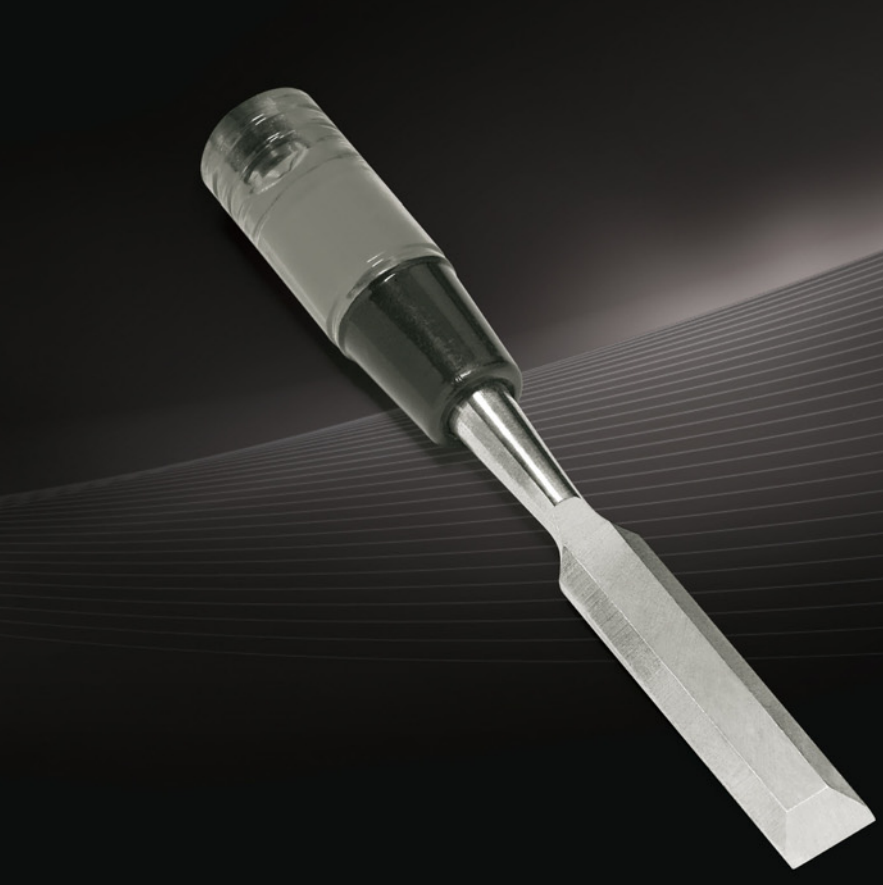
Microsoft

Programming Microsoft® SQL Server® 2008



Leonard Lobel
Andrew J. Brust
Stephen Forte

twentysix
NEW YORK

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

Library of Congress Control Number: 2008935426

Printed and bound in the United States of America.

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

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

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

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

Acquisitions Editor: Ken Jones
Developmental Editor: Sally Stickney
Project Editor: Kathleen Atkins
Editorial Production: Waypoint Press
Technical Reviewer: Kenn Scribner; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Cover: Tom Draper Design

Body Part No. X15-12263

To my partner, Mark, and our children, Adam, Jacqueline, and Joshua, for standing by me through every one of life's turns.

—Leonard Lobel

To my wife, Lauren, and my sons, Sean and Miles. Thank you for your love, your support, and your accommodation of the unreasonable.

—Andrew Brust

To Kathleen, thanks for your support and making me run marathons, which are more painful than book writing and building beta machines.

—Stephen Forte

Contents at a Glance

Part I **Core Fundamentals**

1 Overview 3
2 T-SQL Enhancements 13
3 Exploring SQL CLR 111
4 Server Management 161
5 Security in SQL Server 2008 189

Part II **Beyond Relational**

6 XML and the Relational Database 231
7 Hierarchical Dataand the Relational Database 281
8 Using FILESTREAM for Unstructured Data Storage 307
9 Geospatial Data Types 341

Part III **Reach Technologies**

10 The Microsoft Data Access Machine 377
11 The Many Facets of .NET Data Binding 419
12 Transactions 449
13 Developing Occasionally Connected Systems 491

Part IV **Business Intelligence**

14 Data Warehousing 563
15 Basic OLAP 611
16 Advanced OLAP 639
17 OLAP Queries, Tools, and Application Development 717
18 Expanding Your Business Intelligence with Data Mining 793
19 Reporting Services 879

Table of Contents

Acknowledgments . . . . . **xxi**
Introduction . . . . . **xxv**

Part I Core Fundamentals

1 Overview **3**
Just How Big Is It?. 3
A Book *for* Developers 5
A Book *by* Developers 6
A Book to Show You the Way. 6
Core Technologies 7
Beyond Relational 8
Reaching Out 9
Business Intelligence Strategies 10
Summary 12

2 T-SQL Enhancements **13**
Common Table Expressions 14
Creating Recursive Queries with CTEs 18
The *PIVOT* and *UNPIVOT* Operators. 21
Using *UNPIVOT* 22
Dynamically Pivoting Columns 23
The *APPLY* Operator 25
TOP Enhancements 26
Ranking Functions 28
The *ROW_NUMBER* Function 28
The *RANK* Function 32
The *DENSE_RANK* and *NTILE* Functions 34

Using All the Ranking Functions Together 36
Ranking over Groups Using *PARTITION BY* 37
Exception Handling in Transactions 40
The varchar(max) Data Type 42
The *WAITFOR* Statement 43
DDL Triggers 43
SNAPSHOT Isolation 45
Table-Valued Parameters 45
More than Just Another Temporary Table Solution 46
Working with a Multiple-Row Set 48
Using TVPs for Bulk Inserts and Updates 49
Working with a Single Row of Data 51
Creating Dictionary-Style TVPs 54
Passing TVPs Using ADO.NET 56
TVP Limitations 59
New Date and Time Data Types 59
Separation of Dates and Times 59
More Portable Dates and Times 60
Time Zone Awareness 61
Date and Time Accuracy, Storage, and Format 62
New and Changed Functions 65
The *MERGE* Statement 68
Defining the Merge Source and Target 70
The *WHEN MATCHED* Clause 71
The *WHEN NOT MATCHED BY TARGET* Clause 72
Using *MERGE* for Table Replication 73
The *WHEN NOT MATCHED BY SOURCE* Clause 74
MERGE Output 76
Choosing a Join Method 78
MERGE DML Behavior 79
Doing the "Upsert" 81
The INSERT OVER DML Syntax 90
Extending *OUTPUT...INTO* 90
Consuming *CHANGES* 94
The *GROUPING SETS* Operator 97
Rolling Up by Level 99
Rolling Up All Level Combinations 101
Returning Just the Top Level 103

Mixing and Matching 104
Handling *NULL* Values. 105
New T-SQL Shorthand Syntax 109
Summary 110

3 Exploring SQL CLR 111
Getting Started: Enabling CLR Integration. 112
Visual Studio/SQL Server Integration 113
SQL Server Projects in Visual Studio 114
Automated Deployment. 117
SQL CLR Code Attributes 117
Your First SQL CLR Stored Procedure 118
CLR Stored Procedures and Server-Side Data Access 120
Piping Data with *SqlDataRecord* and *SqlMetaData* 123
Deployment 125
Deploying Your Assembly 125
Deploying Your Stored Procedures. 127
Testing Your Stored Procedures 129
CLR Functions 131
CLR Triggers 136
CLR Aggregates 140
SQL CLR Types. 145
Security 150
Examining and Managing SQL CLR Types in a Database 152
Best Practices for SQL CLR Usage 159
Summary 160

4 Server Management 161
What Is SMO? 161
What About SQL-DMO? 162
Latest Features in SMO 166
Working with SMO in Microsoft Visual Studio 167
Iterating Through Available Servers 169
Retrieving Server Settings 171
Creating Backup-and-Restore Applications 175
Performing Programmatic DBCC Functions with SMO 181
Policy-Based Management. 183
A Simple Policy. 184
Summary 188

5 Security in SQL Server 2008 189

Four Themes of the Security Framework 189
Secure by Design 189
Secure by Default 190
Secure by Deployment 190
Secure Communications 190
SQL Server 2008 Security Overview 191
SQL Server Logins 192
Database Users 193
The *guest* User Account 194
Authentication and Authorization 195
How Clients Establish a Connection 195
Password Policies 197
User-Schema Separation 198
Execution Context 200
Encryption Support in SQL Server 203
Encrypting Data on the Move 204
Encrypting Data at Rest 206
Transparent Data Encryption in SQL Server 2008 211
Creating Keys and Certificates 211
Enabling TDE 213
Querying TDE Views 213
Backing Up the Certificate 214
Restoring an Encrypted Database 215
SQL Server Audit 216
Creating an Audit Object 216
Auditing Options 217
Recording Audits to the File System 219
Recording Audits to the Windows Event Log 220
Auditing Server Events 220
Auditing Database Events 221
Viewing Audited Events 222
Querying Audit Catalog Views 224
How Hackers Attack SQL Server 225
Direct Connection to the Internet 225
Weak System Administrator Account Passwords 226
SQL Server Browser Service 226

SQL Injection . . . 226
Intelligent Observation . . . 227
Summary . . . 228

Part II **Beyond Relational**

6 XML and the Relational Database . . . 231
XML in SQL Server 2000 . . . 233
XML in SQL Server 2008—the *xml* Data Type . . . 234
Working with the *xml* Data Type as a Variable . . . 234
Working with XML in Tables . . . 235
XML Schemas . . . 237
XML Indexes . . . 244
FOR XML Commands . . . 247
FOR XML RAW . . . 248
FOR XML AUTO . . . 248
FOR XML EXPLICIT . . . 250
FOR XML Enhancements . . . 253
OPENXML Enhancements in SQL Server 2008 . . . 261
XML Bulk Load . . . 262
Querying XML Data Using XQuery . . . 263
Understanding XQuery Expressions and XPath . . . 263
SQL Server 2008 XQuery in Action . . . 266
SQL Server XQuery Extensions . . . 275
XML DML . . . 276
Converting a Column to XML . . . 278
Summary . . . 280

7 Hierarchical Data and the Relational Database . . . 281
The *hierarchyid* Data Type . . . 282
Creating a Hierarchical Table . . . 283
The *GetLevel* Method . . . 284
Populating the Hierarchy . . . 285
The *GetRoot* Method . . . 286
The *GetDescendant* Method . . . 286
The *ToString* Method . . . 288
The *GetAncestor* Method . . . 293

Hierarchical Table Indexing Strategies 296
Depth-First Indexing 297
Breadth-First Indexing 298
Querying Hierarchical Tables 299
The *IsDescendantOf* Method 299
Reordering Nodes Within the Hierarchy 301
The *GetReparentedValue* Method 301
Transplanting Subtrees 303
More *hierarchyid* Methods 305
Summary 306

8 Using FILESTREAM for Unstructured Data Storage 307
BLOBs in the Database 307
BLOBs in the File System 309
What's in an Attribute? 309
Enabling FILESTREAM 310
Enabling FILESTREAM for the Machine 311
Enabling FILESTREAM for the Server Instance 312
Creating a FILESTREAM-Enabled Database 313
Creating a Table with FILESTREAM Columns 315
The *OpenSqlFilestream* Native Client API 318
File-Streaming in .NET 319
Understanding FILESTREAM Data Access 321
The Payoff 331
Creating a Streaming HTTP Service 333
Building the WPF Client 338
Summary 340

9 Geospatial Data Types 341
SQL Server 2008 Spaces Out 341
Spatial Models 342
Planar (Flat-Earth) Model 342
Geodetic (Round-Earth) Model 343
Spatial Data Types 344
Defining Space with Well-Known Text 344
Working with *geometry* 345
The *Parse* Method 346
The *STIntersects* Method 347

The *ToString* Method . . . 349
The *STIntersection* Method . . . 350
The *STDimension* Method . . . 350
Working with *geography* . . . 351
On Your Mark 352
The *STArea* and *STLength* Methods . . . 355
Spatial Reference IDs . . . 355
Building Out the *EventLibrary* Database . . . 355
Creating the Event Media Client Application . . . 357
The *STDistance* Method . . . 363
Integrating *geography* with Microsoft Virtual Earth . . . 364
Summary . . . 374

Part III Reach Technologies

10 The Microsoft Data Access Machine . . . 377
ADO.NET and Typed *DataSets* . . . 378
Typed *DataSet* Basics . . . 378
TableAdapter Objects . . . 380
Connection String Management . . . 381
Using the TableAdapter Configuration Wizard . . . 382
More on Queries and Parameters . . . 385
DBDirect Methods and Connected Use of Typed *DataSet* Objects . . . 387
"Pure" ADO.NET: Working in Code . . . 387
Querying 101 . . . 388
LINQ: A New Syntactic Approach to Data Access . . . 392
LINQ to DataSet . . . 392
LINQ Syntax, Deconstructed . . . 393
LINQ to SQL and the ADO.NET Entity Framework: ORM Comes to .NET . . . 395
Why Not Stick with ADO.NET? . . . 396
Building an L2S Model . . . 397
The Entity Framework: Doing ORM the ADO.NET Way . . . 402
XML Behind the Scenes . . . 405
Querying the L2S and EF Models . . . 406
Adding Custom Validation Code . . . 410
Web Services for Data: Using ADO.NET Data Services Against EF Models . . . 411
Creating the Service . . . 412

