide information about the database and database objects. They are:

- ❑ `COL_LENGTH`
- ❑ `COL_NAME`

- ☐ COLUMNPROPERTY
- ☐ DATABASEPROPERTY
- ☐ DATABASEPROPERTYEX
- ☐ DB_ID
- ☐ DB_NAME
- ☐ FILE_ID
- ☐ FILE_NAME
- ☐ FILEGROUP_ID
- ☐ FILEGROUP_NAME
- ☐ FILEGROUPPROPERTY
- ☐ FILEPROPERTY
- ☐ fn_listextendedproperty
- ☐ FULLTEXTCATALOGPROPERTY
- ☐ FULLTEXTSERVICEPROPERTY
- ☐ INDEX_COL
- ☐ INDEXKEY_PROPERTY
- ☐ INDEXPROPERTY
- ☐ OBJECT_ID
- ☐ OBJECT_NAME
- ☐ OBJECTPROPERTY
- ☐ OBJECTPROPERTYEX
- ☐ 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 datatype of the column or parameter
- ☐ `Scale` — The scale for the datatype 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 datatype 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>')
```

Appendix B

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.
- ☐ `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 non-logged 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.
- ☐ `IsShutDown` — 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 log on 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 returns the setting for the specified database and property name. The syntax is pretty much 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.
- ❑ *IsArithmeticAbortEnabled*—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 filename in the current database. The syntax is as follows:

```
FILE_ID('<file_name>')
```

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 file name 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 following properties can be queried:

- ❑ `PopulateStatus` — The possible return values are 0 (idle), 1 (population in progress), 2 (paused), 3 (throttled), 4 (recovering), 5 (shutdown), 6 (incremental population in progress), and 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:

```
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 of datatype `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, or `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 auto create 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:

<code>CnstIsClustKey</code>	<code>CnstIsColumn</code>
<code>CnstIsDisabled</code>	<code>CnstIsNonclustKey</code>

Appendix B

CnstIsNotRepl	ExecIsAnsiNullsOn
ExecIsDeleteTrigger	ExecIsInsertTrigger
ExecIsQuotedIdentOn	ExecIsStartup
ExecIsTriggerDisabled	ExecIsUpdateTrigger
IsCheckCnst	IsConstraint
IsDefault	IsDefaultCnst
IsExecuted	IsExtendedProc
IsForeignKey	IsMSShipped
IsPrimaryKey	IsProcedure
IsReplProc	IsRule
IsSystemTable	IsTable
IsTrigger	IsUniqueCnst
IsUserTable	IsView
OwnerId	TableDeleteTrigger
TableDeleteTriggerCount	TableFulltextCatalogId
TableFulltextKeyColumn	TableHasActiveFulltextIndex
TableHasCheckCnst	TableHasClustIndex
TableHasDefaultCnst	TableHasDeleteTrigger
TableHasForeignKey	TableHasForeignRef
TableHasIdentity	TableHasIndex
TableHasInsertTrigger	TableHasNonclustIndex
TableHasPrimaryKey	TableHasRowGuidCol
TableHasTextImage	TableHasTimestamp
TableHasUniqueCnst	TableHasUpdateTrigger
TableInsertTrigger	TableInsertTriggerCount
TableIsFake	TableIsPinned
TableUpdateTrigger	TableUpdateTriggerCount
TriggerDeleteOrder	TriggerInsertOrder
TriggerUpdateOrder	

The return value from this function will be 1 for True, 0 for False, and NULL if the input was not valid, except for the following:

- ❑ OwnerId (which will return the database user ID of the object owner)
- ❑ TableDeleteTrigger, TableDeleteTriggerCount, TableInsertTrigger, TableInsertTriggerCount, TableUpdateTrigger, TableUpdateTriggerCount (all of which will return the ID of the first trigger with the given type)
- ❑ TableFulltextCatalogId and TableFulltextKeyColumn (both of which will return the full-text catalog ID, or a 0 to indicate that the table has not been full-text indexed, or a NULL to indicate that the input was invalid)

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 as follows:

```
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 sql_variant returned
BaseType	Data types include: char, int, money, nchar, ntext, numeric, nvarchar, real, smalldatetime, smallint, smallmoney, text, timestamp, tinyint, uniqueidentifier, varbinary, varchar	sysname
Precision	The precision of the numeric base data type: datetime = 23 smalldatetime = 16 float = 53 real = 24 decimal (p,s) and numeric (p,s) = p money = 19 smallmoney = 10 int = 10 smallint = 5 tinyint = 3 bit = 1 all other types = 0	int

Table continued on following page

Value	Description	Base type of sql_variant returned
Scale	The number of digits to the right of the decimal point of the numeric base datatype: decimal (p,s) and numeric (p,s) = s money and smallmoney = 4 datetime = 3 all other types = 0	int
TotalBytes	The number of bytes required to hold both the meta data and data of the value. If the value is greater than 900, index creation will fail.	int
Collation	The collation of the particular sql_variant value.	sysname
MaxLength	The maximum data type length, in bytes.	int

TYPEPROPERTY

The TYPEPROPERTY function returns information about a datatype. 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 datatype 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 26 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 26 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 provided as specified in the registry. The *data_source* parameter is a string corresponding to the required OLE DB

Appendix B

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 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 was a table. The syntax is as follows:

```
OPENXML(<idoc_int> [in],<rowpattern> nvarchar[in], [<flags> byte[in]])  
[WITH (<SchemaDeclaration> | <TableName>)]
```

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.

For more information on using OpenXML, please refer to Chapter 20.

Security Functions

The security functions return information about users and roles. They are as follows:

- ☐ HAS_DBACCESS
- ☐ IS_MEMBER
- ☐ IS_SRVROLEMEMBER
- ☐ SUSUSER_ID
- ☐ SUSUSER_NAME
- ☐ SUSUSER_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 it 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.

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 available:

- ☐ ASCII
- ☐ CHAR
- ☐ CHARINDEX
- ☐ DIFFERENCE
- ☐ LEFT
- ☐ LEN
- ☐ LOWER
- ☐ LTRIM
- ☐ NCHAR
- ☐ PATINDEX
- ☐ QUOTENAME
- ☐ REPLACE
- ☐ REPLICATE
- ☐ REVERSE
- ☐ RIGHT
- ☐ RTRIM

Appendix B

- ☐ SOUNDEX
- ☐ SPACE
- ☐ STR
- ☐ STUFF
- ☐ SUBSTRING
- ☐ UNICODE
- ☐ UPPER

ASCII

The ASCII function returns the ASCII code value of the left-most 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>])
```

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>)
```

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>)
```

Appendix B

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
- ☐ 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 datatype 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 datatype.

COALESCE

The COALESCE function is passed an undefined number of arguments and it tests for the first nonnull 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:

Property Name	Description
CodePage	The nonUnicode 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 `DATLENGTH` 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` datatypes because these data types can store variable-length data. The syntax is as follows:

```
DATLENGTH(<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 US 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 `NULL` 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.

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([<object_id> [, '<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 previously, 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()
```

SERVERPROPERTY

The `SERVERPROPERTY` function returns information about the server you are running on. The syntax is as follows:

```
SERVERPROPERTY ('<propertyname>')
```

The following table shows the possible values for *propertyname*:

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 <code>nvarchar</code> 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	Determines whether the server instance is configured in a failover cluster: 1 — Clustered 0 — Not clustered NULL — Invalid input or error
IsFullText Installed	Determines whether 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	Determines whether the server is in integrated security mode: 1 — Integrated security 0 — Not integrated security NULL — Invalid input or error
IsSingleUser	Determines whether the server is a single-user installation: 1 — Single user 0 — Not single user NULL — Invalid input or error

Table continued on following page

Property name	Values returned
IsSync WithBackup	Determines whether 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
LicenseType	What type of license is installed for this instance of SQL Server: PER_SEAT — Perseat mode PER_PROCESSOR — Perprocessor 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 pe-seat mode. Number of processors licensed for this instance of SQL Server, if in perprocessor mode.
ProcessID	Process ID of the SQL Server service. (The <code>ProcessID</code> is useful in identifying which <code>sqlservr.exe</code> belongs to this instance.)
Product Version	Very much like Visual Basic projects, in that the version details of the instance of SQL Server, are returned, in the form of <code>'major.minor.build'</code> .
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>)
```

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 run-time.

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 as follows:

- ❑ `PATINDEX` (this was covered in the *String Functions* section)
- ❑ `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.

C

Finding the Right Tool

So, here it is — tool time. I cover it as an appendix in both the *Beginning* and *Pro* titles for one reason: It doesn't matter whether you're a beginner or an expert; there is always something you're looking to do that goes beyond what comes "out of the box."

I need to start off this appendix by calming expectations a bit. You are not going to hear me make some, "Oh, you *have* to buy this product!" kind of recommendation. Indeed, you're going to find me trying fairly hard to keep from favoring any particular product (though you may hear me frown a bit about what's happened to old favorites).

What I'm after in this appendix is more just to show you some of the tools and features that are out there — essentially, I want to show you the possibilities. Realistically speaking, it doesn't matter to me which tool you use as long as you are getting the most from your experience with SQL that you can (a sentimental kind of comment, I know — but a sincere one).

In this appendix, we're going to look at just a couple of the types of different tools that you should consider adding to your arsenal. The ones I'll touch on here are:

- ☐ ERD (diagramming) tools
- ☐ Code tools
- ☐ Backup utilities

ERD Tools

ERD stands for *Entity Relationship Diagram*. This is the cornerstone of both understanding and relating your database design to others. If you're serious about database development, then buy a serious ERD tool — end of story.

Now that I've gone and said that, I should probably brace you for what you're going to pay — figure on a \$1,000 *minimum*, and probably double that or more (I'm aware of at least one that goes up to \$15K per seat — but it's also a *very* powerful product). What exactly these tools offer you varies by product, but let's take a look at some specific items so you get an idea of what these products can do for you.

Logical and Physical Designs

All of the major products support the concepts of the separation of logical vs. physical design. Logical modeling probably did not get its due in this book, because we stuck largely to the build in tools for modeling, and they just don't address logical design. As you get into more advanced modeling, however, learning to separate logical vs. physical design becomes rather important.

For purposes of context, let me synopsise the difference between the two model types this way:

The logical model is about modeling the high level relationships between the *entities* that you are working with. It tends to have a higher level outlook on things, and lets you concentrate more on what you're modeling and less on the implementation of that model. Keeping a logical model is very important to understanding what we're trying to build as a whole without getting mired in the details. In addition, your logical model may include the representation of data sources that are external to your database (say, a database of credit card numbers that you access via some service rather than store in your own database).

The physical model is what the database actually looks like in the end. The physical model is all about the actual tables and views in use and what the relationships are between those database objects.

Several tables in the physical model may represent one "Entity" in a logical database. For example, an Order is probably just an order in a logical model, but it would require both a header table (for the attributes that are common to the entire order) and a detail table (for the individual items ordered) in the physical database. Likewise, we can model the idea of a many-to-many relationship in a logical table by showing the direct relationship, but in a physical model we have to utilize an associate table that has one to many relationships with the two parent tables (RDBMS systems are not capable of modeling a many-to-many relationship directly at this time).

Your ER tool should facilitate this process. It should allow you to have entities and relationships in the logical model that do not necessarily need to exist in the physical side of things. These can range from query components that are implemented instead of stored procedures, to working tables that only exist in memory somewhere (for example, a data cache that your program components keep track of). Your logical model should be able to pull all of these data constructs together in your logical model, and still be able to easily separate those that won't be in the physical database from those that will. It should also make it easy to migrate those logical constructs that will exist in the actual database into the physical model with little fuss.

General Scripting

Pretty much all of the ER tools are capable of converting your physical diagram into scripts to generate your database. Perhaps what's the greatest about this is that most of them also support multiple platforms — this means that you can develop your entity relationships once, but be able to generate the script for multiple back-end servers. You'll find that almost all support SQL Server and Oracle. Support is also common for DB/2, Access, MySQL, and others. This can be a real time saver when you have to deal with multiple platforms.

Reverse Engineering

OK, so you've been able to create a diagram and generate a database from it — but what if you want to go the other way? The tool you choose should support the notion of *reverse engineering* a database — that

is, connect to a database, scan it, and generate a diagram that properly reflects all the tables, views, triggers, constraints, indexes, and relationships.

I say “should” rather than “could,” because there may well be the times when you do need to reverse engineer a database. It is better to have bought the tool that does this, rather than finding yourself having to buy two ERD tools.

Fortunately, most of the high-end tools I’m talking about in this section will do this for you and actually do a very good job of it. Some will even import sprocs for you.

Synchronization

One of the biggest problem areas in database work is any change made once you have created the physical database. This is even more of a problem once you start getting live data loaded into the database. In short, they are usually a monumental hassle to take care of.

Just the issue of trying to keep track of what you’ve changed in the model and what has actually been propagated out to the database can be very tedious at best—it is incredibly simple to not get a change out to a database. That’s the first place an ERD tool with synchronization can help.

Several of the major ERD vendors have products that will look over your database, compare that to your physical model, and then give you a list of discrepancies. This can be a real lifesaver as you continue to make changes both small (sometimes these are the most dangerous—you tend to forget about them) and large to your database. Another time that this functionality comes into its own is when you are working on a system where remote access is provided for overnight support. Any changes made can then be synchronized back into the ERD. But wait—there’s more.

Once you recognize that you have a difference in your design (which should be reflected in the ERD), you have to write the code to make the actual change.

At first, writing the code to make the change to the database may not seem like too big a deal. Indeed, if what you’re doing is adding one more column to the end of a table, then it’s no big deal at all. Imagine, however, that you want to add a column, but you want it to be in the middle of the column order. Hmmm! That means that you have to:

1. Create the table with the new layout using a different name.
2. Copy all the data to the appropriate columns of the new table.
3. Delete the old table.
4. Rename the new table.

That’s only a few steps—and it’s already a hassle. Now let’s add a little more reality check to it. Now think about if you have foreign keys that reference the old table. Hmmm, again! Those are going to have to be dropped before the table can be deleted (Step 3 in the preceding list); plus they (and the statistics) are going to have to be re-created again. Now what have we got?

1. Create the table with the new layout using a different name.
2. Copy all the data to the appropriate columns of the new table.

3. Drop the foreign keys from any table that is referencing the old table.
4. Delete the old table.
5. Rename the new table.
6. Re-create the foreign keys for all those tables that we just dropped them from.

This is starting to get pretty complex, eh?

OK, so that takes us back to the “more” I mentioned before. Some (not all) of the major ERD tools will script this for you. After the comparison phase of the synchronization is complete, they’ll give you a dialog to compare — one side for the ERD, and one side for the database. You get to decide which side of the diagram wins for each difference — do you propagate the ERD to the database, or do you accept the difference in the database as needing to be part of the ERD?

In short, this kind of thing can save hours and hours of work.

Now, after saying all that, I have to give you a word of caution — these tools are not foolproof in the way they script the changes. If you’re running update scripts against a live database, make sure that you have thoroughly tested the script the ERD tools generates against test databases before you run the change on your live data.

Macros

Some of the tools give you the capability to build macros in your ERD. These may be used for simplifying repetitive tasks, but they can also do things like automate the generation of stored procedures and triggers by automatically substituting the data names and datatypes into trigger and sproc templates that you create.

The really great thing about macro-based development is how flexible it is to change — when you change the table, your sprocs and triggers are also automatically changed (depending on how well you’ve used the macro language).

On the down side of this one is learning time. It takes a while to get to know the macro languages (which are proprietary — if not in actual language, then most certainly in the object model used to represent your database objects), and it takes even longer to fully grasp all the places where you can make use of macros once you’ve learned the language.

Integration with Other Tools (Code Generation)

Yes — you can even get versions of ER tools that either have some code generating capabilities of their own, or they integrate with other tools to do so. This can be really useful from a couple of perspectives — prototyping and integration to logical model.

Let’s say, for example, that you have a logical model that calls for several non-database data objects — the tool can generate template code for you for those objects. For example, it can create the code to expose properties and provide stubs for method calls. This eliminates a ton of rather tedious work.

Much like the preceding synchronization section, you need to be a little careful of these tools. Don't just assume that everything in them is going to be correct — proof the code that was generated and make sure that everything that you expect is there. Even with the time spent looking it over, you can still pick up quite a timesaving using these code generators.

Other Things

Some other items to think of or watch out for include:

- ❑ **Automatic loading of domain data:** Most databases have several domain tables that need to have data loaded into them that will be constant (a table containing all the states or a list of countries are common examples of this). The problem is that, every time you regenerate your database, you have to reload the domain data. Some of the tools will allow you to build “pre-load” scripts right into the ERD — when you generate the database, these tables are preloaded with the desired data every time.
- ❑ **Cut & Paste:** Some of the tools do not support linking and embedding — that is, you can't cut them out of the ERD tool and paste them into Word or some other word processor. Needless to say, this can be a huge hassle when you go to document your database.
- ❑ **Diagram methodologies supported:** You'll find that pretty much all the tools support the two most common methodologies: IDEF1X and IE. What's different is the level that they support these methodologies. For example, at least one of the major tools accepts using IE for the physical database, but won't allow it for the logical database (instead, forcing you to use their own proprietary methodology). If you don't mind this — then no big deal — but be sure you know what you're getting.
- ❑ **Subject areas:** Some databases get positively huge in the number of separate entities that they have. This can make managing your ERD on a computer screen (which is usually no more than 21" at most — and that's only for the luckier of you out there) virtually impossible. One way that some of the tools get around this is through the concept of subject areas. Subject areas allow to effectively filter the main drawing. You decide what tables are going to appear in the subject area. This means that you can manage different parts of the database in smaller sections.
- ❑ **Integration with source control:** Only the truly high-end tools (\$10,000+ a seat) seem to have a decent level of integration with source control utilities (such as Source Safe or PVCS). I've been hoping for a change on this, but have yet to see it. We can, however, keep hoping.

The big thing to understand here is that the storage format of your ERD diagrams is such that you're unlikely to be able to diff a given ERD against another version to expose what actual changes were made.

A Few Examples

Some of the most common products in this space include:

- ❑ **ErWin:** The venerable one. This was the benchmark standard of the old days and used to rule the roost alone. While it has plenty of competition these days, it continues to be one of the pre-eminent products in the premium end of the ER Diagramming market space.

- ❑ **ER Studio:** ER Studio was really the first to put pressure on ErWin in the larger market place. While other products had been competing with ErWin for some time, ER Studio was the first to change the way we thought about some of the features of a ER diagramming tool. Other tools have taken note and ER Studio certainly does not have the product distinctions that it used to have versus the other products in this space, but it remains among the leaders.
- ❑ **PowerDesigner:** Another of the older tools in this space. This one is particularly strong in the logical vs. physical model separation side of modeling.
- ❑ **Visio:** This one lacks some of the more complex features of the above tools, but also lacks the price tag. Visio has ways of representing most of the common diagramming methodologies, but has weaknesses in the cross platform and general scripting side of things.

Coding Tools

In C#, C++, Visual Basic, Java, or any of the other modern programming languages, we have begun to expect very robust development environments. High integration with source control and excellent debugging tools are just the beginning of the things that we now pretty much expect to be there in main-stream development tools. For SQL, we are just now “getting there” on how these work.

With SQL Server 2005, Microsoft moved to using the Visual Studio development environment even for basic T-SQL statements. If you’re in a pure T-SQL or pure .NET language environment, then you’re probably set with that. If, however, you need to support other platforms, you may want to take a look at some of the third-party SQL coding environments out there. Just be prepared for a much less integrated debugging environment.

A Few Examples

Having a separate product for Query Editing has continually lost some favor since version 7.0 of SQL Server did so much to enhance the editing environment. Visual Studio getting smarter about SQL queries has also increased this trend. That said, there remain several tools out there that have their own strengths (perhaps the biggest being the ability to connect to more than just SQL Server). Some of the more common tools include:

- ❑ **Ultra Edit:** This one won’t even actually connect to your database for you. Instead, this is just a text editor. The reason I see this one used is that it allows for a custom keyword library to be used (allowing color coding of keywords and such) while providing very robust text editing capability (which is very handy when doing global search and replace or similar text editor sorts of functions).
- ❑ **SQL Editor:** By the same company as ER Studio (Embarcadero), this one allows for cross-platform queries.
- ❑ **TOAD:** Made by one of the leading manufacturers of cross-platform SQL administration tools, this one is very well known in the Oracle community. It is a fairly robust development environment, but one that has all the pros and cons that one might expect from something that is not beholden to just one database server provider.
- ❑ **SQL Compare:** A tool for comparing two databases and scripting converting differences.

Backup Utilities

OK, so this is more of a DBA thing than a developer thing, but I would argue that developers still need to think about things like backups. Why? Well, for a lot more than just making sure that you don't lose the data on your development system. Instead, I would argue the biggest motivation here should be to help out your end users — particularly if what you're developing is likely to go into an installation that won't have a true DBA.

SQL Server has its own backup options — just like it has its own ways to diagram a database and enter code. With each new release, Microsoft has placed some effort in ensuring that backups are easier than in the past. That said, there is still more that can be done, and there are, as with the other things we've discussed, third-party tools to help.

Some of the major backup tools provide a much more useful user interface than SQL Server's. In addition, many of these tools will allow you to back up to mixed storage devices as well as do things like compress the backup on the fly as it is being written out (you would be amazed at how much space this can save).

Backup utilities are not, as a developer, the first place I would spend my money. However, I still recommend that you take a look at what's available so you're prepared to make recommendations to your customers.

A Few Examples

SQL Backup: Made by the same manufacturer as SQL Compare (Red Gate). This is a relatively low cost third-party backup management utility.

Backup Exec: This one is part of a larger backup utility approach that is not just limited to SQL Server. While it has tools for backing up the entire O/S, it understands the SQL Server backup architecture and can automate that process as well as keep track of a library of past backups.

Summary

The tools for SQL Server work pretty much as tools for anything do — you don't necessarily have to have the right tool for the job, but it sure can make a huge difference in both productivity and the level of satisfaction (vs. frustration) you have in your job and in the end product.

Take the time to look at what's available on the market. Even though some of the tools may seem expensive, be sure you think about what they might save you — as how much that savings is worth. I suspect you'll be buying one or more of the tools listed in this appendix.

D

Very Simple Connectivity Examples

So, here we have a book chock full of stuff on how to use a SQL Server database. This is, of course, all great and wonderful if all you're doing is just working right in the database. This does not, however, reflect the way we actually do things. Sure, we may log into the 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 you'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 to the basics here. The goal, then, is to cut to the chase on these examples. We'll provide a simple example of how to connect, execute a query, and disconnect for C#, VB.NET, and C++.

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 or the coverage that is in the follow-on book to this one: Professional SQL Server 2005 Programming.

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 OLEDB, 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 was 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 user name, password, database and server you wish to connect to. Some of the methods that will exist include such things as connect and disconnect.
- ❑ **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).
- ❑ **Dataset:** 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 it does, there will be some sort of dataset (sometimes called a result set or recordset) that it returns into. Dataset 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). This will also generally allow for data to be updated, inserted, and deleted.

Our examples here will be limited to ADO.NET, but consider the preceding objects, and you should be able to cross over any of the concepts to most other object models.

Connecting in C#

C# is somewhat hard to ignore. When I wrote my last book, this was really just an up and coming thing, but C# has had a fairly warm reception. It is a fairly clean language and is relatively easy to learn, much like VB is, but 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).

So let's look at our code:

```
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

        string strConnect = "Data Source=(local);Initial Catalog=master;Integrated
Security=SSPI";
        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
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"]);
}
finally
{
    // Clean up
    if(rsMyRS != null)
    {
        rsMyRS.Close();
    }

    if(cnMyConn != null)
    {
        cnMyConn.Close();
    }
}
}
}

```

Connecting in VB.NET

Again, we're going with pretty simplistic code that imports a few libraries, opens a connection, runs a query, and then lists what it finds out to the console. Consider stepping through this to look at what is happening in each phase.

```

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

        Dim strConnect As String = _
            "Data Source=(local);Initial Catalog=master;Integrated Security=SSPI"
    
```

```
Dim strCommand As String = _
    "SELECT Name, database_id as ID FROM sys.databases"

Dim rsMyRS As SqlDataReader

Dim cnMyConn As New 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 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

' Clean up
rsMyRS.Close()
cnMyConn.Close()