Testing the Service . . . 414
Building the User Interface . . . 414
Data as a Hosted Service: SQL Server Data Services . . . 415
Summary: So Many Tools, So Little Time . . . 417

11 The Many Facets of .NET Data Binding . . . 419
Windows Forms Data Binding: The Gold Standard . . . 420
Getting Ready . . . 420
Generating the UI . . . 421
Examining the Output . . . 423
Converting to LINQ to SQL . . . 424
Converting to Entity Framework . . . 425
Converting to ADO.NET Data Services . . . 426
Data Binding on the Web with ASP.NET . . . 427
L2S and EF Are Easy . . . 428
Beyond Mere Grids . . . 429
Data Binding Using Markup . . . 430
Using AJAX for Easy Data Access . . . 430
ASP.NET Dynamic Data . . . 435
Data Binding for Windows Presentation Foundation . . . 438
Design Time Quandary . . . 439
Examining the XAML . . . 441
Grand Finale: Silverlight . . . 445
Summary . . . 447

12 Transactions . . . 449
What Is a Transaction? . . . 450
Understanding the ACID Properties . . . 450
Local Transaction Support in SQL Server 2008 . . . 453
Autocommit Transaction Mode . . . 453
Explicit Transaction Mode . . . 453
Implicit Transaction Mode . . . 456
Batch-Scoped Transaction Mode . . . 457
Using Local Transactions in ADO.NET . . . 459
Transaction Terminology . . . 461
Isolation Levels . . . 462
Isolation Levels in SQL Server 2008 . . . 462
Isolation Levels in ADO.NET . . . 467

Distributed Transactions 468
Distributed Transaction Terminology 469
Rules and Methods of Enlistment 470
Distributed Transactions in SQL Server 2008 472
Distributed Transactions in the .NET Framework 473
Writing Your Own Resource Manager 477
Using a Resource Manager in a Successful Transaction 481
Transactions in SQL CLR (CLR Integration) 485
Putting It All Together 489
Summary 490

13 Developing Occasionally Connected Systems 491
Comparing Sync Services with Merge Replication 492
Components of an Occasionally Connected System 493
Merge Replication 494
Getting Familiar with Merge Replication 494
Creating an Occasionally Connected Application with Merge Replication 496
Configuring Merge Replication 499
Creating a Mobile Application Using Microsoft Visual Studio 2008 . . . 520
Sync Services for ADO.NET 533
Sync Services Object Model 534
Capturing Changes for Synchronization 538
Creating an Application Using Sync Services 543
Additional Considerations 557
Summary 560

Part IV Business Intelligence

14 Data Warehousing 563
Data Warehousing Defined 563
The Importance of Data Warehousing 564
What Preceded Data Warehousing 566
Lack of Integration Across the Enterprise 567
Little or No Standardized Reference Data 568
Lack of History 568
Data Not Optimized for Analysis 568
As a Result... 569
Data Warehouse Design 570

The Top-Down Approach of Inmon 572
The Bottom-Up Approach of Kimball 574
What Data Warehousing Is Not 580
OLAP 580
Data Mining 581
Business Intelligence 582
Dashboards and Scorecards 583
Performance Management 585
Practical Advice About Data Warehousing 585
Anticipating and Rewarding Operational Process Change 586
Rewarding Giving Up Control 586
A Prototype Might Not Work to Sell the Vision 586
Surrogate Key Issues 587
Currency Conversion Issues 587
Events vs. Snapshots 588
SQL Server 2008 and Data Warehousing 589
T-SQL *MERGE* Statement 589
Change Data Capture 592
Partitioned Table Parallelism 600
Star-Join Query Optimization 603
SPARSE Columns 604
Data Compression and Backup Compression 605
Learning More 610
Summary 610

15 Basic OLAP 611
Wherefore BI? 611
OLAP 101 613
OLAP Vocabulary 614
Dimensions, Axes, Stars, and Snowflakes 615
Building Your First Cube 617
Preparing Star Schema Objects 617
A Tool by Any Other Name 618
Creating the Project 619
Adding a Data Source View 621
Creating a Cube with the Cube Wizard 625
Using the Cube Designer 626
Using the Dimension Wizard 629

Using the Dimension Designer 632
Working with the Properties Window and Solution Explorer 634
Processing the Cube 635
Running Queries 636
Summary 637

16 Advanced OLAP 639

What We'll Cover in This Chapter 640
MDX in Context 640
And Now a Word from Our Sponsor... 640
Advanced Dimensions and Measures 641
Keys and Names 641
Changing the *All* Member 644
Adding a Named Query to a Data Source View 645
Parent/Child Dimensions 647
Member Grouping 651
User Table Time Dimensions, Attribute Relationships, Best Practice Alerts, and Dimension/Attribute Typing 652
Server Time Dimensions 660
Fact Dimensions 661
Role-Playing Dimensions 664
Advanced Measures 665
Calculations 667
Calculated Members 668
Named Sets 673
More on Script View 674
Key Performance Indicators 677
KPI Visualization: Status and Trend 678
A Concrete KPI 679
Testing KPIs in Browser View 681
KPI Queries in Management Studio 683
Other BI Tricks in Management Studio 688
Actions 689
Actions Simply Defined 690
Designing Actions 690
Testing Actions 692
Partitions, Storage Settings, and Proactive Caching 693
Editing and Creating Partitions 694

Partition Storage Options . . . 696
Proactive Caching . . . 697
Additional Features and Tips . . . 699
Aggregations . . . 700
Algorithmic Aggregation Design . . . 700
Usage-Based Aggregation Design . . . 701
Manual Aggregation Design (and Modification) . . . 702
Aggregation Design Management . . . 704
Aggregation Design and Management Studio . . . 705
Perspectives . . . 705
Translations . . . 707
Roles . . . 712
Summary . . . 715

17 OLAP Queries, Tools, and Application Development . . . 717

Using Excel . . . 719
Connecting to Analysis Services . . . 719
Building the PivotTable . . . 723
Exploring PivotTable Data . . . 725
Scorecards . . . 727
Creating and Configuring Charts . . . 729
In-Formula Querying of Cubes . . . 732
Visual Studio Tools for Office and Excel Add-Ins . . . 737
Excel Services . . . 738
Beyond Excel: Custom OLAP Development with .NET . . . 743
MDX and Analysis Services APIs . . . 744
Moving to MDX . . . 744
Management Studio as an MDX Client . . . 745
OLAP Development with ADO MD.NET . . . 758
Using Analysis Management Objects . . . 769
XMLA at Your (Analysis) Service . . . 771
Analysis Services CLR Support: Server-Side ADO MD.NET . . . 782
Summary . . . 792

18 Expanding Your Business Intelligence with Data Mining . . . 793

Why Mine Your Data? . . . 793
SQL Server 2008 Data Mining Enhancements . . . 797
Getting Started . . . 798
Preparing Your Source Data . . . 798

Creating an Analysis Services Project . . . 800
Using the Data Mining Wizard and Data Mining Structure Designer . . . 802
Creating a Mining Structure . . . 804
Creating a Mining Model . . . 805
Editing and Adding Mining Models . . . 810
Deploying and Processing Data Mining Objects . . . 816
Viewing Mining Models . . . 818
Validating and Comparing Mining Models . . . 827
Nested Tables . . . 830
Using Data Mining Extensions . . . 836
Data Mining Modeling Using DMX . . . 837
Data Mining Predictions Using DMX . . . 848
DMX Templates . . . 856
Data Mining Applied . . . 856
Data Mining and API Programming . . . 857
Using the Windows Forms Model Content Browser Controls . . . 858
Executing Prediction Queries with ADO MD.NET . . . 860
Model Content Queries . . . 860
ADO MD.NET and ASP.NET . . . 861
Using the Data Mining Web Controls . . . 862
Developing Managed Stored Procedures . . . 863
XMLA and Data Mining . . . 865
Data Mining Add-ins for Excel 2007 . . . 866
Summary . . . 877

19 Reporting Services . . . 879

Using the Report Designer . . . 880
Creating a Basic Report . . . 883
Applying Report Formatting . . . 887
Adding a Report Group . . . 890
Working with Parameters . . . 892
Writing Custom Report Code . . . 897
Creating an OLAP Report . . . 900
Creating a Report with a Matrix Data Region . . . 906
Tablix Explained . . . 910
Adding a Chart Data Region . . . 915
Making a Report Interactive . . . 917
Delivering Reports . . . 919
Deploying to the Report Server . . . 919

Accessing Reports Programmatically 928
Administering Reporting Services 937
Using Reporting Services Configuration Manager 937
Using Report Manager and Management Studio 940
Integrating with SharePoint 949
Summary 951

Index 953

Acknowledgments

Working on this book has truly been the most rewarding experience of my professional career thus far, and I need to thank a great many people who have made it possible.

I first met Andrew Brust about 10 years ago, and we've enjoyed a close working relationship and growing friendship for the past 7 of those. I can't count the number of times Andrew has opened doors for me with project, writing, and speaking opportunities—and now, of course, this book. Andrew introduced me to Stephen Forte back in 2004, and after 30 years in the industry, I've learned to find new appreciation for the art of software development through Stephen's enthusiastic (that is, wacky and wild) personality. Andrew and Stephen both made this project significantly easier by producing the original edition of this book—an excellent treatment of Microsoft SQL Server 2005 that set the starting point for this new 2008 edition. It's been an absolute thrill and honor that they invited me to join them this time around and to assume the role of lead author for the new edition. Thanks to you both for entrusting me with that responsibility, as well as for your own hard work on this edition.

We could never have produced a book so broad and deep in scope without the additional aid of the other guest authors. Elsie Pan, Paul Delcogliano, Mark Frawley, and Jeff Bolton each did an outstanding job writing brand-new chapters, and Elsie also revised material from the last edition. Heartfelt thanks go out as well to Kenn Scribner, who performed an incredibly detail-oriented tech review of the entire book, and especially for helping out with updating important material at the bottom of the ninth with two men out and three men on. I'm very grateful for their contributions and professionalism, and I feel privileged to have worked with each of them. I'd also like to thank Jay Rabin, and all the wonderful folks at twentysix New York, for their continuous stream of support and encouragement throughout this whole project.

I was very lucky to have worked closely with Kathleen Atkins, Sally Stickney, and Ken Jones of Microsoft Press; Steve Sagman from Waypoint Press; and copy editor Jennifer Harris. Their superb editorial contributions, project planning, and overall guidance were vital to the production of this book. Double thanks go to Sally, who was always available to me (weekends too) for much-needed guidance as I entered the world of book writing. And the assistance provided by a number of people from various Microsoft product teams helped tackle the challenge of writing about new software as it evolved through several beta releases. So thank you to Steve Lasker, for support with Compact and Sync Services, and to Roger Doherty and Isaac Kunen for support with the "beyond relational" features. In particular, Roger inspired several of the FILESTREAM and geospatial demos found in those chapters. George Sobhy was also a great help with geospatial—he even arranged for a shared desktop demo between New York and Cairo (and thanks to Stephen Forte too, for the introduction).

This would all be moot, of course, without the love and support of my family. Almost all of my spare time over the past year was spent working on this project in one form or another—researching and writing new material, editing the whole book, and coordinating its production—which at times transformed me into an absentee partner and father. I owe an enormous debt of gratitude to my wonderful partner, Mark, and my awesome kids, Adam, Jacqueline, and Josh, for putting up with it all. So thanks, gang, I'm back home now! And thanks most of all to dear Mom, bless her soul, for always encouraging me to write with "expression" since the first grade.

—*Leonard Lobel*

Writing a book is hard, especially for the people in the authors' lives who lend heroic support. Revising a book is hard too, especially for the people in the authors' lives who lend that support *again*.

With that in mind, I'd like to thank my wife, Lauren (who endured this project while at the same time earning her master's degree and being an amazing mom to our two boys). And I thank our boys as well: Miles (who, though only four years old, is nonetheless a veteran of both editions of this book) and Sean (who, at age 18 months, has endured yet another thing that his older brother experienced first). All three have tolerated my intolerable absences from their events and their lives. Each one has my gratitude for the patience and understanding shown to me.

I'd also like to thank everyone at twentysix New York, especially Jay Rabin, for granting me a period of calm, with unprecedented duration, to get the work on this edition done.

Finally, but certainly not least of all, I'd like to thank Leonard Lobel for "taking the wheel" on this edition of the book. Had he not done so, we simply would not *have* this edition of the book. Lenni is prone to thanking me for exposing him to opportunity. What he fails to understand is that by repeatedly succeeding, he makes me look good simply for having the good taste to recommend him.

—*Andrew Brust*

It's nice to have your name on the cover of a book, but without help from various people, this book never would have happened. I'll start with Andrew and Lenni, my wonderful co-authors, both easy to work with, dedicated, and also very patient with me. The folks at Microsoft Press were all great to work with and had considerable energy and flexibility.

I would not have been able to take on this project if I did not have the support of the folks at Telerik, Triton Works, and Dash Soft, three companies that I work very closely with. I would like to give special thanks to the leaders of those firms, Vassil Terziev, Mark Storen, and Remon "FGD" Zakaria, for their understanding and support when deadlines for the book loomed.