End Sub

End Module
```

E

Installing and Using the Samples

Throughout this book, we use quite a few example databases. While many of them are databases we create for ourselves as we go along, some of them are supplied in one form or another by Microsoft. This appendix is all about getting the right samples installed properly.

Samples Used in This Book

Database	Source	Notes
Northwind	Downloaded from Microsoft Website	Follow the link in the SQL Server Books Online.
AdventureWorks	Included with the SQL Server 2005 CD	Not installed by default — make sure you install this before doing the samples in this book.
Accounting	Created In Chapter 5, and added to throughout the book	Altered many times over the course of the book. Make sure, if you are doing an example that uses this database, that you've run the examples from previous chapters.
TestDB	Created and dropped several times in the book	Usually lasts only a page or two. Each use is completely independent of previous times we would have created this database, so be sure and drop the previous version if working on a TestDB from a new chapter.

Table continued on following page

Database	Source	Notes
Northwind Triggers	Created from a script downloadable from the wrox.com Website	Used in Chapter 15 to illustrate several issues related to triggers.
NorthwindBulk	Created from a script downloadable from the wrox.com Website	Creates a database based on the Northwind database, but containing many, many more rows.
Pubs	Downloaded from Microsoft Website	Just another very small database sample that you will see utilized for many examples you might find on the Internet.

Microsoft-Supplied Databases

SQL Server 2005 ships with just one sample database included on the CD, but we also utilize the older sample — Northwind — extensively in this book.

Many at Microsoft were less than thrilled at my choice to stick with the Northwind database for many of the samples in this book. I'm not going to pick any bones about what a difficult decision this was. The reality, however, is that I despise the newer AdventureWorks database from a teaching perspective. It is a step forward in the sense that it is a more robust database and provides for a much broader array of examples. The problem, however, is that it is both overly complex for teaching beginning database concepts as well as even less "real world" in the sense that it utilizes some SQL Server features far more aggressively than you're likely to see in the real world.

There is little doubt that the "real world" databases out there will slowly get more aggressive about utilizing some of the rather impressive features added in SQL Server 2005 over time, but I suspect that is a long way off, and that it will never reach the ambition shown in the design of AdventureWorks. As such, I decided to stick with the much more simplistic (arguably overly simplistic) Northwind for many examples, and utilize AdventureWorks for the more advanced examples where that made sense.

Let's look at how to get both of these installed as well as the even more venerable Pubs sample.

AdventureWorks

It's important to note that this sample, while the only one included on the CD, is *not* installed by default. That is, you need to manually select it at install time or it will not be available to you once you start your server up (you can, however, re-run the installation later to have the AdventureWorks sample installed).

The key here is that you need to choose the "custom install" option rather than "typical." When it comes to the question of what components to install, find the AdventureWorks example (don't confuse this with the AdventureWorksDW database that is a data warehouse example) and set it to be included in the installation. From there, it will be available the first time you go to access your newly installed server.

Northwind

This is an older sample that Microsoft used to provide with its database products. It actually originated with the Access team many years ago, and first made its appearance in SQL Server with version 7.0. You will find a huge number of examples published by miscellaneous authors on the Web that utilize this sample. We utilize this example in virtually every chapter in the book.

This sample is a bit trickier than AdventureWorks to get installed. In order to get this database, I recommend going to the Books Online and searching for Northwind. A link is provided in Books Online, which, if you happen to be utilizing the Web-based version, will always be up-to-date on wherever Microsoft may have chosen to move the installation scripts for this sample.

Once you've downloaded the installer for Northwind, go ahead and run it.

The installers for the Northwind and Pubs samples install only the scripts that create the samples — they will not actually install the databases onto your server — you must do that manually as outlined in the text that follows.

When the install completes, you will find a set of scripts in a directory called “SQL Server 2000 Samples” on whatever drive you chose to install on (C: by default). There will be a number of different files, but double-click on the one called Northwnd.sql — it should load it up on the SQL Server Management Console for you. From there, simply press Execute and it should install Northwind for you.

Note that, by default, the data files will install to the same directory that the master database was installed to (whatever default path you chose for data storage when you installed the server). Also note that you will need to have sysadmin rights to the server you're installing on in order to run these scripts, so, if you're running on a server at work or something like that, you will need to ask your system administrator to install Northwind for you.

Pubs

Pubs is used only in a limited fashion in this book; however, it can be a nice little playground for very simple queries. In addition to those found in this book, you will find some examples still out on the Web that utilize this sample, so it's somewhat nice to have around.

Scripted Samples

There are two samples that are created utilizing scripts that are part of the downloads for this book. You need to store each of these two databases — NorthwindBulk and NorthwindTriggers — on disk, much as you would any other file, and then simply double-click on them. This should launch the SQL Server Management Console. From there, simply execute the script.

Note that NorthwindBulk in particular generates a huge amount of data on a row-by-row basis — the result is that this script can take a fairly long time to run (from several minutes to an hour or two).

Created Samples

We also create a couple of databases from scratch as we go through the book.

Accounting

In the case of the Accounting database, we create that based on commands that are introduced in Chapter 5, and continue to build on that for a few chapters. Simply follow the scripts as they come up.

One thing to take note of is that the Accounting database, in particular, is a cumulative thing. If you skip around, you may find times where the example given in the book does not work for you because you have not done prior steps. If you run into this, all I can suggest is that you go through previous chapters and run the scripts to bring the database state up to the point you want it to be at.

TestDB

This one is pretty nebulous. Suffice to say that this database comes and goes, and may look entirely different in Chapter 13 than it did in Chapter 7. Issues here indicate that you probably need to drop this DB and re-create it step by step based on what the chapter has covered up to the point you're wanting to play with it.

Index

Index

SYMBOLS AND NUMERICS

@@CONNECTIONS, 588
@@CPU_BUSY, 588
@@CURSOR_ROWS, 319, 588–589
@@DATEFIRST, 319, 589
@@DBTS, 589–590
@@ERROR, 319, 373–378, 590
@@FETCH_STATUS, 319, 590
@@IDENTITY, 319–324, 591
@@IDLE, 591
@@IO_BUSY, 591
@@LANGID, 591
@@LANGUAGE, 591
@@LOCK_TIMEOUT, 591–592
@@MAX_CONNECTIONS, 592
@@MAX_PRECISION, 592
@@NESTLEVEL, 592
@@OPTIONS, 319, 592–593
@@PACKET_ERRORS, 594
@@PACK_RECEIVED, 593
@@PACK_SENT, 593
@@PROCID, 594
@@REMSERVER, 320, 594
@@ROWCOUNT, 320, 324–325, 594
@@SERVERNAME, 320, 594–595
@@SERVICENAME, 595
@@SPID, 595
@@TEXTSIZE, 595

@@TIMETICKS, 595
@@TOTAL_ERRORS, 595
@@TOTAL_READ, 596
@@TOTAL_WRITE, 596
@@TRANCOUNT, 320, 596
@@VERSION, 320, 596–597
1NF (first normal form), 211–214
1–9 severity codes, 382
2NF (second normal form), 214–215
3NF (third normal form), 216–218
4NF (fourth normal form), 218
5NF (fifth normal form), 218
10 severity code, 382
11–16 severity codes, 382
17 severity code, 382
18–19 severity codes, 382
20–25 severity codes, 382
1205 errors (deadlocks), 442–445

A

ABS function, 604
accessing data
 indexes, 262–263
 table scans, 262
Accounting sample database, 651, 654
ACID test for transactions, 445–446
ACOS function, 604

Activity Monitor (SQL Server Management Studio), 439–440

administration

- alerts, 562–566
- archiving data, 575
- backups, 567–570
- creating jobs and tasks, 558–559
- index maintenance, 572–575
- notification features, 556–558
- recovery models for backups, 570–571
- restoring backups, 571–572
- scheduling jobs and tasks, 556, 562–563
- SQL Server Management Objects (SMO) object model, 555

AdventureWorks sample database, 3, 5, 651–652

AdventureWorksDW sample for use with Analysis Services, 3, 5

aggregate functions

- AVG, 55, 598
- COUNT, 58–60, 598
- COUNT_BIG, 598
- defined, 597
- GROUP BY clause, 55–60
- GROUPING, 598
- MAX, 56–57, 598
- MIN, 56–57, 599
- STDEV, 599
- STDEVP, 599
- SUM, 599
- VAR, 599
- VARP, 599

alerts, 562–566

ALL **predicate** (SELECT **statement**), 66

ALLOW **settings** (CREATE INDEX **statement**), 275

ALTER DATABASE **statement**, 133–137

ALTER INDEX **statement**, 573–574

ALTER PROC **statement**, 343

ALTER **statement**, 133

ALTER TABLE **statement**, 137–140

ALTER VIEW **statement**, 301–302

alternate keys, 155, 169–170

alternative syntax for joins, 102–104

Analysis Services AdventureWorksDW sample, 3, 5

ANSI standards, 42

APP_NAME **function**, 629

architecture changes, 458–459

archiving data, 575

arguments in errors, 383–384

AS **keyword**, 453

ASC **option**, 271–272

ascending sort order for indexes, 271–272

ASCII **function**, 624

ASIN **function**, 604

assemblies

- stored procedures, 406–407
- user defined functions (UDFs), 423–424

associate table, 87

ATAN **function**, 604

ATN2 **function**, 604

atomicity, 211, 213–214, 425–426

attributes (XML document), 475, 478–479

auditing views, 306–308

authentication types

- NT Authentication, 30
- SQL Server authentication, 30–32
- Windows authentication, 30–31

AUTO **option for retrieving XML data**, 486–487

automatically deleting jobs, 556

AVG **function**, 55, 598

B

backups

- Backup Exec utility, 645
- backup set, 569
- creating backup files, 567–568
- destinations, 569
- differential backups, 569
- full backups, 568
- importance of, 567
- options, 569
- recovery models, 570–571
- restoring, 571–572
- scheduling, 570
- SQL Backup utility, 645
- transaction log, 569

backward compatibility

- DTS (Data Transformation Services), 539
- isql.exe command-line tool, 315
- osql.exe command-line tool, 315

Balanced Trees, 258–260**batches. *See also* stored procedures**

- best uses, 327
- defined, 325
- dependencies between batches, 326
- errors, 325, 327
- precedence, 328–330
- separating scripts into multiple batches, 325–326
- statements that require their own batch, 327–328