We have also had tons of great reviewers. I was blessed to have folks like Kimberly Tripp, Peter DeBetta, and Roman Rehak help out with reading my chapters, as well as Kevin Collins, Remi Caron, Joel Semeniuk, Eileen Rumwell, Steve Lasker, Kirk Haselden, Ted Lee, Sergei Ivanov, Richard Campbell, Goksin Bakir, Malek Kemmou, Jason Gideon, Julie Lerman, Bill Ramos, Tom Halligan, and finally Jack Prilook—who looked at my manuscript 13 times.

I started this book on the first day of classes of my second year of my MBA education. Some days I had to choose whether to write on Policy-Based Management or macroeconomic trends in China and India. I'd like to thank all my group members at EMBA 26, most especially Dr. Ian Miller, Rosa Alvarado, Jason Nocco, Dmitriy Malinovskiy, and Cyrus Kazi. As fate would have it, I type these words on my last day of school. How fitting to finish a book and an MBA in the same weekend.

—Stephen Forte

Introduction

Welcome, developer!

The book you are holding, much like Microsoft SQL Server 2008 itself, builds on a great "previous release." SQL Server 2005 was—architecturally speaking—a groundbreaking upgrade from earlier versions of the product, and the 2005 edition of this book was a new printed resource that provided comprehensive coverage of the revamped platform. This new edition includes thoroughly updated coverage of the most important topics from the past edition, plus brand-new coverage of all the new exciting and powerful features for developers in SQL Server 2008. As with the 2005 edition, we set out to produce the best book for developers who need to program SQL Server 2008 in the many ways that it can be programmed.

To best understand our approach, we ask that you consider likening SQL Server 2008 to, of all things, a Sunday newspaper. A Sunday newspaper is made up of multiple sections, each of which is written separately and appeals to a distinct audience. The sections do have overlapping content and share some readership, of course, but most people don't read the whole paper, and they don't need to. Meanwhile, the entire paper is considered a single publication, and those who read it think of themselves as readers of the paper rather than of one or more of its sections. Likewise, SQL Server has many pieces to it: few people will use them all, and people will need to learn about them gradually, over time, as their business needs dictate.

Our book reflects this reality and in many ways replicates the structure of a Sunday newspaper. For one thing, a great number of authors have been involved in producing the book, drawing on their expertise in their chapters' specific subject matter. For another, the context of certain chapters differs markedly from those of other chapters. Some chapters cover specific subject matter deeply. Others cover a broader range of material, and do so at a higher level. That's an approach we didn't anticipate when we authored the 2005 edition of this book. But it's the approach we found most effective by the time we finished it, and one which we continue to follow in this new edition for SQL Server 2008. We have found that it makes an otherwise overwhelming set of technologies much more approachable and makes the learning process much more modular.

Make no mistake, though—the overall vision for the book is a cohesive one: to explore the numerous programmability points of SQL Server 2008 and, in so doing, provide widespread coverage of the great majority of the product's features, in a voice that caters to developers' interests. Whether you read every chapter in the book or just some of them and whether you read the book in or out of order, our goal has been to provide you with practical information, numerous useful samples, and a combination of high-level coverage and detailed discussion, depending on how deep we thought developers would want to go.

Just as the Sunday newspaper doesn't cover everything that's going on in the world, this book won't teach you everything about SQL Server. For example, we don't cover high-availability/fault tolerance features such as replication, clustering, or mirroring. We don't discuss query plans and optimization, nor do we investigate SQL Server Profiler, SQL Trace, or the Database Engine Tuning Advisor. Some features covered in the 2005 edition have not changed significantly in SQL Server 2008, such as native XML Web Services, Service Broker, Integration Services, and cross-tier debugging. These topics are also not covered, in order to make room for new SQL Server 2008 features. (The 2005 edition chapters that cover those topics are available for you to download from this book's companion Web site, which we explain toward the end of this introduction.)

We discovered as we wrote the book that covering *everything* in the product would result in a book unwieldy in size and unapproachable in style. We hope we struck the right balance, providing a *digestible* amount of information with enough developer detail and enough pointers to other material to help you become a seasoned SQL Server professional.

Who This Book Is For

Now that we have established *what* the book does and does not cover, we'd like to clarify just *who* we believe will be most interested in it and best served by it. In a nutshell, this book is for .NET and SQL Server developers who work with databases and data access, at the business logic/middle-tier layer as well as the application level.

In our perhaps self-centered view of the development world, we think this actually describes *most* .NET developers, but clearly some developers are more interested in database programming in general, and SQL Server specifically, than others, and it is this more interested group we want to reach.

We assume that you have basic, working knowledge of .NET programming on the client and Transact-SQL (T-SQL) on the server, although SQL experience on any platform can easily substitute. We also assume that you are comfortable with the basics of creating tables, views, and stored procedures on the server. On the client tools side, we assume that you are familiar with the prior generation of SQL Server and .NET development tools. If you've already been working with SQL Server Management Studio in SQL Server 2005, you'll feel right at home with the 2008 version, which has been extended to support new server features (and now even includes IntelliSense for T-SQL!). If you're still running SQL Server 2000 or earlier, you'll definitely appreciate SQL Server Management Studio as a vast improvement over the two primary tools that preceded it—Enterprise Manager and Query Analyzer. SQL Server Management Studio essentially represents the fusion of those two tools, packaged in a modern user interface (UI) shell very similar to that provided by Microsoft Visual Studio—complete with customizable menus and toolbars, floatable and dockable panes, solutions, and projects. The primary tool for .NET devel-

opment is, of course, Visual Studio 2008, and experience with any version will also be beneficial for you to have.

Having said all that, we have a fairly liberal policy regarding these prerequisites. For example, if you've only dabbled with SQL and .NET, that's OK, as long as you're willing to try and pick up on things as you read along. Most of our code samples are not that complex. However, our explanations do assume some basic knowledge on your part, and you might need to do a little research if you lack the experience.

Note For the sake of consistency, all the .NET code in this book is written in C#. (The only exceptions to this rule will be found in Chapter 19 for Reporting Services, since only Visual Basic .NET is supported for scripting report expressions and deployments.) However, this book is in no way C#-oriented, and there is certainly nothing C#-specific in the .NET code provided. As we just stated, the code samples are not very complex, and if you are more experienced with Visual Basic .NET than you are with C#, you should have no trouble translating the C# code to Visual Basic .NET on the fly as you read it.

In addition to covering the SQL Server core relational engine, its latest breed of "beyond relational" capabilities, and its ancillary services, this book also provides in-depth coverage of SQL Server's business intelligence (BI) features, including Reporting Services, and the online analytical processing (OLAP) and data mining components of Analysis Services. Although ours is not a BI book per se, it is a database developer's book, and we feel strongly that all these features should be understood by mainstream database developers. BI is really one of the cornerstone features of SQL Server 2008, so the time is right for traditional database developers to "cross over" to the world of BI.

Realizing that these technologies, especially OLAP and data mining, will be new territory for many readers, we assume no knowledge of them on your part. Any reader who meets the prerequisites already discussed should feel comfortable reading about these BI features and, more than likely, feel ready and excited to start working with BI after reading the BI-focused chapters.

How This Book Is Organized

This book is broken down into four parts. Each part follows a specific SQL Server "theme," if you will.

Part I begins with an overview that gives you a succinct breakdown of the chapters in all four parts of the book. Then it dives right in to core SQL Server technologies. We explore the many powerful enhancements made to Transact-SQL (T-SQL), both in SQL Server 2005 and 2008 (in that order). We also introduce you to SQL Server's .NET Common Language Runtime (CLR) integration features, which cut across our discussions of data types and server-side

programming. You'll learn how to programmatically administer the server using Server Management Objects (SMO), which were introduced in SQL Server 2005, and how to use the new administrative framework called Policy-Based Management (PBM) in SQL Server 2008. Then we tackle security. After quickly covering basic SQL Server security concepts, we show how to encrypt your data both while in transit (traveling across the network) and at rest (on disk). We'll also teach the latest security features in SQL Server 2008, including Transparent Data Encryption (TDE) and SQL Server Audit, which you will find extremely useful in today's world of regulatory compliance.

Part II is dedicated to the SQL Server 2008 "beyond relational" release theme, which is all about working with semistructured and unstructured data. This is a concept that broadens our traditional view of relational databases by getting us to think more "outside the box" in terms of all the different types of data that SQL Server can be used to manage, query, and manipulate. We begin with a chapter on XML support (which was spearheaded in SQL Server 2005), and provide detailed coverage that includes the recent XML enhancements made in SQL Server 2008. All the remaining chapters in Part II cover nonrelational features that are brand new in SQL Server 2008. These include hierarchical tables, native file streaming, and geospatial capabilities. These features are designed to enrich the native database engine by bringing unprecedented intelligence and programming convenience down to the database level.

In Part III, we move away from the server and discuss concepts relating to actual database software development, be it in the middle tier or at the application level. This includes data access using "traditional" ADO.NET, language-integrated query (LINQ), the ADO.NET Entity Framework, and the latest innovations, ADO.NET Data Services and SQL Server Data Services. After you succeed in accessing your data, you'll need to deliver that data to your users, and that means data binding. We'll dig in to data binding for Microsoft Windows and ASP.NET Web applications, as well as the newest UI platforms, Windows Presentation Foundation (WPF) and Silverlight. We also cover transactions and various other topics relevant to extending your databases' reach with technologies such as merge replication, Sync Services for ADO.NET, and mobile database application development with SQL Server Compact 3.5.

Part IV is our BI section. In it, we provide efficient, practical coverage of SQL Server Analysis Services and Reporting Services. We are particularly proud of this section because we assume virtually no BI or OLAP knowledge on your part and yet provide truly deep coverage of SQL Server BI concepts, features, and programming. We have a chapter dedicated to the topic of data warehousing. In it, you'll see how to use a new SQL Server 2008 feature called Change Data Capture (CDC) to facilitate incremental updating of large data warehouses. Furthermore, we cover all the new important BI features in SQL Server 2008, expanded to include the latest data mining add-ins for Microsoft Office Excel 2007. The Reporting Services chapter has been written from scratch for the completely reworked and enhanced Report Designer, and also teaches you the many ways that Reporting Services can be programmed and managed.

Together, the four parts of the book provide you with a broad inventory of a slew of SQL Server 2008 developer-relevant features and the conceptual material necessary to understand them. We don't cover everything in SQL Server 2008, but we will arm you with a significant amount of core knowledge and give you the frame of reference necessary to research the product further and learn even more. Where appropriate, we refer you to SQL Server Books Online, which is the complete documentation library for SQL Server 2008 (available from the Start Menu under Programs, Microsoft SQL Server 2008, Documentation And Tutorials).

Code Samples and the Book's Companion Web Site

All the code samples discussed in this book can be downloaded from the book's companion Web site at the following address:

http://www.microsoft.com/mspress/companion/9780735625990/

Important This book and its sample code were written for, and tested against, the Release Candidate (RC0) version of SQL Server 2008 Developer edition, released in June 2008. If and when we discover any incompatibilities with the Release To Manufacturer (RTM) version, or any further service packs that are later released, our intent is to update the sample code and post errata notes on the book's companion Web site, available at *http://www.microsoft.com/mspress/companion/9780735625990/*. Please monitor that site for new code updates and errata postings.

In addition to all the code samples, the book's companion Web site also contains several chapters from the 2005 edition of this book that were not updated for the 2008 edition. These include the chapters on native XML Web Services and Service Broker, which are features that have not been widely adopted since they were introduced in SQL Server 2005 but that continue to be supported in SQL Server 2008. The 2005 edition chapters covering SQL Server Management Studio (the primary graphical tool you'll use for most of your database development work), SQL Server 2005 Express edition, Integration Services, and debugging are posted on the companion Web site as well. With the inclusion of all the new SQL Server 2008 coverage, space constraints simply did not permit us to include these topics (which have not changed significantly in SQL Server 2008) in this new edition. And while we provide completely new coverage on the latest data binding techniques, the 2005 edition covers ADO.NET programming techniques against then-new SQL Server features, and so it is posted on the companion Web site as well. This book's chapter on OLAP Application development has also been revised to include Excel 2007 coverage, and the 2005 edition is available on the companion Web site for those developers who are still working with Excel 2003 against Analysis Service OLAP cubes.

Because this is a developer book, we often include one or more Visual Studio projects as part of the sample code, in addition to SQL Server Management Studio projects containing T-SQL or Analysis Services script files. Within the companion materials parent folder is a child folder for each chapter. Each chapter's folder, in turn, contains either or both of the following two folders: SSMS and VS. The former contains a SQL Server Management Studio solution (.ssmssln file), the latter contains a Visual Studio solution (.sln file). Some chapters might have multiple Visual Studio solutions. After you've installed the companion files, double-click a solution file to open the sample scripts or code in the appropriate integrated development environment (IDE).

Because most of the code is explained in the text, you might prefer to create it from scratch rather than open the finished version supplied in the companion sample code. However, the finished version will still prove useful if you make a small error along the way or if you want to run the code quickly before reading through the narrative that describes it.

Some of the SQL Server Management Studio projects contain embedded connections that are configured to point to a default instance of SQL Server running on your local machine. Similarly, some of the Visual Studio source code contains connections or default connection strings (sometimes in code, sometimes in settings or configuration files, and other times in the *Text* property of controls on the forms in the sample projects) that are configured likewise. If you have SQL Server 2008 installed as the default instance on your local machine with Windows-integrated security and the *AdventureWorks2008* sample database, the majority of the sample code should run without modification. If not, you'll need to modify the server name, instance name, or user credentials accordingly to suit your environment. You'll also need to install *AdventureWorks2008* if you have not already done so. (Download instructions for all the sample databases are given in the sections ahead.)

A number of chapters rely on various popular sample databases available for SQL Server. These include the *Northwind* and just-mentioned *AdventureWorks2008* sample transactional databases, the *AdventureWorksDW2008* sample data warehouse database, and the *Adventure Works DW 2008* Analysis Services sample database. None of our examples use the *pubs* database, which has been around since long before SQL Server 2000.

Using the Sample *Northwind* Database

You can use the version of *Northwind* that came with SQL Server 2000, if you have it, and attach it to your SQL Server 2008 server. Microsoft has also published a Windows Installer file (.msi) that will install the *Northwind* sample database on your server (even the older *pubs* sample database is included). The installer provides both the primary database file and the log file that can be directly attached, as well as T-SQL scripts, which can be executed

to create the databases from scratch. At press time, the download page for the *Northwind* installer file is *http://www.microsoft.com/downloads/details.aspx?FamilyID=06616212-0356-46a0-8da2-eebc53a68034&DisplayLang=en*. An Internet shortcut to this URL is included with this chapter's sample code. If the link does not work for you, try running a Web search on "Northwind and pubs Sample Databases for SQL Server 2000."

Using the Sample *AdventureWorks2008* Databases

As of SQL Server 2005, and updated for SQL Server 2008, Microsoft provides the *AdventureWorks* family of databases. You can download these sample databases from CodePlex, which is Microsoft's open source Web site (in fact, all of Microsoft's official product code samples are hosted on CodePlex). This book uses the *AdventureWorks2008* relational online transaction processing (OLTP) database, the *AdventureWorksDW2008* relational data warehouse database, and the *AdventureWorksAS2008* Analysis Services database. The latest version of these sample databases are designed for use only with SQL Server 2008 and will not work with SQL Server 2005. (The older *AdventureWorks* databases for SQL Server 2005 are still available on CodePlex at the time of this writing, however.)

At press time, the download location for all sample *AdventureWorks2008* databases is *http://www.codeplex.com/MSFTDBProdSamples*. Click the Releases tab on this page to select any of the sample databases for downloading to your machine. An Internet shortcut to this URL is included on the book's companion Web site. If the link does not work for you, try running a Web search on "SQL Server 2008 product sample databases."

The *AdventureWorks2008* OLTP database uses the new FILESTREAM feature in SQL Server 2008, and therefore requires that FILESTREAM be enabled for the instance on which *AdventureWorks2008* is installed. Chapter 8 is devoted to FILESTREAM, and you should refer to the "Enabling FILESTREAM" section in that chapter, which shows how to enable FILESTREAM in order to support *AdventureWorks2008*.

Important The samples for this book are based on the 32-bit version of the sample *AdventureWorks2008* databases, which is almost—but not exactly—identical to the 64-bit version. If you are using the 64-bit version of these sample databases, some of your query results might vary slightly from those shown in the book's examples.

System Requirements

To follow along with the book's text and run its code samples successfully, we recommend that you install the Developer edition of SQL Server 2008, which is available to a great number of developers through Microsoft's MSDN Premium subscription, on your PC. You will also need Visual Studio 2008; we recommend that you use the Professional edition or one of the Team edition releases, each of which is also available with the corresponding edition of the MSDN Premium subscription product.

Important To cover the widest range of features, this book is based on the Developer edition of SQL Server 2008. The Developer edition possesses the same feature set as the Enterprise edition of the product, although Developer edition licensing terms preclude production use. Both editions are high-end platforms that offer a superset of the features available in other editions (Standard, Workgroup, Web, and Express). We believe that it is in the best interest of developers for us to cover the full range of developer features in SQL Server 2008, including those available only in the Enterprise and Developer editions.

Most programmability features covered in this book are available in every edition of SQL Server 2008. One notable exception is the lack of Analysis Services support in the Workgroup, Web, and Express editions. Users of production editions other than the Enterprise edition should consult the SQL Server 2008 Features Comparison page at *http://msdn.microsoft.com/en-us/library/cc645993.aspx* for a comprehensive list of features available in each edition, in order to understand which features covered in the book are available to them in production.

To run these editions of SQL Server and Visual Studio, and thus the samples in this book, you'll need the following 32-bit hardware and software. (The 64-bit hardware and software requirements are not listed here but are very similar.)

- 600-MHz Pentium III–compatible or faster processor (1-GHz minimum, but 2GHz or faster processor recommended).
- Microsoft Windows 2000 Server with Service Pack (SP) 4 or later; Windows 2000 Professional Edition with SP4 or later; Windows XP with SP2 or later; Windows Server 2003 (any edition) with SP1 or later; Windows Small Business Server 2003 with SP1 or later; or Windows Server 2008 (any edition).
- For SQL Server 2008, at least 512 MB of RAM (1 GB or more recommended).
- For Visual Studio 2008, 192 MB (256 MB recommended).
- For SQL Server 2008, approximately 1460 MB of available hard disk space for the recommended installation. Approximately 200 MB of additional available hard disk space for SQL Server Books Online.
- For Visual Studio 2008, maximum of 20 GB of available space required on installation drive. This includes space for the installation of the full set of MSDN documentation.

- Internet connection required to download the code samples for each chapter from the companion Web site. A few of the code samples require an Internet connection to run as well.
- CD-ROM or DVD-ROM drive recommended.
- Super VGA (1024 × 768) or higher resolution video adapter and monitor recommended.
- Microsoft Mouse or compatible pointing device recommended.
- Microsoft Internet Explorer 6.0 SP1 or later. Microsoft Internet Explorer 7.0 recommended.
- For SQL Server Reporting Services, Microsoft Internet Information Services (IIS) 6.0 or later and ASP.NET 2.0 or later.

Support for This Book

Every effort has been made to ensure the accuracy of this book and the companion content. As corrections or changes are collected, they will be added to a Microsoft Knowledge Base article.

Microsoft Press provides support for books and companion content at the following Web site:

http://www.microsoft.com/learning/support/books/

Questions and Comments

If you have comments, questions, or ideas regarding the book or the companion content, or questions that are not answered by visiting the preceding sites, please send them to Microsoft Press via e-mail to:

mspinput@microsoft.com

Or send them via postal mail to

Microsoft Press
Attn: Programming Microsoft SQL Server 2008 Editor
One Microsoft Way
Redmond, WA 98052-6399

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

Part I

Core Fundamentals

In this part:

Chapter 1 Overview . . . 3
Chapter 2 T-SQL Enhancements . . . 13
Chapter 3 Exploring SQL CLR . . . 111
Chapter 4 Server Management . . . 161
Chapter 5 Security in SQL Server 2008 . . . 189

Chapter 1

Overview

—Leonard Lobel

This is a book about Microsoft SQL Server 2008, written specifically with the developer in mind.

Microsoft's latest release of SQL Server improves upon its predecessors—SQL Server 2005 and earlier—in every key area, just as you would expect of any new version. There are many enhancements and new features in the product that yield important benefits across the board. Collectively, of course, these product enhancements and new features continue to bolster SQL Server's competitive position as an industry-strength database platform capable of handling the most demanding workloads. In this book, our particular focus is on *programmability for database development*—a space in which the product continues to advance in new and powerful ways with the release of SQL Server 2008, the most programmable version of SQL Server to date.

Features like clustering, mirroring, and the ability to add new RAM and CPUs on the fly with zero downtime—just to name a few—are certainly important in any serious enterprise-level system. But to reiterate, this is a database book that targets developers, not administrators (although we do have an entire chapter dedicated to programmatically administering SQL Server). Reliability, availability, and scalability features that contribute toward making SQL Server a rock-solid platform are quite impressive and significant in their own right, but they are not particularly programmable and so they are not covered in this book.

If you're a developer, however, then welcome! This book is just for you. Whether you're programming against SQL Server 2008 natively at the database tier or further up the stack using .NET, this book shows you the way. Within these chapters, you'll find detailed coverage of the newest and most important SQL Server programming features. Together, we'll explore the plethora of ways in which SQL Server 2008 can be programmed—empowering you to rapidly develop rich database applications for your end users, while having fun in the process.

Just How Big Is It?

With each version of SQL Server, we tend to ask the same set of questions about the new release. Is it big? How many features have been added, what are they, and how relevant are they to my needs? What previous barriers have been removed? Has the product at its core changed radically since the last version? What's no longer supported? After probing like this for a while, we typically arrive at the ultimate question: Should I upgrade?

Let's be clear about this. SQL Server 2008 is *not* the watershed release that its predecessor SQL Server 2005 was roughly three years ago. At that time, SQL Server 2005 really redefined the product beyond just a relational database engine—which was already quite matured in SQL Server 2000 nearly six years prior. In SQL Server 2000 (and earlier versions), the relational database engine *was* the product. New advances beyond the relational database engine—in particular, business intelligence (BI) services for extraction, transformation, and loading (ETL) and for reporting and analysis—began appearing as early as 1999 and continued to steadily emerge after the arrival of SQL Server 2000. These capabilities were integrated into the product sporadically as a patchwork of add-ons, wizards, and management consoles. Over time, the result was a relational database system loosely bundled with a collection of value-added features supported by a somewhat inconsistent toolset.

Microsoft changed all that with SQL Server 2005, by giving the platform a complete structural overhaul that revolutionized the product. Disruptive change in the software industry—resulting from the distributed nature of a growing Internet for business-to-business (B2B) integration, as well as a growing market for BI, including online analytical processing (OLAP), data mining, and reporting—was the driving force behind the product's radical makeover. As of SQL Server 2005, the relational database engine no longer takes center stage. Rather, it is positioned alongside a series of core services that together comprise the overall product. The result is a broader, richer, and more consistent set of features and services that are built into—rather than bolted onto—the platform.

SQL Server 2008, in turn, represents a natural evolution of this model, building on and extending this new and improved architecture established in SQL Server 2005. Thus, upgrading from SQL Server 2000 (or earlier) to either 2005 or 2008 are both "leaps forward" in terms of platform architecture, whereas upgrading from 2005 to 2008 is more of an incremental step in that regard. By that comparison, the 2008 release isn't really all "that big."

On the flip side, SQL Server 2008 can definitely be viewed as a major product release in its own right. Microsoft has loaded SQL Server 2008 with many exciting new features that bring unprecedented intelligence, convenience, and programming power down to the database level. SQL Server 2008 builds on the Common Language Runtime (CLR) integration introduced in SQL Server 2005 to usher in a new breed of native data types based on rich CLR user-defined types (UDTs), enabling hierarchical and geospatial capabilities in the database.

Many significant enhancements to Transact-SQL (T-SQL) have been added as well. Backup compression, data compression, and Change Data Capture (CDC) help us cope with rapidly growing data warehouses. New security features such as Transparent Data Encryption (TDE) and SQL Server Audit make it easier to meet the demands of increasingly stringent requirements for regulatory compliance. We also get native streaming capabilities between the database and the file system for enhanced binary large object (BLOB) storage and a new synchronization model for replication between occasionally connected client systems. There are also new BI features for analysis and reporting. Consider as well improvements in reliability,

availability, and scalability, and one could arguably maintain that SQL Server 2008 is in fact quite a big release. Our job isn't to convince you one way or the other—it's simply to help you understand and program the features you need to get the most out of Microsoft SQL Server 2008.

Note Recent survey polls continue to indicate that many sites are still running SQL Server 2000. To aid in the upgrade process for these sites, Microsoft supports a direct migration path from SQL Server 2000 to SQL Server 2008 (that is, without requiring an intermediate upgrade to SQL Server 2005 along the way).

Important Notification Services, which was available for SQL Server 2000 as a separate add-on and then later integrated into the core product in SQL Server 2005, has been discontinued in SQL Server 2008. (This is the only SQL Server 2005 component dropped by Microsoft in SQL Server 2008.)

According to Microsoft, Notification Services will continue to be supported as part of the SQL Server 2005 product support life cycle. Moving forward however, support for key notification scenarios will be incorporated into SQL Server Reporting Services. Some notification requirements are already addressed by existing Reporting Services features, such as standard and data driven subscriptions (discussed in Chapter 19). Features to support additional notification scenarios can be expected in future releases.

A Book *for* Developers

If you've gotten this far, we've already established that you're a developer (or, at least, that's one of the hats you wear). In tailoring the content of this book, there are a few other assumptions that we make about you as well.

First, we expect that you're already knowledgeable in relational database concepts—whether that experience is with SQL Server or non-Microsoft platforms. As such, you already know about tables, views, primary and foreign key relationships, stored procedures, functions, and triggers. These essentials are assumed knowledge and are not covered in this book. Similarly, we don't discuss proper relational design with respect to the rules of data normalization, strategic indexing practices, and other relational fundamentals. We also assume that you have at least basic familiarity with SQL statement syntax—again, either T-SQL in SQL Server or SQL dialects in other platforms.

With that baseline established, what do we assume that you want to learn about? Well, that would be just about everything else that a developer could want to learn about Microsoft SQL Server 2008! It starts with the most powerful extensions to T-SQL and the relational database engine. We also assume that you're thirsty for knowledge in wider spaces beyond

relational technologies, such as unstructured and semistructured data, client data access, security, and BI with data warehousing, analysis, and reporting.