bcp (Bulk Copy Program), 39**BEGIN statement, 356–359****BEGIN TRAN statement, 426–427****beginning transactions, 426–427****Bigint datatype, 12****Binary datatype, 14****binary sort, 257****Birbeck et. al (*Professional XML*), 470****Bit datatype, 12****blocks of code, 356–359****Books Online (BOL), 20–21****Boyce-Codd form, 218****branching rules for jobs, 556****breakpoints, 401****B-Trees, 258–260****Bulk Copy Program (bcp), 39****bulk update lock (BU), 438****bulk-logged recovery model for backups, 570****Business Intelligence Studio, 518–522****business rules, 1****C****C# connection strings, 648–649****calculations**

- ABS function, 604
- ACOS function, 604
- ASIN function, 604
- ATAN function, 604
- ATN2 function, 604
- CEILING function, 604

COS function, 605**COT function, 605****DEGREES function, 605****EXP function, 605****FLOOR function, 605****LOG function, 605****LOG10 function, 606****PI function, 606****POWER function, 606****RADIANS function, 606****RAND function, 606****ROUND function, 606****SIGN function, 607****SIN function, 607****SQRT function, 607****SQUARE function, 607****TAN function, 607****callable processes, 389–390****Callstack Window (Debugger), 402****cascading, 162–163****CASE function, 629–630****CASE statement, 360–366****CAST function, 201–203, 630****CATCH blocks, 368****cdata directive, 502–503****CEILING function, 604****Char datatype, 13****CHAR function, 624****CHARINDEX function, 624****CHECK constraints, 170–171****checking the delta of an update, 455–457****checkpoints in transaction log, 428–429****choosing datatypes, 242****clustered indexes, 7–8, 263–266, 278–281****clustered tables, 263–264****cluster-key, 263****COALESCE function, 630****Codd, E.F. (IBM), 208****code**

- generating, 334–335
- grouping code into blocks, 356–359
- protecting, 308–309
- storing, 309

coding tools, 644

COLLATE option

COLLATE option

CREATE DATABASE statement, 118

CREATE TABLE statement, 125

collation order (indexes), 257–258

COLLATIONPROPERTY function, 630

COL_LENGTH function, 607–608

COL_NAME function, 607, 609

COLUMNPROPERTY function, 609

columns

datatypes, 7

domain data, 6

INCLUDE columns, 272

indexes, 281

number of, 7

rules, 6, 121–122

COLUMNS_UPDATED function, 463–465

command files for SSIS packages, 548

command object, 648

command-line tools

isql.exe, 315

osql.exe, 315, 330

SQLCMD, 330–334

COMMIT TRAN statement, 427

committing transactions, 427

compatibility of lock modes, 438

Computer Manager, 21–27

concurrency

defined, 431

triggers, 462

configurations for SSIS packages, 548

Configure Flat File dialog, 544

connection managers for SSIS packages, 548

connection object, 648

connection strings

C#, 648–649

SQL Server Management Studio, 32

VB.NET, 649–650

@@CONNECTIONS function, 588

connectivity models, 647

constraints

best uses, 182

CHECK constraints, 170–171

DEFAULT constraints, 171–173

defined, 8, 151

diagrams, 236–237

disabling, 173–174, 176–177

domain constraints, 152–153

entity constraints, 153

foreign key constraints, 158–169

ignoring bad data, 174–176

implied indexes, 277

naming, 154–155

normalization, 251–252

primary key constraints, 155–158

referential integrity constraints, 154

UNIQUE constraints, 169–170

CONTAINSTABLE function, 619

Continue option (Debugger), 400

control-of-flow statements

BEGIN, 356–359

CASE, 360–366

ELSE clause, 352–356

END, 356–359

IF . . . ELSE, 349–352

TRY/CATCH blocks, 368

WAITFOR

best uses, 367

DELAY parameter, 368

syntax, 368

TIME parameter, 368

WHILE, 366–367

conversions between datatypes, 15–17, 201

CONVERT function, 201–203, 298, 630

correlated subqueries, 190–194

COS function, 605

COT function, 605

COUNT function, 58–60, 598

COUNT_BIG function, 598

covered queries, 272

@@CPU_BUSY function, 588

CREATE ASSEMBLY statement, 406

CREATE DATABASE statement, 115–120

CREATE DEFAULT statement, 327

CREATE FUNCTION statement, 410

CREATE INDEX statement, 270–276

CREATE PROCEDURE statement, 327, 342

CREATE RULE statement, 327

CREATE statement, 115

CREATE TABLE statement, 121–133

CREATE TRIGGER statement, 328, 449–453

CREATE VIEW statement, 289–290, 301, 328

creating

assemblies, 406

backup files, 567–568

databases

with Management Studio, 142–145

with T-SQL, 115–120

indexes, 270–276

jobs, 558–561

operators, 557–558

report models, 518–525, 527

reports, 529–530

SSIS package, 540–546

stored procedures, 342

subqueries, 186–190

tables

with Management Studio, 145–148

with T-SQL, 121–133

tasks, 558–560

triggers, 449–450, 452–453

user defined functions (UDFs), 410

views

with Management Studio, 302–306

with T-SQL, 289–291

CROSS JOIN, 100–101, 104

CURRENT_TIMESTAMP function, 631

CURRENT_USER function, 631

Cursor datatype, 13

@@CURSOR_ROWS function, 319, 588–589

CURSOR_STATUS function, 600

custom error messages

creating, 385–386

triggers, 457

D

data access

indexes, 262–263

table scans, 262

data administration

alerts, 562–566

archiving data, 575

backups, 567–570

creating jobs and tasks, 558–559

index maintenance, 572–575

notification features, 556–558

recovery models for backups, 570–571

restoring backups, 571–572

scheduling jobs and tasks, 556, 562–563

SQL Server Management Objects (SMO) object model, 555

data backups

Backup Exec utility, 645

backup set, 569

creating backup files, 567–568

destinations, 569

differential backups, 569

full backups, 568

importance of, 567

options, 569

recovery models, 570–571

restoring, 571–572

scheduling, 570

SQL Backup utility, 645

transaction log, 569

Data Definition Language (DDL) triggers, 448

data integrity

constraints, 151, 181–182

triggers, 453–457

Data Manipulation Language (DML)

DML triggers, 448

Transact-SQL (T-SQL), 41

data pages, 256

data source views, 523–525

data sources

reports, 519–522

SSIS package, 541–546

data storage. See storage

Data Transformation Services (DTS), 38, 539

Database Consistency Checker (DBCC), 283–285

database name (in fully qualified names),
114–115

Database object, 2–3

database objects

defaults, 11

defined, 2

database objects (continued)

fully qualified names, 111–112

naming, 7, 17–18, 111–115

database owner (dbo) schema, 113–114

database samples

Accounting, 651, 654

AdventureWorks, 3, 5, 651–652

installing, 651–654

Northwind, 3, 6, 651–653

pubs, 3, 6, 652–653

TestDB, 651, 654

DATABASEPROPERTY function, 609–611

DATABASEPROPERTYEX function, 611

databases

archiving data, 575

backup files, 567–568

creating

with Management Studio, 142–145

with T-SQL, 115–120

de-normalization, 241

locks, 435

modifying, 133–137

normalization

Boyce-Codd form, 218

constraints, 251–252

defined, 208–209

example, 243–252

fifth normal form (5NF), 218

first normal form (1NF), 211–214

fourth normal form (4NF), 218

key candidates, 210

preparation for, 209–210

second normal form (2NF), 214–215

third normal form (3NF), 216–218

relationships

diagrams, 235–240

domain, 222

many-to-many, 223–226

one-to-one, 219–220

one-to-one or many, 221

one-to-zero, one, or many, 222–223

zero or one-to-one, 220

splitting, 128

storage, 255–256

storage tips, 242–243

DATALength function, 631

dataset object, 648

datatypes

BigInt, 12

Binary, 14

Bit, 12

Char, 13

choosing, 242

columns, 7

conversions, 15–17, 201

Cursor, 13

DateTime, 13

Decimal, 12

Float, 13

Image, 14

Int, 12

Money, 12

NChar, 14

Ntext, 14

Numeric, 12

NVarChar, 14

SmallDateTime, 13

SmallInt, 12

SmallMoney, 13

Sql_variant, 15

Table, 14

Text, 14

Timestamp/rowversion, 13

TinyInt, 12

TYPEPROPERTY function, 618

Unicode, 14

UniqueIdentifier, 13

unsigned numeric, 15

user defined, 11

VarBinary, 14

VarChar, 14

XML, 15

date

setting the first day of the week, 319, 589

truncating date from a datetime field, 354

DATEADD function, 298, 600–601

DATEDIFF function, 600–601

@@DATEFIRST function, 319, 589

DATENAME function, 601–602

DATEPART function, 601–602

DateTime datatype, 13

DAY function, 601–602

DBCC (Database Consistency Checker), 283–285

DB_ID function, 611

DB_NAME function, 611

dbo (database owner) schema, 113–114

DBREINDEX option, 285–286

db_size property, 120

@@DBTS function, 589–590

DDL (Data Definition Language) triggers, 448

deadlocks, 442–445

Debugger

breakpoints, 401

Callstack Window, 402

Continue option, 400

icons, 400–401

Locals Window, 401

Output Window, 402

Remove All Breakpoints option, 401

Restart option, 401

Run To Cursor option, 401

setup, 396–397

starting, 397

Step Into option, 400

Step Out option, 401

Step Over option, 400

Stop Debugging option, 401

Toggle Breakpoints option, 401

using, 397–400, 402–405

Watch Window, 401

yellow arrow, 400

debugging

stored procedures, 398–400

triggers, 466–468

user defined functions (UDFs), 423

Decimal datatype, 12

declaration part of an XML document, 473

DECLARE statement in scripts, 317

declaring

parameters, 344–345

variables, 317

DEFAULT constraints, 171–173

DEFAULT option (CREATE TABLE statement), 123

defaults

best uses, 182

defined, 11, 180

dropping, 180

listing, 181

deferred name resolution, 391

DEGREES function, 605

DELAY parameter (WAITFOR statement), 368

DELETE statement, 75–77, 298

DELETE triggers, 452

deleting jobs, 556

delta of an update, checking with triggers, 455–457

de-normalization, 241

dependencies between batches, 326

dependency chains, 164

deploying

report models, 528–529

reports, 537–538

derived tables, 195–197

DESC option, 271–272

descending sort order for indexes, 271–272

destinations for backups, 569

detail tables, 214–215

determinism in user defined functions (UDFs), 421–423

DevStudio IDE environment, 28

diagramming tools, 227–229

diagrams

constraints, 236–237

defined, 8

entity-relationship diagram (ERD), 8–9, 227

ERD tools, 639–644

indexes, 236

keys, 236

relationships, 235–240

tables, 230–234

DIFFERENCE function

DIFFERENCE function, 624

differential backups, 569

directives

 cdata, 502–503

 defined, 493–494

 element, 494–495

 hide, 495–498

 id, 498, 501

 idref, 498–501

 idrefs, 498–501

 xml, 495

 xmltext, 502

dirty pages, 428

dirty reads, 432–433

disabling

 constraints, 173–174, 176–177

 jobs, 566

DISTINCT predicate (SELECT statement), 64–66

DML (Data Manipulation Language)

 DML triggers, 448

 Transact-SQL (T-SQL), 41

document part of an XML document, 472

Document Type Definition (DTD), 481–482

domain constraints, 152–153

domain data (columns), 6

domain relationship, 222

domain tables, 222

drivers for NetLibs (Net-Libraries), 23

DROP INDEX statement, 281

DROP PROC statement, 344

DROP statement, 140–141

DROP TRIGGER statement, 466

DROP VIEW statement, 302

DROP_EXISTING option, 273

dropping

 custom error messages, 386

 defaults, 180

 indexes, 281

 rules, 180

 stored procedures, 344

 triggers, 466

 views, 302

DTD (Document Type Definition), 481–482

DTS (Data Transformation Services), 38, 539

durable transactions, 427

dynamic SQL, 334–335

E

editing

 databases, 133–137

 report models, 528

 SSIS packages, 550–553

 stored procedures, 343

 system tables, 4

 tables, 137–140

 views, 301, 306

18–19 severity codes, 382

element directive, 494–495

elements (XML document), 473–474, 478–479

11–16 severity codes, 382

ELSE clause, 352–356

enabling NetLibs (Net-Libraries), 24–25

encrypting views, 308–309

END statement, 356–359

entity constraints, 153

entity data (rows), 6

entity-relationship diagram (ERD), 8–9, 227

ER Studio, 644

ERD tools

 costs, 639

 ER Studio, 644

 ErWin, 643

 integration with other tools, 642–643

 logical design, 640

 macros, 642

 physical design, 640

 PowerDesigner, 644

 reverse engineering, 640–641

 scripting, 640

 synchronization, 641–642

 Visio, 644

@@ERROR function, 319, 373–378, 590

error handling

 custom error messages

 creating, 385–386

 triggers, 457

FILENAME option (CREATE DATABASE statement)

@@ERROR function, 373–378
 before errors happen, 378–381
 example, 387–388
 inline errors, 372
 manually raising errors, 381–385

errors

arguments, 383–384
 batches, 325, 327
 joins, 99
 returning error numbers, 319, 590
 severity code, 382