As we began explaining, SQL Server 2005 was actually the groundbreaking release that redefined the product by incorporating a litany of features and services into the platform, while earlier versions had traditionally been focused on just the relational database engine. With respect to that fact, this book includes updated coverage of SQL Server 2005 enhancements as well as the very latest new features in SQL Server 2008. This makes it an ideal resource whether you are upgrading to SQL Server 2008 from any earlier version of SQL Server or from another relational database platform.

A Book *by* Developers

We, the authors and coauthors of this book, are first and foremost developers just like you. All of us have built careers in the pursuit of writing code that powers our applications, especially database applications. We've committed ourselves to building quality solutions that work with data and deliver that data in the most compelling ways to our businesses, partners, and customers. And, we love SQL Server!

Like SQL Server 2008 itself, this book builds on a previous edition written for SQL Server 2005, and it is the cumulative result of many years of work put in by many authors. We were fortunate enough to have a product manager from the Microsoft SQL Server product team contribute to the chapter on security. Our chapters on data mining, data warehousing, reporting, replication, and transactions were written by experts on those subjects as well.

Our approach has been to add value to the product's documentation by providing a developer-oriented investigation of the new and improved features and services in SQL Server 2008. As such, this book features an abundance of sample code, including a library of Visual Studio and SQL Server Management Studio sample projects that you can download from the book's companion Web site. (See the Introduction for details on downloading and using the sample code.)

A Book to Show You the Way

This book was carefully organized to present a potentially overwhelming array of new developer-oriented SQL Server 2008 features in the most coherent manner possible. To best achieve that, the chapters have been categorized into four primary sections, which are summarized at a high level in this overview.

Core Technologies

In Part I, we focus on core SQL Server technologies. These include enhancements to T-SQL, extended programmability with SQL CLR code in .NET languages such as Microsoft Visual Basic .NET and C#, server management, and security.

In Chapter 2, we explore the significant enhancements made to Transact-SQL (T-SQL)—which still remains the best programming tool for exploiting many new and old SQL Server features alike. We start with SQL Server 2005 enhancements, covering the ins and outs of writing recursive queries with common table expressions (CTEs) and examining scalar functions that provide the basis of ranking. We then go on to learn about exception handling and data definition language (DDL) triggers.

Then the chapter digs into the powerful extensions to T-SQL added in SQL Server 2008. Table-valued parameters (TVPs) allow entire result sets to be passed between stored procedures and functions on the server, as well as between client and server using Microsoft ADO.NET. New date and time features are then explored, including separate data types for dates and times, time zone awareness, and improvements in date and time range, storage, and precision. We then show many ways to use *MERGE*, a new data manipulation language (DML) statement that encapsulates all the individual operations typically involved in any merge scenario. From there, you'll learn about INSERT OVER DML, which enhances our ability to capture change data from the *OUTPUT* clause of any DML statement. Last, we look at *GROUPING SETS*, an extension to the traditional *GROUP BY* clause that increases our options for slicing and dicing data in aggregation queries.

Chapter 3 provides thorough coverage of SQL CLR programming—which lets you run compiled .NET code on SQL Server—as well as guidance on when and where you should put it to use. We go beyond mere stored procedures, triggers, and functions to explain and demonstrate the creation of CLR types and aggregates—entities that cannot be created *at all* in T-SQL. We also cover the different methods of creating SQL CLR objects in Microsoft Visual Studio 2008 and how to manage their deployment, both from Visual Studio and from T-SQL scripts in SQL Server Management Studio and elsewhere.

In Chapter 4, we show you how to conduct administrative tasks programmatically, using Server Management Objects (SMO) introduced in SQL Server 2005. You'll learn how to use SMO to perform database backups and restores, execute Database Consistency Check (DBCC) runs, and more, all from your own code. We'll also learn about the Policy-Based Management (PBM) feature, new in SQL Server 2008, which helps developers work with administrators to ensure that development and production machines comply with the same configuration defined through custom policies.

Chapter 5 discusses SQL Server security at length and examines your choices for keeping data safe and secure from prying eyes. We begin with the basic security concepts concerning logins, users, roles, authentication, and authorization. You then go on to learn about key-based

encryption support added in SQL Server 2005, which protects your data both while in transit and at rest. Important new security features added in SQL Server 2008 are then examined, which include Transparent Data Encryption (TDE) and SQL Server Audit. TDE allows you to encrypt entire databases in the background without special coding requirements. With SQL Server Audit, virtually any action taken by any user can be recorded for auditing in either the file system or the Windows event log. The chapter concludes by providing crucial guidance for adhering to best practices and avoiding common security pitfalls.

Beyond Relational

With the release of SQL Server 2008, Microsoft continues to redefine how we think of and use nonrelational data in the relational database world. One of the key release themes in SQL Server 2008 is "beyond relational," and by the time you complete Part II, you'll understand and appreciate the major strides Microsoft has made in this arena.

SQL Server 2005 embraced semistructured data by introducing the *xml* data type and a lot of rich XML support to go along with it. That innovation was an immeasurable improvement over the use of plain *varchar* or *text* columns to hold strings of XML (which was common practice in earlier versions of SQL Server), and thus revolutionized the storage of XML in the relational database. It empowered the development of database applications that work with hierarchical XML data *natively*—within the environment of the relational database system—something not previously possible using ordinary string columns. In Chapter 6, we cover the *xml* data type, XQuery extensions to T-SQL, server-side XML Schema Definition (XSD) collections, XML column indexing, XML enhancements in SQL Server 2008, and many other XML features available in SQL Server.

But native XML support was only the first step in a venture that Microsoft has pursued much more aggressively in SQL Server 2008, with new features added for handling a wider variety of nonrelational types, including unstructured data and spatial types. In the rest of Part II, we explore these new features and show how you can use them to build modern applications that demand unified services for storing and manipulating structured, semistructured, and unstructured data in the database.

As of SQL Server 2008, XML is no longer our only option for working with hierarchical data in the database. In Chapter 7, we explore the new *hierarchyid* data type that enables you to cast a hierarchical structure over any relational table. This data type is implemented as a "system CLR" type, which is nothing more really than a SQL CLR user-defined type (UDT), just like the ones we learned how to create for ourselves in Chapter 3 (except that you don't need to enable SQL CLR on the server in order to use *hierarchyid*). The value stored in a *hierarchyid* data type encodes the complete path of any given node in the tree structure, from the root down to the specific ordinal position among other sibling nodes sharing the same parent. Using methods provided by this new type, you can now efficiently build, query, and manipulate tree-structured data in your relational tables.

In Chapter 8, you'll learn all about the new FILESTREAM feature in SQL Server 2008, which greatly enhances the storage of unstructured BLOB data in the database—an increasingly common scenario given the accelerating data explosion of our times. Previously, we've had to choose between storing BLOB data in the database using *varbinary(max)* columns or outside the database as unstructured binary streams (typically, as files in the file system). Neither approach is without significant drawbacks—which is where FILESTREAM comes in. This highly efficient abstraction layer allows you to logically treat BLOB data as an integral part of the database, while SQL Server 2008 stores the BLOB data physically separate from the database in the NTFS file system behind the scenes. It will even seamlessly integrate database transactions with NTFS file system transactions against your BLOB data. Following the walkthroughs in this chapter, you'll learn how to leverage this powerful new feature from your .NET applications by building Windows, Web, and Windows Presentation Foundation (WPF) applications that use FILESTREAM for BLOB data storage.

Chapter 9 explores the new *geometry* and *geography* data types. These new system CLR types in SQL Server 2008 provide geospatial capabilities at the database level that make it easy for you to integrate location-awareness into your applications. Respectively, *geometry* and *geography* support spatial computations against the two basic geospatial surface models: planar (flat) and geodetic (round-earth). With geographical data (represented by coordinates) stored in these data types, you can easily determine intersections and calculate length, area, and distance measurements against that data. This chapter first quickly covers the basics and then provides walkthroughs in which we build several geospatial database applications, including one that integrates with Microsoft Virtual Earth. While this is a vast topic that could fill its own book, our chapter covers the fundamentals you'll need for working with geospatial data.

Reaching Out

After we've covered so much material about what you can do on the server and in your database, we move to Part III of the book, where we cover technologies and techniques more relevant to building applications that work with your databases and extend their reach.

We start with Chapter 10, which first covers Microsoft ADO.NET and the data access features of Microsoft Visual Studio, including typed *DataSet* objects. After this core coverage of ADO.NET, we provide an overview of new data access technologies, including the concepts and syntax of language-integrated query (LINQ). We'll look at LINQ To SQL and ADO.NET Entity Framework Object Relational Mapping (ORM) technologies, ADO.NET Data Services, and SQL Server Data Services.

In Chapter 11, we cover data binding, in *droves*. We start with Windows Forms using Visual Studio drag-and-drop binding. Next we look at ASP.NET data binding using designers, code, and markup techniques. We then move on to using ASP.NET Asynchronous JavaScript and

XML (AJAX) for data presentation, and we introduce the new ASP.NET Dynamic Data feature set. We finish with data binding in WPF and Silverlight 2.0, showing you how they compare to each other and to the other data binding models too. Regardless of your application type, we'll help you manage your data with ease.

No matter how you write and package your code—whether it be in T-SQL or a .NET language; exposed as a conventional stored procedure, function, or Web service; or deployed to the client or the server—you must keep your data consistent to ensure its integrity. The key to consistency is transactions, and as with other SQL Server programmability features, transactions can be managed from a variety of places. If you're writing T-SQL code or you're writing client code using the ADO.NET *SqlClient* provider or *System.Transactions*, you need to be aware of the various transaction isolation levels supported by SQL Server, the appropriate scope of your transactions, and best practices for writing transactional code. In Chapter 12, we get you there.

We couldn't round out the reach story without covering merge replication and synchronization between distributed database environments. We start Chapter 13 with a walkthrough for creating a synchronized client/server database application using conventional SQL Server Merge Replication. We then examine a new feature in SQL Server 2008 called Change Tracking, designed to work in tandem with the new Sync Services for ADO.NET—features that together enhance SQL Server Compact 3.5 applications running on an ever-increasing number of Windows-based mobile devices. These devices are used not just by consumers, but also by the mobile workforce—people who need ready access to their data no matter where they are, in both wireless online and offline settings. Mobile applications frequently alternate between connected and disconnected states. You'll learn how Merge Replication and Change Tracking with Sync Services for ADO.NET make it possible to implement mobile client and Windows desktop applications that work in offline mode, and seamlessly synchronize their data whenever a connection to the server is made available.

Business Intelligence Strategies

In Part IV of this book, we help you take your broad-based but tactical database management and programming knowledge and extend it to the realm of strategic analysis. Specifically, we teach you how to build a true data warehouse and then capitalize on your data warehouse through OLAP, data mining, and reporting.

In Chapter 14, we show you the ropes of data warehousing and explain how to use several new important SQL Server 2008 features designed specifically to help you work better with the data warehouses that will back your business intelligence (BI). The chapter begins by providing some important background, design guidance, and practical advice for building data warehouses. We then move on to provide hands-on coverage of new and improved SQL Server features that facilitate the process. These include applied use of the *MERGE* statement (which we also cover in Chapter 2), Change Data Capture (CDC), data and backup

compression, and more—all of which are new in SQL Server 2008. CDC allows you to capture change data on CDC-enabled tables without resorting to triggers or code changes. This in turn facilitates the ETL processes that bring large data warehouses up-to-date incrementally. And as data warehouses continue to grow larger than ever before, data and backup compression are vital and welcome indeed. Other new and advanced data warehousing features covered in this chapter include partitioned table parallelism, star-join query optimization, and sparse columns.

In the remaining chapters, we show you how to use these transformed data repositories as the basis for sophisticated SQL Server Analysis Services databases that support cutting-edge BI features. Many books on SQL Server exclude coverage of Analysis Services on the grounds that it is a "specialized" subject, but we respectfully disagree with that notion. The very premise of SQL Server 2008 Analysis Services and its unified dimensional model paradigm is the *mainstream* appeal and accessibility of BI. We show you how easy BI programming can be and how powerfully it complements conventional relational databases and conventional database programming.

In Chapter 15, we take you through the basics of designing OLAP cubes using the Analysis Services project designers in Visual Studio. We show you how to build, deploy, and query OLAP cubes that support actionable, drill-down analysis of your data. We kept this chapter fairly short to provide a sort of "quick start" approach to BI for busy developers.

In Chapter 16, we take the basic cube we built in Chapter 15 and use Visual Studio designers to implement an array of new OLAP features brought to you by Analysis Services 2008. By the end of the chapter, your cube will have a number of the features underlying Microsoft's unified dimensional model. The chapter is long, but you can read it at your own pace, immediately mastering new features as you read each section.

In Chapter 17, we provide comprehensive coverage of a host of OLAP application development techniques. We cover the creation of OLAP user interfaces (UIs) in Microsoft Office Excel 2007, using PivotTables, charts, and new in-cell *CUBE* formulas. We then cover how to publish these assets to Web dashboards using Excel Services, a component of Microsoft Office SharePoint Server.