 state values, 383
 stored procedures, 371–372
 1205 errors (deadlocks), 442–445
 WITH LOG option, 384
 WITH NOWAIT option, 385
 WITH SETERROR option, 385

ErWin, 643

escalation of locks, 435–436

establishing precedence with batches, 328–330

ETL (Extract, Transform, and Load) tool, 540

exclusive joins, 85

exclusive locks, 436

EXEC statement, 334–339

Execute Package Utility, 547–549

EXECUTE statement, 334–339

executing SSIS packages, 547–550

exercise solutions, 577–586

EXISTS operator, 197–201

EXP function, 605

EXPLICIT option for retrieving XML data, 487–493

extended stored procedures (XPs), 393

Extensible Markup Language (XML)

defined, 469–471
 development history, 470
 documents
 attributes, 475, 478–479
 declaration part, 473
 document part, 472
 Document Type Definition (DTD), 481–482
 element content, 481
 elements, 473–474, 478–479

example, 476–481
 nodes, 474–475
 parts of (illustration), 471–472
 valid, 481–482
 well-formedness, 475–476
 XML schema, 482
 FOR XML clause, 63, 483–484
 functionality in SQL Server, 482
 hierarchy, 489–490
 namespaces, 479–480
 OPENXML function, 508–514, 620
 Report Definition Language (RDL), 537
 retrieving XML data
 AUTO option, 486–487
 directives, 494–503
 EXPLICIT option, 487–493
 PATH option, 503–508
 RAW option, 484–486
 storage, 469–470
 support for, 482

Extensible Stylesheet Language Transformations (XSLT), 514–516

extents

defined, 256
 locks, 435

Extract, Transform, and Load (ETL) tool, 540

F

factorials, 393–396

failure of transactions, 429–430

@@FETCH_STATUS function, 319, 590

fields required for views, 299

fifth normal form (5NF), 218

FILEGROUP_ID function, 612

FILEGROUP_NAME function, 612

FILEGROUPPROPERTY function, 612–613

filegroups, 8

FILEGROWTH option (CREATE DATABASE statement), 118

FILE_ID function, 612

FILE_NAME function, 612

FILENAME option (CREATE DATABASE statement), 116–117

FILEPROPERTY function

FILEPROPERTY function, 613

FILLFACTOR option, 272–273, 285–286

Filter Data dialog, 531

filtering data

in reports, 531–532

with views, 293–295

finding orphaned records, 189–190

firing order of triggers, 459–461

first day of the week, setting, 319, 589

first normal form (1NF), 211–214

FIRST triggers, 461

flattening data with views, 295–297

Float datatype, 13

FLOOR function, 605

flow control statements

BEGIN, 356–359

CASE, 360–366

ELSE clause, 352–356

END, 356–359

IF . . . ELSE, 349–352

TRY/CATCH blocks, 368

WAITFOR

best uses, 367

DELAY parameter, 368

syntax, 368

TIME parameter, 368

WHILE, 366–367

FOR ATTACH option (CREATE DATABASE statement), 118

FOR XML clause, 63, 483–484

FOR/AFTER triggers, 450–452

foreign key constraints, 158–169

FORMATMESSAGE function, 631

fourth normal form (4NF), 218

fragmentation, 282–283

FREETEXTTABLE function, 619

FROM clause (SELECT statement), 42–45

full backups, 568

FULL JOIN, 99–100

full recovery model for backups, 570

full-text catalogs, 12

FULLTEXTCATALOGPROPERTY function, 613

FULLTEXTSERVICEPROPERTY function, 614

fully qualified names

database name, 114–115

defined, 111

schema name, 112–114

server name, 114–115

functions

ABS, 604

ACOS, 604

APP_NAME, 629

ASCII, 624

ASIN, 604

ATAN, 604

ATN2, 604

AVG, 55, 598

CASE, 629–630

CAST, 201–203, 630

CEILING, 604

CHAR, 624

CHARINDEX, 624

COALESCE, 630

COLLATIONPROPERTY, 630

COL_LENGTH, 607, 609

COL_NAME, 607–608

COLUMNPROPERTY, 609

COLUMNS_UPDATED, 463–465

@@CONNECTIONS, 588

CONTAINSTABLE, 619

CONVERT, 298

CONVERT, 201–203, 630

COS, 605

COT, 605

COUNT, 58–60, 598

COUNT_BIG, 598

@@CPU_BUSY, 588

CURRENT_TIMESTAMP, 631

CURRENT_USER, 631

@@CURSOR_ROWS, 319, 588–589

CURSOR_STATUS, 600

DATABASEPROPERTY, 609–611

DATABASEPROPERTYEX, 611

DATALength, 631

DATEADD, 298, 600–601

DATEDIFF, 600–601

@@DATEFIRST, 319, 589

DATENAME, 601–602
 DATEPART, 601–602
 DAY, 601–602
 DB_ID, 611
 DB_NAME, 611
 @@DBTS, 589–590
 defined, 597
 DEGREES, 605
 DIFFERENCE, 624
 @@ERROR, 373–378, 590
 EXP, 605
 @@FETCH_STATUS, 319, 590
 FILEGROUP_ID, 612
 FILEGROUP_NAME, 612
 FILEGROUPPROPERTY, 612–613
 FILE_ID, 612
 FILE_NAME, 612
 FILEPROPERTY, 613
 FLOOR, 605
 FORMATMESSAGE, 631
 FREETEXTTABLE, 619
 FULLTEXTCATALOGPROPERTY, 613
 FULLTEXTSERVICEPROPERTY, 614
 GETANSINULL, 631
 GETDATE, 298, 601–602
 GETUTCDATE, 601–602
 GROUP BY clause, 55–60
 GROUPING, 598
 HAS_DBACCESS, 621
 HOST_ID, 632
 HOST_NAME, 632
 IDENT_CURRENT, 632
 IDENT_INCR, 632
 IDENTITY, 633
 @@IDENTITY, 319–324, 591
 IDENT_SEED, 632
 @@IDLE, 591
 INDEX_COL, 614
 INDEXKEY_PROPERTY, 614
 INDEXPROPERTY, 614–615
 @@IO_BUSY, 591
 ISDATE, 633
 IS_MEMBER, 621
 ISNULL, 194–195, 633
 ISNUMERIC, 633
 IS_SRVROLEMEMBER, 621
 @@LANGID, 591
 @@LANGUAGE, 591
 LEFT, 625
 LEN, 625
 @@LOCK_TIMEOUT, 591–592
 LOG, 605
 LOG10, 606
 LOWER, 625
 LTRIM, 625
 MAX, 56–57, 598
 @@MAX_CONNECTIONS, 592
 @@MAX_PRECISION, 592
 MIN, 56–57, 599
 MONTH, 601–602
 NCHAR, 625
 @@NESTLEVEL, 592
 NEWID, 633
 NULLIF, 634
 OBJECT_ID, 615
 OBJECT_NAME, 615
 OBJECTPROPERTY, 615–617
 OPENDATASOURCE, 619
 OPENQUERY, 619
 OPENROWSET, 619–620
 OPENXML, 508–514, 620
 @@OPTIONS, 319, 592–593
 @@PACKET_ERRORS, 594
 @@PACK_RECEIVED, 593
 @@PACK_SENT, 593
 PARSENAME, 634
 PATINDEX, 625
 PERMISSIONS, 634
 PI, 606
 POWER, 606
 @@PROCID, 594
 QUOTENAME, 626
 RADIANS, 606
 RAND, 606
 @@REMSERVER, 320, 594
 REPLACE, 626
 REPLICATE, 626
 REVERSE, 626

functions (continued)

RIGHT, 626
ROUND, 606
@@ROWCOUNT, 320, 324–325, 594
ROWCOUNT_BIG, 634
RTRIM, 627
SCOPE_IDENTITY, 634
@@SERVERNAME, 320, 594–595
SERVERPROPERTY, 635–636
@@SERVICENAME, 595
SESSIONPROPERTY, 637
SESSION_USER, 636
SIGN, 607
SIN, 607
SOUNDEX, 627
SPACE, 627
@@SPID, 595
SQL_VARIANT_PROPERTY, 617–618
SQRT, 607
SQUARE, 607
STATS_DATE, 637
STDEV, 599
STDEV, 599
STDEV, 599
STR, 627
STUFF, 627
SUBSTRING, 627–628
SUM, 599
USER_ID, 622
USER_NAME, 622
USER_SID, 622
USER_SNAME, 622
SYSTEM_USER, 637
TAN, 607
TEXTPTR, 638
@@TEXTSIZE, 595
TEXTVALID, 638
@@TIMETICKS, 595
@@TOTAL_ERRORS, 595
@@TOTAL_READ, 596
@@TOTAL_WRITE, 596
@@TRANCOUNT, 320, 596
TYPEPROPERTY, 618
UNICODE, 628
UPDATE, 463

UPPER, 628
USER, 623
user defined functions (UDFs)
 creating, 410
 debugging, 423
 defined, 10–11, 409
 determinism, 421–423
 EXECUTE statement, 339
 .NET assemblies, 423–424
 parameters, 410
 recursion, 418
 return values, 410–421
 stored procedures, 409
USER_ID, 623
USER_NAME, 637
VAR, 599
VARP, 599
@@VERSION, 320, 596–597
YEAR, 601, 603

G

generating code, 334–335

GETANSINULL **function, 631**

GETDATE **function, 298, 601–602**

GETUTCDATE **function, 602**

global variables. See system functions

Globally Unique Identifier (GUID), 125

GO **statement, 325–327**

goal of scripts, 315

granularity of locks, 435

GROUP BY **clause**

 aggregate functions, 55–60

 SELECT statement, 52–60

grouping code into blocks, 356–357, 359

GROUPING **function, 598**

GUID (Globally Unique Identifier), 125

H

handling errors

 custom error messages

 creating, 385–386

 triggers, 457

@@ERROR function, 373–378
 before errors happen, 378–381
 example, 387–388
 inline errors, 372
 manually raising errors, 381–385
HAS_DBACCESS function, 621
HAVING clause (SELECT statement), 61–63
header tables, 214–215
heaps, 264
hide directive, 495–498
hiding sensitive data, 291–293
hierarchy of XML, 489–490
hints
 optimizer hints for locks, 439
 OPTION clause, 63
HOST_ID function, 632
HOST_NAME function, 632
HTML, 470

I

icons in Debugger, 400–401
id directive, 498, 501
IDENT_CURRENT function, 632
IDENT_INCR function, 632
IDENTITY function, 633
@@IDENTITY function, 319–324, 591
IDENTITY keyword, 123
IDENTITY option (CREATE TABLE statement), 123–124
identity values, returning, 319–321, 591
IDENT_SEED function, 632
@@IDLE function, 591
idref directive, 498–501
idrefs directive, 498–501
IF...ELSE statement, 349–352
IGNORE_DUP_KEY option, 273
ignoring bad data with constraints, 174–176
Image datatype, 14
image functions, 637–638
implicit transactions, 431
implied indexes, 277
Import/Export Wizard, 540
INCLUDE columns, 272
index pages, 256
Index Tuning Wizard, 282
INDEX_COL function, 614
indexed views, 8, 10, 310–313
indexes
 ascending sort order, 271–272
 B-Trees, 258–260
 clustered, 7–8, 263–266, 278–281
 collation order, 257–258
 column order, 281
 creating, 270–276
 data access, 262–263
 defined, 7, 257
 descending sort order, 271–272
 diagrams, 236
 dropping, 281
 fragmentation, 282–283
 implied indexes, 277
 INCLUDE columns, 272
 Index Tuning Wizard, 282
 maintenance, 278, 282–285, 572–575
 non-clustered, 7–8, 263, 266–270
 page splits, 257, 260–262, 283–285
 rebuilding, 285–286
 reorganizing, 574–575
 selectivity, 277–278
 sorting options, 257
 sysindexes system table, 265
 triggers, 465
 XML indexes, 276
INDEXKEY_PROPERTY function, 614
INDEXPROPERTY function, 614–615
infinite loops, 367
inline errors, 372
INNER JOIN, 81–89, 102–103
input parameters, 344
INSERT INTO...SELECT statement, 70–72
INSERT statement, 66–70, 298
INSERT triggers, 452
installing sample databases, 651–654
INSTEAD OF triggers, 298–299, 450, 461
Int datatype, 12
integer datatypes, 12

Integration Services

Execute Package Utility, 547–549

features, 38–39

SSIS packages

command files, 548

configurations, 548

connection managers, 548

creating, 540–546

data sources, 541–546

executing, 547–550

execution options, 548

logging, 549

modifying, 550–553

reports, 548

setting values, 548

verification, 548

Integration Services Project, 550–552

integrity of data

constraints, 151, 181–182

triggers, 453–457

intent locks, 437–438

inversion keys, 155

@@IO_BUSY function, 591

ISDATE function, 633

IS_MEMBER function, 621

ISNULL function, 194–195, 633

ISNUMERIC function, 633

isolation levels for locks, 440–442

isql.exe command-line tool, 315

IS_SRVROLEMEMBER function, 621

J

jobs

alerts, 562–566

branching rules, 556

creating, 558–561

deleting, 556

disabling, 566

notification of job success or failure, 556–558

scheduling, 556, 562–563

joins

alternative syntax, 102–104

CROSS JOIN, 100–101, 104

defined, 79–81

errors, 99

exclusive, 85

FULL JOIN, 99–100

INNER JOIN, 81–89, 102–103

OUTER JOIN, 89–99, 103

subqueries, 204–205

UNION operator, 104–109

views, 295–298

K

key constraints

foreign key constraints, 158–169

primary key constraints, 155–158

UNIQUE constraints, 169–170

keys

candidates for normalization, 210

cluster-key, 263

diagrams, 236

locks, 435

keywords

AS, 453

EXISTS, 197–201

IDENTITY, 123

OUTPUT, 349

Knight et. al (*Professional SQL Server 2005*

***Integrations Services*), 547**

L

@@LANGID function, 591

@@LANGUAGE function, 591

LEFT function, 625

legacy system functions

@@CONNECTIONS, 588

@@CPU_BUSY, 588

@@CURSOR_ROWS, 319, 588–589

@@DATEFIRST, 319, 589

@@DBTS, 589–590

@@ERROR, 319, 373–378, 590

@@FETCH_STATUS, 319, 590

@@IDENTITY, 319–324, 591

@@IDLE, 591

@@IO_BUSY, 591

@@LANGID, 591

@@LANGUAGE, 591
 @@LOCK_TIMEOUT, 591–592
 @@MAX_CONNECTIONS, 592
 @@MAX_PRECISION, 592
 @@NESTLEVEL, 592
 @@OPTIONS, 319, 592–593
 @@PACKET_ERRORS, 594
 @@PACK_RECEIVED, 593
 @@PACK_SENT, 593
 @@PROCID, 594
 @@REMSERVER, 320, 594
 @@ROWCOUNT, 320, 324–325, 594
 @@SERVERNAME, 320, 594–595
 @@SERVICENAME, 595
 @@SPID, 595
 @@TEXTSIZE, 595
 @@TIMETICKS, 595
 @@TOTAL_ERRORS, 595
 @@TOTAL_READ, 596
 @@TOTAL_WRITE, 596
 @@TRANCOUNT, 320, 596
 @@VERSION, 320, 596–597
LEN function, 625
limitations of Transact-SQL (T-SQL), 341
linking table, 87
listing
 defaults, 181
 locks, 439–440
 rules, 181
Locals Window (Debugger), 401
lock manager, 431
lock modes
 bulk update lock (BU), 438
 compatibility, 438
 exclusive locks, 436
 intent locks, 437–438
 schema locks, 438
 shared locks, 436
 update locks, 437
lockable resources, 435
locks
 deadlocks, 442–445
 defined, 431
 dirty reads, 432–433

 escalation, 435–436
 granularity, 435
 isolation levels, 440–442
 listing, 439–440
 lost updates, 434–435
 non-repeatable reads, 433–434
 optimizer hints, 439
 performance, 436
 phantom reads, 434
@@LOCK_TIMEOUT function, 591–592
LOG function, 605
LOG ON option (CREATE DATABASE statement), 118
LOG10 function, 606
logging for SSIS packages, 548
logical design, 640
logins, 11
logs
 transaction log
 active portion, 429
 backup, 569
 checkpoints, 428–429
 defined, 6
 dirty pages, 428
 how it works, 428
 Windows event log, 566
lookup tables, 222
loops
 infinite loops, 367
 recursive triggers, 458
 WHILE statement, 366–367
lost updates, 434–435
LOWER function, 625
LTRIM function, 625

M

macros in ERD tools, 642
maintaining indexes, 278, 282–285, 572–575
Management Studio
 Activity Monitor, 439–440
 alerts, 562–566
 authentication types, 30–31
 Connection dialog box, 28–32

Management Studio (continued)

- connection strings, 32
- database creation, 142–145
- features, 28–38
- Index Tuning Wizard, 282
- job creation, 559–561
- job scheduling, 562–563
- Object Explorer, 38
- Properties Window, 234–235
- Query Window, 33–38
- scripting, 148–149
- table creation, 145–148
- task creation, 559–560
- View Builder, 302–306
- manually raising errors, 381–385**
- many-to-many relationships, 223–226**
- master system database, 3–4**
- mathematical functions**
 - ABS, 604
 - ACOS, 604
 - ASIN, 604
 - ATAN, 604
 - ATN2, 604
 - CEILING, 604
 - COS, 605
 - COT, 605
 - DEGREES, 605
 - EXP, 605
 - FLOOR, 605
 - LOG, 605
 - LOG10, 606
 - PI, 606
 - POWER, 606
 - RADIANS, 606
 - RAND, 606
 - ROUND, 606
 - SIGN, 607
 - SIN, 607
 - SQRT, 607
 - SQUARE, 607
 - TAN, 607
 - uses, 603
- MAX function, 56–57, 598**
- @@MAX_CONNECTIONS function, 592**

- MAXDROP option, 275**
- @@MAX_PRECISION function, 592**
- MAXSIZE option (CREATE DATABASE statement), 117**
- memory, 22, 24, 26**
- merge table, 87**
- message ID, 381–382**
- message string, 381–382**
- metadata**
 - tables, 6
 - views, 310
- metadata functions**
 - COL_LENGTH, 607–608
 - COL_NAME, 607, 609
 - COLUMNPROPERTY, 609
 - DATABASEPROPERTY, 609–611
 - DATABASEPROPERTYEX, 611
 - DB_ID, 611
 - DB_NAME, 611
 - FILEGROUP_ID, 612
 - FILEGROUP_NAME, 612
 - FILEGROUPPROPERTY, 612–613
 - FILE_ID, 612
 - FILE_NAME, 612
 - FILEPROPERTY, 613
 - FULLTEXTCATALOGPROPERTY, 613
 - FULLTEXTSERVICEPROPERTY, 614
 - INDEX_COL, 614
 - INDEXKEY_PROPERTY, 614
 - INDEXPROPERTY, 614–615
 - OBJECT_ID, 615
 - OBJECT_NAME, 615
 - OBJECTPROPERTY, 615–617
 - SQL_VARIANT_PROPERTY, 617–618
 - TYPEPROPERTY, 618
 - uses, 607
- MIN function, 56–57, 599**
- mixing versions of SQL Server, 3**
- model system database, 3–4**
- modifying**
 - databases, 133–137
 - report models, 528
 - SSIS packages, 550–553

stored procedures, 343
system tables, 4
tables, 137–140
views, 301, 306

Money **datatype**, 12

MONTH **function**, 601–602

msdb **system database**, 3–4

N

NAME **option** (CREATE DATABASE **statement**), 116

name resolution, 391

Named Pipes NetLib, 22, 25

names

constraints, 154–155
fully qualified names
 database name, 114–115
 defined, 111
 schema name, 112–114
 server name, 114–115
objects, 7, 17–18, 111–115
report models, 526–527

namespaces (XML), 479–480

NChar **datatype**, 14

NCHAR **function**, 625

nesting

blocks of code, 357
stored procedures, 389–390
triggers, 458

nesting queries, 186–190

@@NESTLEVEL **function**, 592

.NET assemblies

stored procedures, 406–407
user defined functions (UDFs), 423–424

NetLibs (Net-Libraries)

drivers, 23
enabling, 24–25
Named Pipes, 22, 25
Shared Memory, 22, 24, 26
TCP/IP, 22–23, 25
VIA, 22

network configuration, 21–22

NEWID **function**, 633

nodes (XML documents), 474–475

non-clustered indexes, 7–8, 263, 266–270

non-repeatable reads, 433–434

normalization

Boyce-Codd form, 218
constraints, 251–252
defined, 208–209
example, 243–252
fifth normal form (5NF), 218
first normal form (1NF), 211–214
fourth normal form (4NF), 218
key candidates, 210
preparation for, 209–210
second normal form (2NF), 214–215
third normal form (3NF), 216–218

Northwind sample database, 3, 6, 651–653

NOT FOR REPLICATION **option**

CREATE TABLE **statement**, 124

CREATE TRIGGER **statement**, 453

notification features, 556–558

NT Authentication, 30

Ntext **datatype**, 14

NULL **data**, 17

NULLIF **function**, 634

NULL/NOT NULL **option** (CREATE TABLE **statement**), 125

Numeric **datatype**, 12

NVarChar **datatype**, 14

O

Object Explorer (SQL Server Management Studio), 38

OBJECT_ID **function**, 615

OBJECT_NAME **function**, 615

OBJECTPROPERTY **function**, 615–617

objects

Database, 2–3
defaults, 11
defined, 2
fully qualified names, 111–112
naming, 7, 17–18, 111–115

OLAP (online analytical processing), 207

OLTP (online transaction-processing), 207

ON option

- CREATE DATABASE statement, 116
- CREATE INDEX statement, 275–276
- CREATE TABLE statement, 127
- CREATE TRIGGER statement, 449
- 1–9 severity codes, 382**
- one-to-one or many relationship, 221**
- one-to-one relationship, 219–220**
- one-to-zero, one, or many relationship, 222–223**
- online analytical processing (OLAP), 207**
- ONLINE option, 275**
- online transaction-processing (OLTP), 207**
- OPENDATASOURCE function, 619
- OPENQUERY function, 619
- OPENROWSET function, 619–620
- OPENXML function, 508–514, 620
- operators**
 - EXISTS, 197–201
 - UNION, 104–109
- operators (for data administration), 557–558**
- optimizer hints for locks, 439**
- OPTION clause (SELECT statement), 63
- optional parameters, 346**
- @@OPTIONS function, 319, 592–593
- options, returning information about, 319, 592–593**
- ORDER BY clause (SELECT statement), 49–52
- orphaned records, 189–190**
- osql.exe command-line tool, 315, 330
- OUTER JOIN, 89–99, 103
- OUTPUT keyword, 349
- output parameters, 344, 346–349**
- Output Window (Debugger), 402**
- ownership, 112**

P

- @@PACKET_ERRORS function, 594
- @@PACK_RECEIVED function, 593
- @@PACK_SENT function, 593
- PAD_INDEX option, 272
- page splits, 257, 260–262, 283–285**
- page types, 256**

pages

- defined, 256
- dirty pages, 428
- locks, 435

parameters

- SQLCMD command-line tool, 334
- stored procedures
 - datatype, 344
 - declaring, 344–345
 - default, 345–346
 - input parameters, 344
 - name, 344
 - optional parameters, 346
 - OUTPUT keyword, 349
 - output parameters, 344, 346–349
 - required parameters, 345
 - values, 345–346
- triggers, 448–449
- user defined functions (UDFs), 410
- WAITFOR statement, 368

PARSENAME function, 634

PATH option for retrieving XML data, 503–507

PATINDEX function, 625

performance

- locks, 436
- queries, 204
- splitting databases, 128
- stored procedures, 391–392
- triggers, 461–465

PERMISSIONS function, 634

phantoms, 434

physical design, 640

PI function, 606

POWER function, 606

PowerDesigner, 644

precedence, establishing with batches, 328–330

preparation for normalization, 209–210

primary filegroup, 8

primary key constraints, 155–157

@@PROCID function, 594

Professional SQL Server 2005 Integrations

- Services (Knight et. al), 547

Professional XML (Birbeck et. al), 470

Profiler, 40**properties**

db_size, 120

Properties Window (SQL Server Management Studio), 234–235

protecting source code, 308–309pubs **sample database, 3, 6, 652–653****Q****queries**

covered queries, 272

derived tables, 195–197

EXISTS operator, 197–201

nesting, 186–190

performance, 204

SQLCMD command-line tool, 331

subqueries

correlated, 190–194

joins, 204–205

nested, 186–190

Query Window (SQL Server Management Studio), 33–38QUOTENAME **function, 626****R**RADIANS **function, 606****raising errors manually, 381–385**RAND **function, 606**RAW **option for retrieving XML data, 484–486****RDBMS (Relational Database Management System)**

business rules, 1

features, 1

objects, 2–3

RDL (Report Definition Language), 537READ COMMITTED **isolation level, 440–441**READ UNCOMMITTED **isolation level, 441****rebuilding indexes, 285–286****records**

orphaned records, 189–190

phantoms, 434

recovery models for backups, 570–571**recovery of transactions, 429–430****recursion**

stored procedures, 393–396

triggers, 458

user defined functions (UDFs), 418

reference materials (Books Online), 20–21**referential integrity constraints, 154****Relational Database Management System (RDBMS)**

business rules, 1

features, 1

objects, 2–3

relationships

diagrams, 235–240

domain, 222

many-to-many, 223–226

one-to-one, 219–220

one-to-one or many, 221

one-to-zero, one, or many, 222–223

zero or one-to-one, 220

@@REMSERVER **function, 320, 594****reorganizing indexes, 574–575**REPEATABLE READ **isolation level, 442**REPLACE **function, 626**REPLICATE **function, 626****Report Builder, 530–531****Report Definition Language (RDL), 537****report models**

benefits of using, 532

creating, 518–525, 527

data source views, 523–525

data sources, 519–522

deploying, 528–529

modifying, 528

naming, 526–527

report model generation rules, 525–526

statistics, 526

Report Server Projects, 532–538**Reporting Services, 518****reporting user interface, 529–530****reports**

creating, 529–530

deploying, 537–538

filtering data, 531–532

reports (continued)

- right amount of data, 517
- running, 530–531
- sort order, 532
- SSIS packages, 548
- visual appeal, 517

required fields for views, 299

required parameters, 345

Restart option (Debugger), 401

restoring backups, 571–572

restrictions. *See* constraints

retrieving XML data

- AUTO option, 486–487
- EXPLICIT option, 487–493
- FOR XML clause, 63, 483–484
- PATH option, 503–508
- RAW option, 484–486

RETURN statement, 369–371

return values

- stored procedures, 369–371
- triggers, 448–449
- user defined functions (UDFs)
 - scalar values, 410–413
 - tables, 414–421

returning

- error numbers, 319, 590
- identity values, 319–321, 591
- name of the local server, 320, 594–595
- number of active transactions, 320, 596
- number of rows affected by last statement, 320, 324–325, 594
- option information, 319, 592–593
- SQL Server information, 320
- tables
 - with stored procedures, 342–343
 - with user defined functions (UDFs), 414–421
- value of server called in stored procedures, 320, 594

reverse engineering, 640–641

REVERSE function, 626

RID (row identifier) locks, 435

RIGHT function, 626

roles, 11

ROLLBACK TRAN statement, 427

rolling back transactions, 427, 465

root node (XML document), 474–475

ROUND function, 606

row identifier (RID) locks, 435

@@ROWCOUNT function, 320, 324–325, 594

ROWCOUNT_BIG function, 634

ROWGUIDCOL option (CREATE TABLE statement), 124–125

rows

- entity data, 6
- locks, 435
- number of, 7
- returning number of rows affected by last statement, 320, 324–325, 594
- rules for identifiers, 121–122
- storage, 257

rowset functions

- CONTAINSTABLE, 619
- FREETEXTTABLE, 619
- OPENDATASOURCE, 619
- OPENQUERY, 619
- OPENROWSET, 619–620
- OPENXML, 508–514, 620
- uses, 618

RTRIM function, 627

rules

- best uses, 182
- branching rules for jobs, 556
- business rules, 1
- columns, 6
- defined, 11, 178–179
- dropping, 180
- identifiers, 121–122
- listing, 181
- naming rules, 18
- report model generation rules, 525–526
- syntax
 - ALTER statement, 133
 - BEGIN TRAN statement, 427
 - COMMIT TRAN statement, 427
 - CREATE FUNCTION statement, 410
 - CREATE INDEX statement, 270
 - CREATE statement, 115
 - CREATE TABLE statement, 121

CREATE TRIGGER statement, 449
 CREATE VIEW statement, 289–290
 DELETE statement, 75–77
 DROP statement, 141
 DROP TRIGGER statement, 466
 IF . . . ELSE statements, 349
 INSERT statement, 66–70
 ROLLBACK TRAN statement, 427
 SAVE TRAN statement, 427
 SELECT statement, 42
 SQLCMD command-line tool, 330–331
 stored procedures, 342
 UPDATE statement, 72–75
 WAITFOR statement, 368

Run To Cursor **option (Debugger)**, 401

running

reports, 530–531
 scripts, 334–335

runtime errors in batches, 327

S

sample databases

Accounting, 651, 654
 AdventureWorks, 3, 5, 651–652
 installing, 651–654
 Northwind, 3, 6, 651–653