We provide a basic tutorial on multidimensional expression language (MDX) queries and show you how to run MDX queries from the SQL Server Management Studio MDX query window. We also show you how to run MDX queries from your own applications through application programming interface (API)–level programming with ADO MD.NET. We cover management of Analysis Services databases, both interactively using SQL Server Management Studio and programmatically using Analysis Management Objects (AMO). We introduce you to the Web Services–based XML for Analysis (XMLA) standard on which both ADO MD.NET and AMO are built, showing you how to create XMLA scripts in Management Studio and manipulate XMLA programmatically in .NET code. We also introduce you to Analysis Services' own .NET CLR integration, and show you how to create managed stored procedures in .NET using AMO and server-side ADO MD.NET.

Chapter 18 is all about data mining, and it provides a self-contained, end-to-end treatment of the topic, including the newest data mining features added in SQL Server 2008. We start with a conceptual introduction, and then we provide a tutorial on designing, training, browsing, and deploying your mining structures and models in Visual Studio. We then switch to SQL Server Management Studio, showing you how to manage your mining structures and models interactively, using the graphical user interface (GUI) tools provided, and programmatically, through SQL Data Mining Extensions (DMX) scripts. We then head back to .NET programming, showing you how to embed the Analysis Services Data Mining Model Browsers into your applications; how to use DMX, ADO.NET, and data binding together to build compelling Windows Forms and ASP.NET data mining applications; and how to use server-side ADO MD.NET to build DMX stored procedures. We finish by showing you how to leverage the powerful new data mining add-ins for Excel 2007.

Chapter 19 covers Reporting Services, which has been enhanced significantly in SQL Server 2008. We start with a tutorial on the new Report Designer in Visual Studio 2008 and then show you how to use Reporting Services to quickly and easily build sophisticated reports against both relational and OLAP databases. You'll see how to create full-scale reporting information systems that expose all the information in the feature-packed databases you've learned to build, maintain, extend, and develop against in the rest of the book. We'll show you how to deliver reports with the flexible layouts users want by using the new *tablix* data region (a hybrid of *table* and *matrix*) in your reports. Next we give you an overview of the report server configuration and administration tools, and we teach you how to deploy the reports for your users and how to embed reports into your Windows Forms and ASP.NET client applications. We also show you how to use the management and execution Web Services exposed by Reporting Services to programmatically integrate reporting and report administration into your custom applications and deployment scripts.

Summary

In this opening chapter, we compared the Microsoft SQL Server 2008 release with earlier product versions, and we discussed the wide range of programmability features at our disposal. In the process, we outlined the various chapters and how they are organized, accompanied with an overview of the extensive SQL Server 2008 product feature set for developers that you'll learn about throughout the rest of this book. Given the broad range of capabilities in that feature set, by no means does this book need to be read in any particular order. Read it from start to finish if you want, or jump right to the chapters that are most relevant for your needs. Either way, you'll find the practical guidance and information you need to get your job done.

Chapter 2

T-SQL Enhancements

—Stephen Forte and Leonard Lobel

By now, you must have heard that you can write Microsoft SQL Server stored procedures in any language that uses the common language runtime (CLR), such as Microsoft Visual C# or Microsoft Visual Basic .NET. This is great news if you've never mastered Transact-SQL (T-SQL), right? We hate to be the bearer of bad news, but CLR stored procedures are not a cure-all for your SQL Server programming challenges. As you'll see in Chapter 3, writing a stored procedure in a language that uses the CLR is useful for a number of database programming dilemmas—for instance, CLR stored procedures are often a good replacement for extended stored procedures in earlier versions of SQL Server. For almost everything else, you will want to use T-SQL.

In fact, reports of T-SQL's death have been greatly exaggerated. In most cases, using T-SQL for your queries and stored procedures is more efficient than writing CLR stored procedures.

Given that T-SQL is alive and well, let's look at how it has changed since SQL Server 2000. T-SQL has been improved in many ways. In this chapter, we'll begin by exploring the most notable changes in T-SQL introduced in SQL Server 2005, including the following:

- Common table expressions (CTEs)
- The *PIVOT* and *UNPIVOT* operators
- The *APPLY* operator
- Enhancements to the *TOP* parameter
- Ranking functions
- Exception handling using *TRY* and *CATCH* blocks
- The *varchar(max)* data type
- The *WAITFOR* statement
- Data definition language (DDL) triggers
- The SNAPSHOT isolation level

We'll continue with an in-depth look at these significant T-SQL features, which are new in SQL Server 2008:

- Table-valued parameters (TVPs)
- New date and time data types

- The *MERGE* statement
- The INSERT OVER DML syntax
- The *GROUPING SETS* operator
- New T-SQL shorthand syntax

T-SQL provides several important "beyond relational" data types as well, such as the *xml*, *hierarchyid*, *varbinary(max) FILESTREAM*, *geography*, and *geometry* data types. Chapters 6 though 9 in Part II provide in-depth coverage of these special SQL Server 2008 data types.

Common Table Expressions

A common table expression (CTE) closely resembles a nonpersistent view. It is a temporary named result set that you define in your query and that will be used by the *FROM* clause of the query. Each CTE is defined only once (but can be referred to as many times as necessary while still in scope) and lives for as long as the query lives. You can use CTEs to perform recursive operations. Here is the syntax to create a CTE:

```
WITH <name of your CTE>(<column names>)
AS
(
<actual query>
)

SELECT * FROM <name of your CTE>
```

Note Many of our examples in this chapter use the *AdventureWorks2008* sample database. To run these examples, you will need to download and install the sample database on your machine. The book's Introduction provides details and instructions for obtaining the *AdventureWorks2008* sample database.

An example of a simple CTE using *AdventureWorks2008* is shown in Listing 2-1.

LISTING 2-1 A simple CTE

```
USE AdventureWorks2008
GO
WITH AllMRContacts
AS
(
  SELECT * FROM Person.Person WHERE Title = 'Mr.'
)
SELECT LastName + ', ' + FirstName AS Contact
 FROM AllMRContacts
 ORDER BY LastName
```

The results are shown here:

```
Contact
-------------------
Abbas, Syed
Achong, Gustavo
Adams, Jay
Adams, Ben
Adina, Ronald
Agcaoili, Samuel
 ...
```

The following example gets a count of all the sales a salesperson made in the *AdventureWorks2008* orders system as a CTE and then executes a simple inner join with the *SalesPerson* table to return more information about the salesperson, such as his or her quota. This demonstrates how a CTE is joined to your calling query. You can do this without a CTE, but think about all the times you have created a temp table or a throwaway view and joined back to it—now you can use a CTE instead and keep the complexity of aggregating in the CTE only, thereby simplifying your code. The code is shown in Listing 2-2.

LISTING 2-2 CTE-to-query join

```
WITH OrderCountCTE(SalesPersonID, OrderCount)
AS
(
  SELECT SalesPersonID, COUNT(*)
   FROM  Sales.SalesOrderHeader
   WHERE SalesPersonID IS NOT NULL
   GROUP BY SalesPersonID
)
SELECT
  sp.BusinessEntityID,
  FirstName + ' ' + LastName as SalesPerson,
  oc.OrderCount,
  sp.SalesYTD
 FROM Sales.vSalesPerson AS sp
  INNER JOIN OrderCountCTE AS oc ON oc.SalesPersonID = sp.BusinessEntityID
 ORDER BY oc.OrderCount DESC
```

The results look like this:

```
BusinessEntityID  SalesPerson              OrderCount  SalesYTD
----------------- ------------------------ ----------- --------------
277               Jillian Carson           473         3857163.6332
275               Michael Blythe           450         4557045.0459
279               Tsvi Reiter              429         2811012.7151
276               Linda Mitchell           418         5200475.2313
289               Jae Pak                  348         5015682.3752
282               José Saraiva             271         3189356.2465
281               Shu Ito                  242         3018725.4858
278               Garrett Vargas           234         1764938.9859
```

```
283             David Campbell          189         3587378.4257
290             Ranjit Varkey Chudukatil 175        3827950.238
284             Tete Mensa-Annan        140         1931620.1835
288             Rachel Valdez           130         2241204.0424
286             Lynn Tsoflias           109         1758385.926
280             Pamela Ansman-Wolfe     95          0.00
274             Stephen Jiang           48          677558.4653
287             Amy Alberts             39          636440.251
285             Syed Abbas              16          219088.8836
 :
```

CTEs can also eliminate self-joins in some of your queries. Take a look at the example in Listing 2-3. We will create a table named *Products* and insert duplicates into the *ProductName* column.

LISTING 2-3 Inserting duplicates into the *AdventureWorks2008 ProductName* column

```
CREATE TABLE Products
 (ProductID int NOT NULL,
  ProductName varchar(25),
  Price money NULL,
  CONSTRAINT PK_Products PRIMARY KEY NONCLUSTERED (ProductID)
 )
GO
INSERT INTO Products VALUES (1, 'Widgets', 25)
INSERT INTO Products VALUES (2, 'Gadgets', 50)
INSERT INTO Products VALUES (3, 'Thingies', 75)
INSERT INTO Products VALUES (4, 'Whoozits', 90)
INSERT INTO Products VALUES (5, 'Whatzits', 5)
INSERT INTO Products VALUES (6, 'Gizmos', 15)
INSERT INTO Products VALUES (7, 'Widgets', 24)
INSERT INTO Products VALUES (8, 'Gizmos', 36)
INSERT INTO Products VALUES (9, 'Gizmos', 36)
```

One common problem found in databases is having duplicate product names with different product IDs. If you run a duplicate-finding query, that query will return all the records (the duplicates and the good values). This increases the difficulty of automatically deleting duplicates. If you want to find the *ProductName* duplicates without also including the first instance of the name in the table, you can use a self-join, as shown in Listing 2-4.

LISTING 2-4 Self-join without CTE

```
SELECT * FROM Products WHERE ProductID NOT IN
(SELECT MIN(ProductID) FROM Products AS P
 WHERE Products.ProductName = P.ProductName)
```

The self-join returns data like this:

```
ProductID ProductName Price
--------- ----------- -----------
8         Gizmos      36.00
9         Gizmos      36.00
7         Widgets     24.00
```

You can also rewrite your query using a CTE to eliminate the confusing-looking self-join and get the same results. This technique does not offer a performance gain over self-joins; it is just a convenience for code maintainability. The preceding self-join example is rewritten in Listing 2-5 as a CTE and yields the same results; notice that we are joining our CTE with the *Products* table.

LISTING 2-5 Self-join as a CTE

```
WITH MinProductRecords AS
(
  SELECT MIN(ProductID) AS ProductID, ProductName
   FROM Products
   GROUP BY ProductName
   HAVING COUNT(*) > 1
)
SELECT P.*
 FROM Products AS P
  INNER JOIN MinProductRecords AS MP
  ON P.ProductName = MP.ProductName AND P.ProductID > MP.ProductID
```

After you investigate your duplicates using the preceding CTE, you might want to delete the duplicate data. You might also want to update any foreign keys in related tables to use the original *ProductID* value. If your duplicate data does not have any related child rows in another table, or if you have updated them to the correct *ProductID*, you can delete the duplicate data by just rewriting the CTE, as shown in Listing 2-6, replacing the *SELECT* * with a *DELETE*.

LISTING 2-6 Deleting duplicates in a CTE

```
WITH MinProductRecords AS
(
  SELECT MIN(ProductID) AS ProductID, ProductName
   FROM Products
   GROUP BY ProductName
   HAVING COUNT(*) > 1
)
DELETE Products
 FROM Products AS P
  INNER JOIN MinProductRecords AS MP
  ON P.ProductName = MP.ProductName AND P.ProductID > MP.ProductID
```

Creating Recursive Queries with CTEs

The true power of CTEs emerges when you use them recursively to perform hierarchical queries on tree-structured data. In fact, this was a major reason that Microsoft built CTEs, in addition to ANSI SQL-92 compliance. A recursive CTE is constructed from a minimum of two queries. The first, the anchor member, is a nonrecursive query; the second, the recursive member, is the recursive query. Within your CTE's parentheses (after the *AS* clause), you define queries that are independent or refer back to the same CTE. The anchor and recursive members are separated by a *UNION ALL* statement. Anchor members are invoked only once; recursive members are invoked repeatedly until the query returns no rows. You can append multiple anchor members to one another using a *UNION* or *UNION ALL* operator, depending on whether you want to eliminate duplicates. (You must append recursive members using a *UNION ALL* operator.) Here is the syntax:

```
WITH SimpleRecursive(field names)
AS
(
   <Select Statement for the Anchor Member>

   UNION ALL

   <Select Statement for the Recursive Member>
)

SELECT * FROM SimpleRecursive
```

The example in Listing 2-7 demonstrates this feature. We'll create a table of employees and a self-referencing field back to *EmployeeID* named *ReportsTo*. We'll then write a query that returns all the employees who report to Stephen (*EmployeeID=2*) and all the employees who report to Stephen's subordinates.

LISTING 2-7 Example table for recursive CTE queries

```
CREATE TABLE EmployeeTree
 (EmployeeID int PRIMARY KEY,
  EmployeeName nvarchar(50),
  ReportsTo int)
GO

--insert some data, build a reporting tree
INSERT INTO EmployeeTree VALUES(1, 'Richard', NULL)
INSERT INTO EmployeeTree VALUES(2, 'Stephen', 1)
INSERT INTO EmployeeTree VALUES(3, 'Clemens', 2)
INSERT INTO EmployeeTree VALUES(4, 'Malek', 2)
INSERT INTO EmployeeTree VALUES(5, 'Goksin', 4)
INSERT INTO EmployeeTree VALUES(6, 'Kimberly', 1)
INSERT INTO EmployeeTree VALUES(7, 'Ramesh', 5)
```

Listing 2-8 shows the recursive query to determine which employees report to Stephen.

LISTING 2-8 Recursive CTE query

```
WITH SimpleRecursive(EmployeeID, EmployeeName, ReportsTo)
AS
(
  SELECT EmployeeID, EmployeeName, ReportsTo
   FROM EmployeeTree WHERE EmployeeID = 2
  UNION ALL
  SELECT p.EmployeeID, p.EmployeeName, p.ReportsTo
   FROM EmployeeTree AS P
    INNER JOIN SimpleRecursive A ON A.EmployeeID = P.ReportsTo
)
SELECT sr.EmployeeName AS Employee, et.EmployeeName AS Boss
 FROM SimpleRecursive AS sr
  INNER JOIN EmployeeTree AS et ON sr.ReportsTo = et.EmployeeID
```

Here are the results:

```
Employee    Boss
----------- ------------
Stephen     Richard
Clemens     Stephen
Malek       Stephen
Goksin      Malek
Ramesh      Goskin
```

This recursion starts where *EmployeeID = 2* (the anchor member or the first *SELECT*). It picks up that record and then, using the recursive member (the *SELECT* after the *UNION ALL*), picks up all the records that report to Stephen and that record's children. (Goksin reports to Malek, and Malek reports to Stephen.) Each subsequent recursion tries to find more children that have as parents the employees found by the previous recursion. Eventually, the recursion returns no results, and that is what causes the recursion to stop (the reason why Kimberly is not returned). If the anchor member is changed to *EmployeeID = 1*, Kimberly will also be returned in the results.

By design, the recursive member keeps looking for children and can cycle on indefinitely. If you suspect many cycles will occur and want to limit the number of recursive invocations, you can specify the *MAXRECURSION* option right after the outer query using the *OPTION* clause.

```
OPTION(MAXRECURSION 25)
```

This option causes SQL Server to raise an error when the CTE exceeds the specified limit. By default, the limit is 100 (if you've omitted the option). To specify no option, you must set *MAXRECURSION* to *0*. You can also run the same query to find direct reports and subordinates only one level deep (including direct reports Clemens and Malek and Malek's subordinate Goksin but skipping Ramesh, who is three levels deep), as shown in Listing 2-9.

LISTING 2-9 Recursive query with *MAXRECURSION*

```
WITH SimpleRecursive(EmployeeID, EmployeeName, ReportsTo)
AS
(
  SELECT EmployeeID, EmployeeName, ReportsTo
   FROM EmployeeTree WHERE EmployeeID = 2
  UNION ALL
  SELECT p.EmployeeID, p.EmployeeName, p.ReportsTo
   FROM EmployeeTree AS P
    INNER JOIN SimpleRecursive A ON A.EmployeeID = P.ReportsTo
)
SELECT sr.EmployeeName AS Employee, et.EmployeeName AS Boss
 FROM SimpleRecursive AS sr
  INNER JOIN EmployeeTree AS et ON sr.ReportsTo = et.EmployeeID
OPTION(MAXRECURSION 2)
```

Here are the results:

```
Employee   Boss
---------- ------------
Stephen    Richard
Clemens    Stephen
Malek      Stephen
Goksin     Malek
```

You will also see that the query raises the following error message:

```
Msg 530, Level 16, State 1, Line 2
The statement terminated. The maximum recursion 2 has been exhausted
before statement completion.
```

One way to avoid the exception is to use a generated column to keep track of the level you are on and include that in the *WHERE* clause instead of using *MAXRECURSION*. The revised example in Listing 2-10 returns the same data as the preceding example but without the error.

LISTING 2-10 Controlling recursion without *MAXRECURSION*

```
WITH SimpleRecursive(EmployeeID, EmployeeName, ReportsTo, SubLevel)
AS
(
  SELECT EmployeeID, EmployeeName, ReportsTo, 0
   FROM EmployeeTree WHERE EmployeeID = 2
  UNION ALL
  SELECT p.EmployeeID, p.EmployeeName, p.ReportsTo, SubLevel + 1
   FROM EmployeeTree AS P
    INNER JOIN SimpleRecursive A ON A.EmployeeID = P.ReportsTo
    WHERE SubLevel <= 2
)
SELECT sr.EmployeeName AS Employee, et.EmployeeName AS Boss
 FROM SimpleRecursive sr
  INNER JOIN EmployeeTree AS et ON sr.ReportsTo = et.EmployeeID
```

Note SQL Server 2008 introduces a new *hierarchyid* data type that can implement a more robust tree structure over a recursive self-joining table than the example we've shown here. The *hierarchyid* data type is covered in Chapter 7.

The *PIVOT* and *UNPIVOT* Operators

Let's face it—users usually want to see data in tabular format, which is a bit of a challenge given that data in SQL Server is most often stored in a highly relational form. *PIVOT* is a T-SQL operator that you can specify in your *FROM* clause to rotate rows into columns and create a traditional crosstab query.

Using *PIVOT* is easy. In your *SELECT* statement, you specify the values you want to pivot on. The following example in the *AdventureWorks2008* database uses the order years (calculated using the *DatePart* function) as the columns. The *FROM* clause looks normal except for the *PIVOT* statement. This statement creates the value you want to show in the rows of the newly created columns. This example uses the aggregate *SUM* of *TotalDue* (a calculated field in the *FROM* clause). Then we use the *FOR* operator to list the values we want to pivot on in the *OrderYear* column. The example is shown in Listing 2-11.

LISTING 2-11 Creating tabular results with the *PIVOT* operator

```
SELECT
  CustomerID,
  [2001] AS Y2001, [2002] AS Y2002, [2003] AS Y2003, [2004] AS Y2004
 FROM
 (
    SELECT CustomerID, DATEPART(yyyy, OrderDate) AS OrderYear, TotalDue
     FROM Sales.SalesOrderHeader
 ) AS piv
PIVOT
(
  SUM(TotalDue) FOR OrderYear IN([2001], [2002], [2003], [2004])
) AS child
ORDER BY CustomerID
```

Here are the results:

```
CustomerID  Y2001      Y2002       Y2003        Y2004
----------- ---------- ----------- ------------ ------------
1           40732.6067 72366.1284  NULL         NULL
2           NULL1      5653.6715   12118.0275   4962.2705
3           39752.8421 168393.7021 219434.4265  51925.3549
4           NULL       263025.3113 373484.299   143525.6018
5           NULL       33370.6901  60206.9999   20641.1106
6           NULL       NULL        668.4861     2979.3473
```

```
7           NULL       6651.036    3718.7804    NULL
8           NULL       NULL        19439.2466   10900.0347
9           NULL       320.6283    11401.5975   5282.8652
10          NULL       96701.7401  291472.2172  204525.9634
...
```

That's all there is to it. Of course, this example is simplified to show you the concept; other, more sophisticated, aggregates are possible, and you can even use CTEs in the *FROM* clause. In any case, using *PIVOT* is simple.

Using *UNPIVOT*

You can use the *UNPIVOT* operator to normalize data that is already pivoted. For example, suppose you obtain pivoted data that shows, for each vendor, the number of orders placed by each employee. The code in Listing 2-12 creates such a table.

LISTING 2-12 Example table containing pivoted data

```
CREATE TABLE VendorEmployee
 (VendorID int,
  Emp1Orders int,
  Emp2Orders int,
  Emp3Orders int,
  Emp4Orders int,
  Emp5Orders int)
GO

INSERT INTO VendorEmployee VALUES(1, 4, 3, 5, 4, 4)
INSERT INTO VendorEmployee VALUES(2, 4, 1, 5, 5, 5)
INSERT INTO VendorEmployee VALUES(3, 4, 3, 5, 4, 4)
INSERT INTO VendorEmployee VALUES(4, 4, 2, 5, 4, 4)
INSERT INTO VendorEmployee VALUES(5, 5, 1, 5, 5, 5)
```

Our table looks like this:

```
VendorID    Emp1Orders  Emp2Orders  Emp3Orders  Emp4Orders  Emp5Orders
----------- ----------- ----------- ----------- ----------- -----------
1           4           3           5           4           4
2           4           1           5           5           5
3           4           3           5           4           4
4           4           2           5           4           4
5           5           1           5           5           5
```

You might want to unpivot the data to display columns for vendor ID, employee, and number of orders. Listing 2-13 shows how to use the *UNPIVOT* operator to achieve this goal.

LISTING 2-13 Using the *UNPIVOT* operator

```
SELECT VendorId, Employee, Orders AS NumberOfOrders
 FROM
  (SELECT VendorId, Emp1Orders, Emp2Orders, Emp3Orders, Emp4Orders, Emp5Orders
    FROM VendorEmployee
  ) AS p
UNPIVOT
(
  Orders FOR Employee IN
   (Emp1Orders, Emp2Orders, Emp3Orders, Emp4Orders, Emp5Orders)
) AS unpvt
```

Here are the results:

```
VendorID  Employee     NumberOfOrders
--------- ------------ ----------------------
1         Emp1Orders   4
1         Emp2Orders   3
1         Emp3Orders   5
1         Emp4Orders   4
1         Emp5Orders   4
2         Emp1Orders   4
...
```

Dynamically Pivoting Columns

The problem with *PIVOT* is the same problem with *CASE* and other methods: you have to specify the columns. Consider the code in Listing 2-14.

LISTING 2-14 Statically driven *PIVOT*

```
SELECT *
 FROM (SELECT CustomerID, YEAR(OrderDate) AS OrderYear, TotalDue
        FROM Sales.SalesOrderHeader) AS header
PIVOT
(
  SUM(TotalDue) FOR orderyear IN([2002],[2003],[2004])
) AS piv
```

The results show us a nice crosstab query with the years displayed as columns:

```
CustomerID  2002       2003       2004
----------- ---------- ---------- -----------
14324       NULL       2264.2536  3394.9247
22814       NULL       5.514      NULL
11407       NULL       59.659     NULL
28387       NULL       NULL       645.2869
19897       NULL       NULL       659.6408
15675       2699.9018  2682.9953  2580.1529
24165       NULL       2699.9018  666.8565
...
```

Because this data goes only up to 2004, what happens when you add 2005 to the data? Do you want to go into all your queries and add the current year to the *IN* clause? We can accommodate new years in the data by dynamically building the *IN* clause and then programmatically writing the entire SQL statement. Once you have dynamically written the SQL statement, you can execute it using *sp_executesql*, as shown in Listing 2-15. Since all we have to do is generate dynamically the *IN* clause, creating a dynamic *PIVOT* in SQL Server is much easier than creating a dynamic *CASE* statement. The results are exactly the same as those shown following Listing 2-14, except that as new yearly data is added to the table, the query dynamically adds the column for it. Remember that your reporting engine will most likely not accommodate the new dynamic columns, but data-bound controls will.

LISTING 2-15 Dynamically driven *PIVOT*

```
DECLARE @tblOrderDate AS TABLE(y int NOT NULL PRIMARY KEY)
INSERT INTO @tblOrderDate
 SELECT DISTINCT YEAR(OrderDate) FROM Sales.SalesOrderHeader

DECLARE @cols AS nvarchar(max)
DECLARE @years AS int
SET @years = (SELECT MIN(y) FROM @tblOrderDate)
SET @cols = N''
WHILE @years IS NOT NULL BEGIN
  SET @cols = @cols + N',[' + CAST(@years AS nvarchar(max)) + N']'
  SET @years = (SELECT MIN(y) FROM @tblOrderDate WHERE y > @years)
END
SET @cols = SUBSTRING(@cols, 2, LEN(@cols))
DECLARE @sql AS nvarchar(max)
SET @sql = '
  SELECT *
   FROM
    (SELECT CustomerId, YEAR(OrderDate) AS OrderYear, TotalDue
      FROM Sales.SalesOrderHeader
    ) AS a
    PIVOT
    (
      SUM(TotalDue) FOR OrderYear IN(' + @cols + N')
    ) AS b
'
PRINT @sql -- for debugging
EXEC sp_executesql @sql
```

You can accomplish the same results using the newer CTE syntax instead of using the *table* variable, as shown in Listing 2-16.

LISTING 2-16 Dynamically driven *PIVOT* using a CTE

```
DECLARE @cols AS nvarchar(MAX)

WITH YearsCTE AS
 (SELECT DISTINCT YEAR(OrderDate) as [Year] FROM Sales.SalesOrderHeader)

SELECT @cols = ISNULL(@cols + ',[', '[') + CAST([YEAR] AS nvarchar(10)) + ']'
 FROM YearsCTE
 ORDER BY [YEAR]

-- Construct the full T-SQL statement and execute it dynamically.
DECLARE @sql AS nvarchar(MAX)
SET @sql = '
  SELECT *
   FROM
    (SELECT CustomerId, YEAR(OrderDate) AS OrderYear, TotalDue
      FROM Sales.SalesOrderHeader
    ) AS a
    PIVOT
    (
      SUM(TotalDue) FOR OrderYear IN(' + @cols + N')
    ) AS b
'
PRINT @sql -- for debugging
EXEC sp_executesql @sql
```

The *APPLY* Operator

APPLY is an operator that you specify in the *FROM* clause of a query. It enables you to invoke a table-valued function (TVF) for each row of an outer table. The flexibility of *APPLY* is evident when you use the outer table's columns as your function's arguments. The *APPLY* operator has two forms: *CROSS APPLY* and *OUTER APPLY*. *CROSS APPLY* doesn't return the outer table's row if the TVF returns an empty set for it; the *OUTER APPLY* returns a row with *NULL* values instead of the function's columns.