 pubs, 3, 6, 652–653
 TestDB, 651, 654

SAVE TRAN **statement**, 427

saving transactions, 427

scalar values in user defined functions (UDFs),
 410–413

scheduling

backups, 570
 jobs and tasks, 556, 562–563

schema binding, 309

schema locks, 438

SCHEMABINDING **option**, 309

schemas

dbo (database owner), 113–114
 defined, 112–113
 names (in fully qualified names), 112–114

Sch-M (schema modification lock), 438

Sch-S (schema stability lock), 438

SCOPE_IDENTITY **function**, 634

scripting

ERD tools, 640
 Management Studio, 148–149

scripts

batches
 best uses, 327
 dependencies, 326
 errors, 327
 precedence, 328–330
 statements that require their own batch,
 327–328
 DECLARE statement, 317
 examples, 316
 goal, 315
 @@IDENTITY function, 322–324
 @@ROWCOUNT function, 324–325
 running, 334–335
 separating into multiple batches, 325–326
 SQLCMD command-line tool, 332–333
 storage, 315
 USE statement, 316
 uses, 315

searched CASE statement, 360, 362–366

second normal form (2NF), 214–215

secondary filegroups, 8

secondary files, 8

security

EXEC statement, 338
 stored procedures, 390
 TRUSTWORTHY option (CREATE DATABASE
 statement), 119

security functions

HAS_DBACCESS, 621
 IS_MEMBER, 621
 IS_SRVROLEMEMBER, 621
 SUSER_ID, 622
 SUSER_NAME, 622
 SUSER_SID, 622
 SUSER_SNAME, 622
 USER, 623
 USER_ID, 623
 uses, 620

SELECT statement

SELECT statement

- ALL predicate, 66
- FROM clause, 42–49
- DISTINCT predicate, 64–66
- FOR XML clause, 63
- GROUP BY clause, 52–60
- HAVING clause, 61–63
- nested queries, 187–190
- OPTION clause, 63
- ORDER BY clause, 49–52
- setting values for variables, 317–318
- syntax rules, 42
- WHERE clause, 45–49
- selectivity in indexes, 277–278**
- self-referencing tables, 160–161**
- separating scripts into multiple batches, 325–326**
- SERIALIZABLE isolation level, 442**
- Server Management Objects (SMO) object model, 555**
- @@SERVERNAME function, 320, 594–595**
- SERVERPROPERTY function, 635–636**
- servers**
 - returning the name of the local server, 320, 594–595
 - server name (in fully qualified names), 114–115
- service management, 21–22**
- @@SERVICENAME function, 595**
- SESSIONPROPERTY function, 637**
- SESSION_USER function, 636**
- SET statement, 317–318**
- setting**
 - firing order of triggers, 460–461
 - first day of the week, 319, 589
 - isolation levels for locks, 440–442
 - values for SSIS packages, 548
 - values for variables, 317–318
- 17 severity code, 382**
- severity codes, 382**
- SGML, 470**
- shared locks, 436**
- Shared Memory NetLib, 22, 24, 26**
- SIGN function, 607**
- simple recovery model for backups, 571**

- SIN function, 607**
- SIZE option (CREATE DATABASE statement), 117**
- SmallDateTime datatype, 13**
- SmallInt datatype, 12**
- SmallMoney datatype, 13**
- SMO (Server Management Objects) object model, 555**
- solutions to exercises, 577–586**
- Sort and Group dialog, 532**
- sorting options**
 - indexes, 257
 - reports, 532
- SORT_IN_TEMPDB option, 274**
- SOUNDEX function, 627**
- source code**
 - generating, 334–335
 - grouping code into blocks, 356–359
 - protecting, 308–309
 - storing, 309
- SPACE function, 627**
- sp_addmessage stored procedure, 385–386**
- sp_dboption stored procedure, 458**
- sp_depends stored procedure, 181**
- sp_dropmessage stored procedure, 386**
- sp_helptext stored procedure, 306–307**
- @@SPID function, 595**
- splitting databases, 128**
- sprocs. See stored procedures**
- sp_settriggerorder stored procedure, 460**
- SQL Backup utility, 645**
- SQL Compare, 644**
- SQL Editor, 644**
- SQL scripts. See scripts**
- SQL Server**
 - mixing with SQL Server 2000 and 2005, 3
 - returning information about, 320, 596–597
 - XML support, 482
- SQL Server authentication, 30–32**
- SQL Server Books Online, 20–21**
- SQL Server Business Intelligence Studio, 518–522**
- SQL Server Computer Manager, 21–27**

SQL Server Integration Services (SSIS)

Execute Package Utility, 547–549

features, 38–39

SSIS packages

command files, 548

configurations, 548

connection managers, 548

creating, 540–546

data sources, 541–546

executing, 547–550

execution options, 548

logging, 548

modifying, 550–553

reports, 548

setting values, 548

verification, 548

SQL Server Management Objects (SMO) object model, 555**SQL Server Management Studio**

Activity Monitor, 439–440

alerts, 562–566

authentication types, 30–31

Connection dialog box, 28–32

connection strings, 32

database creation, 142–145

features, 28–38

Index Tuning Wizard, 282

job creation, 559–561

job scheduling, 562–563

Object Explorer, 38

Properties Window, 234–235

Query Window, 33–38

scripting, 148–149

table creation, 145–148

task creation, 559–560

View Builder, 302–306

SQL Server Profiler, 40**SQL Server storage. See storage****SQL (Structured Query Language), 41**SQLCMD **command-line tool**

parameters, 334

queries, 331

scripts, 332–333

syntax, 330–331

uses, 330

Sql_variant **datatype, 15**SQL_VARIANT_PROPERTY **function, 617–618**SQRT **function, 607**SQUARE **function, 607****SSIS (SQL Server Integration Services)**

Execute Package Utility, 547–549

features, 38–39

SSIS packages

command files, 548

configurations, 548

connection managers, 548

creating, 540–546

data sources, 541–546

executing, 547–550

execution options, 548

logging, 548

modifying, 550–553

reports, 548

setting values, 548

verification, 548

starting the Debugger, 397**state values, 383****statements. See statements by specific statement names****statistics, 526**STATISTICS_NORECOMPUTE **option, 274**STATS_DATE **function, 637**STDEV **function, 599**STDEVP **function, 599**Step Into **option (Debugger), 400**Step Out **option (Debugger), 401**Step Over **option (Debugger), 400**Stop Debugging **option (Debugger), 401****storage**

databases, 255–256

extents, 256

page splits, 257

pages, 256

rows, 257

scripts, 315

source code, 309

storage (continued)

tips, 242–243

XML, 469–470

stored procedures. See also batches; triggers

callable processes, 389–390

creating, 342

debugging, 398–399

deferred name resolution, 391

defined, 10

dropping, 344

@@ERROR function, 374–378

errors, 371–372

examples, 342–343

extended stored procedures (XPs), 393

modifying, 343

nesting, 389–390

parameters

datatype, 344

declaring, 344–345

default, 345–346

input parameters, 344

name, 344

optional parameters, 346

OUTPUT keyword, 349

output parameters, 344, 346–349

required parameters, 345

values, 345–346

performance, 391–392

recursion, 393–396

return values, 369–371

returning value of server called, 320, 594

security, 390

sp_addmessage, 385–386

sp_dboption, 458

sp_depends, 181

sp_dropmessage, 386

sp_helptext, 306–307

sp_settriggerorder, 460

syntax, 342

user defined functions (UDFs), 409

WITH RECOMPILE option, 392–393

STR function, 627

string functions

ASCII, 624

CHAR, 624

CHARINDEX, 624

DIFFERENCE, 624

LEFT, 625

LEN, 625

LOWER, 625

LTRIM, 625

NCHAR, 625

PATINDEX, 625

QUOTENAME, 626

REPLACE, 626

REPLICATE, 626

REVERSE, 626

RIGHT, 626

RTRIM, 627

SOUNDEX, 627

SPACE, 627

STR, 627

STUFF, 627

SUBSTRING, 627–628

UNICODE, 628

UPPER, 628

uses, 623

Structured Query Language (SQL), 41

STUFF function, 627

subqueries

correlated, 190–194

joins, 204–205

nested, 186–190

SUBSTRING function, 627–628

SUM function, 599

support for XML, 482

SUSER_ID function, 622

SUSER_NAME function, 622

SUSER_SID function, 622

SUSER_SNAME function, 622

synchronization with ERD tools, 641–642

syntax

ALTER statement, 133

BEGIN TRAN statement, 427

COMMIT TRAN statement, 427

CREATE FUNCTION statement, 410

CREATE INDEX statement, 270

CREATE statement, 115

CREATE TABLE statement, 121

CREATE TRIGGER statement, 449

- CREATE VIEW statement, 289–290
- DELETE statement, 75–77
- DROP statement, 141
- DROP TRIGGER statement, 466
- IF . . . ELSE statements, 349
- INSERT statement, 66–70
- ROLLBACK TRAN statement, 427
- SAVE TRAN statement, 427
- SELECT statement, 42
- SQLCMD command-line tool, 330–331
- stored procedures, 342
- UPDATE statement, 72–75
- WAITFOR statement, 368
- syntax errors in batches, 327**
- syscomments **system table, 307**
- sysindexes **system table, 265**
- sysobjects **system table, 307**
- system databases, 3–5**
- system functions**
 - APP_NAME, 629
 - CASE, 629–630
 - CAST, 630
 - COALESCE, 630
 - COLLATIONPROPERTY, 630
 - @@CONNECTIONS, 588
 - CONVERT, 630
 - @@CPU_BUSY, 588
 - CURRENT_TIMESTAMP, 631
 - CURRENT_USER, 631
 - @@CURSOR_ROWS, 319, 588–589
 - DATALength, 631
 - @@DATEFIRST, 319, 589
 - @@DBTS, 589–590
 - @@ERROR, 319, 373–378, 590
 - @@FETCH_STATUS, 319, 590
 - FORMATMESSAGE, 631
 - GETANSINULL, 631
 - HOST_ID, 632
 - HOST_NAME, 632
 - IDENT_CURRENT, 632
 - IDENT_INCR, 632
 - IDENTITY, 633
 - @@IDENTITY, 319–324, 591
 - IDENT_SEED, 632
 - @@IDLE, 591
 - @@IO_BUSY, 591
 - ISDATE, 633
 - ISNULL, 633
 - ISNUMERIC, 633
 - @@LANGID, 591
 - @@LANGUAGE, 591
 - @@LOCK_TIMEOUT, 591–592
 - @@MAX_CONNECTIONS, 592
 - @@MAX_PRECISION, 592
 - @@NESTLEVEL, 592
 - NEWID, 633
 - NULLIF, 634
 - @@OPTIONS, 319, 592–593
 - @@PACKET_ERRORS, 594
 - @@PACK_RECEIVED, 593
 - @@PACK_SENT, 593
 - PARSENAME, 634
 - PERMISSIONS, 634
 - @@PROCID, 594
 - @@REMSERVER, 320, 594
 - @@ROWCOUNT, 320, 324–325, 594
 - ROWCOUNT_BIG, 634
 - SCOPE_IDENTITY, 634
 - @@SERVERNAME, 320, 594–595
 - SERVERPROPERTY, 635–636
 - @@SERVICENAME, 595
 - SESSIONPROPERTY, 637
 - SESSION_USER, 636
 - @@SPID, 595
 - STATS_DATE, 637
 - SYSTEM_USER, 637
 - @@TEXTSIZE, 595
 - @@TIMETICKS, 595
 - @@TOTAL_ERRORS, 595
 - @@TOTAL_READ, 596
 - @@TOTAL_WRITE, 596
 - @@TRANCOUNT, 320, 596
 - USER_NAME, 637
 - uses, 628
 - @@VERSION, 320, 596–597
- system tables**
 - modifying, 4
 - syscomments, 307

system tables (continued)

sysindexes, 265

sysobjects, 307

SYSTEM_USER function, 637

T

Table datatype, 14

table scans, 262

tables

associate table, 87

clustered, 263–264

columns

datatypes, 7

number of, 7

rules, 6

constraints

best uses, 182

CHECK constraints, 170–171

DEFAULT constraints, 171–173

defined, 8, 151

diagrams, 236–237

disabling, 173–174, 176–177

domain constraints, 152–153

entity constraints, 153

foreign key constraints, 158–169

ignoring bad data, 174–176

implied indexes, 277

naming, 154–155

normalization, 251–252

primary key constraints, 155–158

referential integrity constraints, 154

UNIQUE constraints, 169–170

creating

with Management Studio, 145–148

with T-SQL, 121–133

defined, 6

derived tables, 195–197

detail tables, 214–215

diagrams, 230–234

domain data (columns), 6

domain tables, 222

entity data (rows), 6

example, 7

GUID (Globally Unique Identifier), 125

header tables, 214–215

heaps, 264

joins

alternative syntax, 102–104

CROSS JOIN, 100–101, 104

defined, 79–81

errors, 99

exclusive, 85

FULL JOIN, 99–100

INNER JOIN, 81–89, 102–103

OUTER JOIN, 89–99, 103

subqueries, 204–205

UNION operator, 104–109

linking table, 87

locks, 435

lookup tables, 222

merge table, 87

metadata, 6

modifying, 137–140

reports, 530

returning

with stored procedures, 342–343

with user defined functions (UDFs), 414–421

rows, number of, 7

self-referencing, 160–161

system tables

modifying, 4

syscomments, 307

sysindexes, 265

sysobjects, 307

triggers

architecture changes, 458–459

best uses, 182

checking the delta of an update, 455–457

common uses, 448, 457

concurrency, 462

creating, 449–450, 452–453

custom error messages, 457

data integrity, 181, 453–457

DDL (Data Definition Language) triggers, 448

debugging, 466–468

defined, 8, 448

DELETE triggers, 452

- DML (Data Manipulation Language) triggers, 448
- dropping, 466
- firing order, 459–461
- FIRST triggers, 461
- FOR/AFTER triggers, 450–452
- indexes, 465
- INSERT triggers, 452
- INSTEAD OF triggers, 298–299, 450, 461
- nesting, 458
- parameters, 448–449
- performance, 461–465
- recursion, 458
- return values, 448–449
- transaction rollbacks, 465
- turning on/off, 459
- UPDATE triggers, 452, 463–465
- TAN function, 607**
- tasks**
 - creating, 558–560
 - scheduling, 556, 562–563
- TCP/IP NetLib, 22–23, 25**
- tempdb **system database, 3, 5**
- 10 severity code, 382**
- TestDB sample database, 651, 654**
- Text **datatype, 14**
- text functions, 637–638**
- TEXTIMAGE_ON **option (CREATE TABLE statement), 127**
- TEXTPTR **function, 638**
- @@TEXTSIZE **function, 595**
- TEXTVALID **function, 638**
- third normal form (3NF), 216–218**
- time**
 - DateTime datatype, 13
 - SmallDateTime datatype, 13
 - truncating time from a datetime field, 354
 - UTC (Universal Time Coordinate) time, 602
- TIME **parameter (WAITFOR statement), 368**
- Timestamp/rowversion **datatype, 13**
- @@TIMETICKS **function, 595**
- TinyInt **datatype, 12**
- TOAD, 644**
- Toggle Breakpoints **option (Debugger), 401**
- tools and utilities**
 - backup utilities, 645
 - Books Online (BOL), 20–21
 - Bulk Copy Program (bcp), 39
 - coding tools, 644
 - Computer Manager, 21–27
 - Database Consistency Checker (DBCC), 283–285
 - Debugger, 396–401
 - diagramming tools, 227–229
 - ERD tools, 639–644
 - Execute Package Utility, 547–549
 - Extract, Transform, and Load (ETL) tool, 540
 - Index Tuning Wizard, 282
 - isql.exe command-line tool, 315
 - Activity Monitor, 439–440
 - alerts, 562–566
 - authentication types, 30–31
 - Connection dialog box, 28–32
 - connection strings, 32
 - database creation, 142–145
 - features, 28–38
 - Index Tuning Wizard, 282
 - job creation, 559–561
 - job scheduling, 562–563
 - Object Explorer, 38
 - Properties Window, 234–235
 - Query Window, 33–38
 - scripting, 148–149
 - table creation, 145–148
 - task creation, 559–560
 - View Builder, 302–306
 - osql.exe command-line tool, 315
 - Profiler, 40
 - SQL Server Integration Services (SSIS), 38–39
 - SQLCMD command-line tool, 330–334
- @@TOTAL_ERRORS **function, 595**
- @@TOTAL_READ **function, 596**
- @@TOTAL_WRITE **function, 596**
- @@TRANCOUNT **function, 320, 596**
- transaction log**
 - active portion, 429
 - backup, 569
 - checkpoints, 428–429

transaction log (continued)

- defined, 6
- dirty pages, 428
- how it works, 428

transactions

- ACID test, 445–446
- atomicity, 425–426
- beginning, 426–427
- committing, 427
- deadlocks, 442–445
- dirty reads, 432–433
- durable transactions, 427
- failure, 429–430
- implicit, 431
- lost updates, 434–435
- non-repeatable reads, 433–434
- phantom reads, 434
- recovery, 429–430
- returning the number of active transactions, 320, 596
- rolling back, 427, 465
- saving, 427

Transact-SQL (T-SQL)

- ALTER DATABASE statement, 133–137
- ALTER INDEX statement, 573–574
- ALTER PROC statement, 343
- ALTER statement, 133
- ALTER TABLE statement, 137–140
- ALTER VIEW statement, 301–302
- batches
 - best uses, 327
 - defined, 325
 - dependencies between batches, 326
 - errors, 325, 327
 - precedence, 328–330
 - separating scripts into multiple batches, 325–326
 - statements that require their own batch, 327–328
- BEGIN statement, 356–359
- BEGIN TRAN statement, 426–427
- CASE statement, 360–366
- COMMIT TRAN statement, 427
- compliance to standards, 42

- CREATE ASSEMBLY statement, 406
- CREATE DATABASE statement, 115–120
- CREATE DEFAULT statement, 327
- CREATE FUNCTION statement, 410
- CREATE INDEX statement, 270–276
- CREATE PROCEDURE statement, 327, 342
- CREATE RULE statement, 327
- CREATE statement, 115
- CREATE TABLE statement, 121–133
- CREATE TRIGGER statement, 328, 449–453
- CREATE VIEW statement, 289–290, 301, 328
- Data Manipulation Language (DML), 41
- DECLARE statement, 317
- defined, 41
- DELETE statement, 75–77, 298
- DROP INDEX statement, 281
- DROP PROC statement, 344
- DROP statement, 140–141
- DROP TRIGGER statement, 466
- DROP VIEW statement, 302
- END statement, 356–359
- IF . . . ELSE statement, 349–352
- INSERT INTO . . . SELECT statement, 70–72
- INSERT statement, 66–70, 298
- limitations, 341
- new programming constructs, 41
- RETURN statement, 369–371
- ROLLBACK TRAN statement, 427
- SAVE TRAN statement, 427
- SELECT statement
 - ALL predicate, 66
 - DISTINCT predicate, 64–66
 - FOR XML clause, 63
 - FROM clause, 42–45
 - GROUP BY clause, 52–60
 - HAVING clause, 61–63
 - nested queries, 187–190
 - OPTION clause, 63
 - ORDER BY clause, 49–52
 - setting values for variables, 317–318
 - syntax rules, 42
 - WHERE clause, 45–49
- SET statement, 317–318
- TRY/CATCH blocks, 368

UPDATE statement, 72–75, 298

USE statement, 316

WAITFOR statement, 367–368

WHILE statement, 366–367

transformations

Data Transformation Services (DTS), 38, 539

defined, 39

XSLT (Extensible Stylesheet Language Transformations), 514–516

triggers

architecture changes, 458–459

best uses, 182

checking the delta of an update, 455–457

common uses, 448, 457

concurrency, 462

creating, 449–450, 452–453

custom error messages, 457

data integrity, 181, 453–457

DDL (Data Definition Language) triggers, 448

debugging, 466–468

defined, 8, 448

DELETE triggers, 452

DML (Data Manipulation Language) triggers, 448

dropping, 466

firing order, 459–461

FIRST triggers, 461

FOR/AFTER triggers, 450–452

indexes, 465

INSERT triggers, 452

INSTEAD OF triggers, 298–299, 450, 461

nesting, 458

parameters, 448–449

performance, 461–465

recursion, 458

return values, 448–449

transaction rollbacks, 465

turning on/off, 459

UPDATE triggers, 452, 463–465

truncating date or time from a datetime field, 354

TRUSTWORTHY option (CREATE DATABASE statement), 119

TRY/CATCH blocks, 368

T-SQL (Transact-SQL). See Transact-SQL (T-SQL)

turning on/off triggers, 459

1205 errors (deadlocks), 442–445

20–25 severity codes, 382

TYPEPROPERTY function, 618

types of authentication

NT Authentication, 30

SQL Server authentication, 30–32

Windows authentication, 30–31

types of data. See datatypes

U

UDFs (user defined functions)

creating, 410

debugging, 423

defined, 10–11, 409

determinism, 421–423

EXECUTE statement, 339

.NET assemblies, 423–424

parameters, 410

recursion, 418

return values

scalar values, 410–413

tables, 414–421

stored procedures, 409

Ultra Edit, 644

Unicode datatypes, 14

UNICODE function, 628

Uniform Resource Identifier (URI), 479

UNION operator, 104–109

UNIQUE constraints, 169–170

UniqueIdentifier datatype, 13

Universal Time Coordinate (UTC) time, 602

unsigned numeric datatypes, 15

UPDATE function, 463

update locks, 437

UPDATE statement, 72–75, 298

Update Statistics dialog, 526

updates

checking the delta of an update, 455–457

lost updates, 434–435

UPDATE triggers, 452, 463–465

UPPER function, 628

URI (Uniform Resource Identifier), 479

USE statement

USE statement, 316

user defined datatypes, 11

user defined functions (UDFs)

creating, 410

debugging, 423

defined, 10–11, 409

determinism, 421–423

EXECUTE statement, 339

.NET assemblies, 423–424

parameters, 410

recursion, 418

return values

scalar values, 410–413

tables, 414–421

stored procedures, 409

USER function, 623

USER_ID function, 623

USER_NAME function, 637

users

concurrency, 431

defined, 11

UTC (Universal Time Coordinate) time, 602

utilities and tools

backup utilities, 645

Books Online (BOL), 20–21

Bulk Copy Program (bcp), 39

coding tools, 644

Computer Manager, 21–27

Database Consistency Checker (DBCC), 283–285

Debugger, 396–401

diagramming tools, 227–229

ERD tools, 639–644

Execute Package Utility, 547–549

Extract, Transform, and Load (ETL) tool, 540

Index Tuning Wizard, 282

isql.exe command-line tool, 315

Management Studio

Activity Monitor, 439–440

alerts, 562–566

authentication types, 30–31

Connection dialog box, 28–32

connection strings, 32

database creation, 142–145

features, 28–38

Index Tuning Wizard, 282

job creation, 559–561

job scheduling, 562–563

Object Explorer, 38

Properties Window, 234–235

Query Window, 33–38

scripting, 148–149

table creation, 145–148

task creation, 559–560

View Builder, 302–306

osql.exe command-line tool, 315

Profiler, 40

SQL Server Integration Services (SSIS), 38–39

SQLCMD command-line tool, 330–334

V

validation of XML documents, 481–482

values

parameters, 345

return values

stored procedures, 369–371

triggers, 448–449

user defined functions (UDFs), 410–421

state values, 383

variables, 317–318

VAR function, 599

VarBinary datatype, 14

VarChar datatype, 14

variables

declaring, 317

setting values, 317–318

VARP function, 599

VB.NET connection strings, 649–650

verification for SSIS packages, 548

@@VERSION function, 320, 596–597

versions of SQL Server

mixing, 3

returning information about, 320, 596–597

VIA NetLib, 22

View Builder (SQL Server Management Studio), 302–306

views

- auditing, 306–308
- creating
 - with Management Studio, 302–306
 - with T-SQL, 289–291
- data source views, 523–525
- defined, 9–10
- DELETE statement, 298
- dropping, 302
- encrypting, 308–309
- filtering data, 293–295
- flattening data, 295–297
- hiding sensitive data, 291–293
- indexed views, 8, 10, 310–313
- INSERT statement, 298
- INSTEAD OF triggers, 298–299
- joins, 295–298
- metadata, 310
- modifying, 301, 306
- required fields, 299
- schema binding, 309
- syntax rules, 289
- UPDATE statement, 298
- WITH CHECK OPTION, 299–301

Visio, 644**visual appeal of reports, 517****W****WAITFOR statement**

- best uses, 367
- DELAY parameter, 368
- syntax, 368
- TIME parameter, 368

Watch Window (Debugger), 401**well-formedness of XML documents, 475–476****WHERE clause**

- correlated subqueries, 190–193
- DELETE statement, 75–77
- SELECT statement, 45–49

WHILE statement, 366–367**Windows authentication, 30–31****Windows event log, 566**

- WITH APPEND **option (CREATE TRIGGER statement), 452–453**

- WITH CHECK OPTION, **299–301**

- WITH DB CHAINING ON|OFF **option (CREATE DATABASE statement), 118–119**

- WITH ENCRYPTION **option**

- ALTER VIEW statement, 309

- CREATE TRIGGER statement, 450

- WITH LOG **option (errors), 384**

- WITH NOWAIT **option (errors), 385**

- WITH **option (CREATE INDEX statement), 272**

- WITH RECOMPILE **option (stored procedures), 392–393**

- WITH SETERROR **option (errors), 385**

X**XML (Extensible Markup Language)**

- defined, 469–471
- development history, 470
- documents
 - attributes, 475, 478–479
 - declaration part, 473
 - document part, 472
 - Document Type Definition (DTD), 481–482
 - element content, 481
 - elements, 473–474, 478–479
 - example, 476–481
 - nodes, 474–475
 - parts of (illustration), 471–472
 - valid, 481–482
 - well-formedness, 475–476
 - XML schema, 482
- FOR XML clause, 63, 483–484
- functionality in SQL Server, 482
- hierarchy, 489–490
- namespaces, 479–480
- OPENXML function, 508–514, 620
- Report Definition Language (RDL), 537
- retrieving XML data
 - AUTO option, 486–487
 - directives, 494–503
 - EXPLICIT option, 487–493

XML (Extensible Markup Language) (continued)

- `PATH` option, 503–508
- `RAW` option, 484–486
- storage, 469–470
- support for, 482
- `XML datatype`, 15
- `xml directive`, 495
- XML indexes**, 276
- `xmltext directive`, 502
- XPATH**, 503–508
- XPs (extended stored procedures)**, 393

XSLT (Extensible Stylesheet Language Transformations), 514–516

Y

- `YEAR` function, 601, 603
- yellow arrow in Debugger, 400

Z

- zero or one-to-one relationship, 220

powered by

books24x7[®]

Programmer to Programmer™

Take your library wherever you go.

Now you can access more than 70 complete Wrox books online, wherever you happen to be! Every diagram, description, screen capture, and code sample is available with your subscription to the **Wrox Reference Library**. For answers when and where you need them, go to wrox.books24x7.com and subscribe today!

Find books on

- ASP.NET
- C#/C++
- Database
- General
- Java
- Mac
- Microsoft Office
- .NET
- Open Source
- PHP/MySQL
- SQL Server
- Visual Basic
- Web
- XML



www.wrox.com