To see how *APPLY* works, we'll first create a TVF that returns a table. Listing 2-17 shows a simple function that returns as a table the top *n* rows for a customer from the *SalesOrderHeader* table.

LISTING 2-17 Returning a table

```
CREATE FUNCTION fnGetCustomerOrders(@CustomerID int, @TopRecords bigint)
RETURNS TABLE
 AS RETURN
  SELECT TOP (@TopRecords) *
   FROM Sales.SalesOrderHeader
   WHERE CustomerID = @CustomerID
   ORDER BY OrderDate DESC
```

After creating the *fnGetCustomerOrders* TVF, we call it from the query, as shown in Listing 2-18.

LISTING 2-18 Executing a query with *APPLY*

```
SELECT * FROM Sales.Customer cust
 CROSS APPLY fnGetCustomerOrders(CustomerID, 100)
```

This query returns all the records from the *Customers* table and then, as additional fields, the records from the *Orders* table (by way of the *fnGetCustomerOrders* function) that match for the customer ID because that's what's being passed in dynamically as the first argument to *fnGetCustomerOrders*. Because we passed the value 100 for the second parameter, rows for up to the first 100 orders per customer are generated and returned by this query.

TOP Enhancements

In SQL Server 2000 and earlier versions, *TOP* allows you to limit the number of rows returned as a number or a percentage in *SELECT* queries. As of SQL Server 2005, you can use *TOP* in *DELETE*, *UPDATE*, and *INSERT* queries and can also specify the number (or percentage) of rows by using variables or any valid numeric returning expression (such as a subquery). The main reason for allowing *TOP* with *DELETE*, *UPDATE*, and *INSERT* was to replace the *SET ROWCOUNT* option, which SQL Server traditionally didn't optimize very well.

You can specify the *TOP* limit as a literal number or an expression. If you're using an expression, you must enclose it in parentheses. The expression should be of the *bigint* data type when you are not using the *PERCENT* option and a *float* value in the range 0 through 100 when you are using the *PERCENT* option. You might find it useful to create an expression for *TOP* and make it a parameter that you pass in to a stored procedure, as shown in Listing 2-19.

LISTING 2-19 Using *TOP* enhancements in a stored procedure

```
CREATE PROCEDURE uspReturnTopOrders(@NumberOfRows bigint)
AS
  SELECT TOP (@NumberOfRows) SalesOrderID
   FROM Sales.SalesOrderHeader
   ORDER BY SalesOrderID
```

Executing the stored procedure is easy. Just pass in the number of records you want (in this case, 100), as shown here:

```
EXEC uspReturnTopOrders @NumberOfRows = 100
```

Here are the results:

```
SalesOrderID
------------
43659
43660
43661
43662
43663
 :

(100 row(s) affected)
```

Using a subquery can be powerful when you're doing things on the fly. The following example shows how to get the *TOP n* orders based on how many rows are in the *SalesPerson* table:

```
SELECT TOP (SELECT COUNT(*) FROM Sales.SalesPerson)
  SalesOrderID, RevisionNumber, OrderDate
 FROM Sales.SalesOrderHeader
 ORDER BY SalesOrderID
```

Because there are 17 rows in the *SalesPerson* table, the query returns only the top 17 rows from the *SalesOrderHeader* table:

```
SalesOrderID  Revision  NumberOrderDate
------------- --------- -------------------------
43659         1         2001-07-01 00:00:00.000
43660         1         2001-07-01 00:00:00.000
43661         1         2001-07-01 00:00:00.000
43662         1         2001-07-01 00:00:00.000
43663         1         2001-07-01 00:00:00.000
43664         1         2001-07-01 00:00:00.000
43665         1         2001-07-01 00:00:00.000
43666         1         2001-07-01 00:00:00.000
43667         1         2001-07-01 00:00:00.000
43668         1         2001-07-01 00:00:00.000
43669         1         2001-07-01 00:00:00.000
43670         1         2001-07-01 00:00:00.000
43671         1         2001-07-01 00:00:00.000
43672         1         2001-07-01 00:00:00.000
43673         1         2001-07-01 00:00:00.000
43674         1         2001-07-01 00:00:00.000
43675         1         2001-07-01 00:00:00.000

(17 row(s) affected)
```

Using the *PERCENT* option is just as easy. Just add the *PERCENT* keyword, and make sure that your variable is a *float*. In Listing 2-20, we're asking for the top 10 percent, so we'll get back 3,147 records because the *AdventureWorks2008 SalesOrderHeader* table has approximately 31,465 records in it.

LISTING 2-20 Returning *TOP* percentages

```
DECLARE @NumberOfRows AS float
SET @NumberOfRows = 10

SELECT TOP (@NumberOfRows) PERCENT *
 FROM Sales.SalesOrderHeader
 ORDER BY OrderDate
```

Ranking Functions

Databases hold data. Users sometimes want to perform simple calculations or algorithms on that data to rank the results in a specific order—like gold, silver, and bronze medals in the Olympics or the top 10 customers by region. Starting with SQL Server 2005, functionality is provided for using ranking expressions with your result set. You can select a number of ranking algorithms, which are then applied to a column that you specify and applied in the scope of the executing query. If the data changes, your ranking algorithm will return different data the next time it is run. This comes in handy in Microsoft .NET Framework applications for paging and sorting in a grid, as well as in many other scenarios.

The *ROW_NUMBER* Function

The most basic ranking function is *ROW_NUMBER*. It returns a column as an expression that contains the row's number in the result set. This number is used only in the context of the result set; if the result changes, the *ROW_NUMBER* changes. The *ROW_NUMBER* expression takes an *ORDER BY* statement with the column you want to use for the row count and the *OVER* operator, which links the *ORDER BY* to the specific ranking function you are using. The *ORDER BY* in the *OVER* clause replaces an *ORDER BY* at the end of the SQL statement.

The simple example in Listing 2-21 gives a row number to each row in the result set, ordering by *SalesOrderID*.

LISTING 2-21 Row number ranking

```
SELECT
  SalesOrderID,
  CustomerID,
  ROW_NUMBER() OVER (ORDER BY SalesOrderID) AS RowNumber
 FROM Sales.SalesOrderHeader
```

The results are shown here:

```
SalesOrderID    CustomerID    RowNumber
--------------- ------------- ---------------
43659           676           1
43660           117           2
43661           442           3
43662           227           4
43663           510           5
43664           397           6
43665           146           7
43666           511           8
43667           646           9
 :
```

ORDER BY Options

The ranking functions order your result set by the fields specified in the *ORDER BY* statement contained in the *OVER* clause. Alternatively, you can include an additional *ORDER BY* statement in your result set; this optional statement is distinct from the *ORDER BY* clause in the *OVER* expression. SQL Server allows this, but if you choose this option, the *ROW_NUMBER* function's results are displayed in the order in which they are determined in the additional *ORDER BY* statement, *not* by the *ORDER BY* statement contained within the *OVER* clause. The results can, therefore, be confusing. To illustrate, if we provide an additional *ORDER BY CustomerID* clause to the very same query we just ran, we get these "jumbled" results:

```
SalesOrderID    CustomerID    RowNumber
--------------- ------------- --------------
43860           1             202
44501           1             843
45283           1             1625
46042           1             2384
46976           2             3318
47997           2             4339
49054           2             5396
50216           2             6558
51728           2             8070
57044           2             13386
63198           2             19540
69488           2             25830
65310           3             21652
71889           3             28231
53616           3             9958
 :
```

As you can see, if you expect the results to be sorted by the *OVER* clause's *ORDER BY* statement, you'd expect results ranked by *SalesOrderID*, when in fact they're ordered by *CustomerID*.

If you choose the *ROW_NUMBER* function to run against a nonunique column that contains multiple copies of the same value (also known as "ties," such as the same amount of items sold and the same time in a race), *ROW_NUMBER* breaks the tie and still produces a running count so that no rows have the same number. In Listing 2-22, for example, *CustomerID* can repeat, which will generate several ties; SQL Server simply increases the running count for each row, regardless of how many ties exist.

LISTING 2-22 Row number ranking with ties

```
SELECT
  SalesOrderID,
  CustomerID,
  ROW_NUMBER() OVER (ORDER BY CustomerID) AS RowNumber
 FROM Sales.SalesOrderHeader
```

The results are shown here:

```
SalesOrderID    CustomerID     RowNumber
--------------- -------------- -----------------
43860           1              1
44501           1              2
45283           1              3
46042           1              4
46976           2              5
47997           2              6
49054           2              7
50216           2              8
51728           2              9
57044           2              10
63198           2              11
69488           2              12
44124           3              13
 :
```

Grouping and Filtering with *ROW_NUMBER*

When you want to include a *GROUP BY* function in your query, ranking functions do not work. The easy way around this limitation is to create your *GROUP BY* in a CTE and then perform your ranking on the results, as shown in Listing 2-23.

LISTING 2-23 Grouping by row number

```
WITH CustomerSum
AS
(
  SELECT CustomerID, SUM(TotalDue) AS TotalAmt
   FROM Sales.SalesOrderHeader
   GROUP BY CustomerID
)
```

```
--this appends a row number to the end of the result set
SELECT
  *,
  ROW_NUMBER() OVER (ORDER BY TotalAmt DESC) AS RowNumber
 FROM CustomerSum
```

Here are the results:

```
CustomerID    TotalAmt        RowNumber
------------- --------------- ---------------
678           1179857.4657    1
697           1179475.8399    2
170           1134747.4413    3
328           1084439.0265    4
514           1074154.3035    5
155           1045197.0498    6
72            1005539.7181    7
 :
```

To filter by a *ROW_NUMBER*, you have to put the *ROW_NUMBER* function in a CTE, as shown in Listing 2-24.

LISTING 2-24 Filtering by row number

```
WITH NumberedRows AS
(
  SELECT
    SalesOrderID,
    CustomerID,
    ROW_NUMBER() OVER (ORDER BY SalesOrderID) AS RowNumber
   FROM Sales.SalesOrderHeader
)

SELECT * FROM NumberedRows
 WHERE RowNumber BETWEEN 100 AND 200
```

Here are the results:

```
SalesOrderID    CustomerID    RowNumber
--------------- ------------- --------------
43759           13257         100
43760           16352         101
43761           16493         102
 :
43857           533           199
43858           36            200
```

The *RANK* Function

The ranking function you will probably use the most is *RANK*, which ranks the data in the *ORDER BY* clause in the order you specify. *RANK* is syntactically exactly like *ROW_NUMBER* but with true ranking results. It works just like in the Olympics, when two people tie for the gold medal—the next rank is bronze. For example, with the *RANK* function, if four rows are tied with the value *1*, the next row value for the rank column will be *5*. Consider the code in Listing 2-25.

LISTING 2-25 The *RANK* function

```
SELECT
  SalesOrderID,
  CustomerID,
  RANK() OVER (ORDER BY CustomerID) AS Rank
 FROM Sales.SalesOrderHeader
```

Here are the results:

```
SalesOrderID    CustomerID    Rank
--------------- ------------- ----------------
43860           1             1
44501           1             1
45283           1             1
46042           1             1
46976           2             5
47997           2             5
49054           2             5
50216           2             5
51728           2             5
57044           2             5
63198           2             5
69488           2             5
44124           3             13
 :
```

Just as with the other ranking functions, *RANK* needs the aid of a CTE to work with aggregates. Consider this query that ranks the customers from highest to lowest by total sales. We have to use a CTE to perform the aggregate first and then rank over the newly created aggregate expression, as shown in Listing 2-26.

LISTING 2-26 Ranked aggregates

```
WITH CustomerSum AS
(
  SELECT CustomerID, SUM(TotalDue) AS TotalAmt
   FROM Sales.SalesOrderHeader
   GROUP BY CustomerID
)
SELECT
  *,
  RANK() OVER (ORDER BY TotalAmt DESC) AS Rank
 FROM CustomerSum
```

The results are shown here. Notice that customer 678 is in first place:

```
CustomerID  TotalAmt              Rank
----------- --------------------- --------------------
678         1179857.4657          1
697         1179475.8399          2
170         1134747.4413          3
328         1084439.0265          4
514         1074154.3035          5
 :
```

As stated earlier, it is important to remember that the ranking functions provided by SQL Server are valid only for the scope of the running query. If the underlying data changes and then you run the same query again, you will get different results. In Listing 2-27, for example, let's modify a record from customer 697. By changing one of the detail rows for an order placed by customer 697 and increasing the order quantity, we place customer 697 as our top customer.

LISTING 2-27 Changing *RANK* results with underlying data changes

```
UPDATE Sales.SalesOrderDetail
 SET OrderQty = 50  -- the original value was 2
 WHERE SalesOrderDetailID = 535
```

Now rerun the same query. Notice that customer 697 has now surpassed customer 678 as the top customer:

```
CustomerID  TotalAmt              Rank
----------- --------------------- --------------------
697         1272595.5474          1
678         1179857.4657          2
170         1134747.4413          3
328         1084439.0265          4
514         1074154.3035          5
 :
```

The *DENSE_RANK* and *NTILE* Functions

The last two ranking functions we will cover are *DENSE_RANK* and *NTILE*. *DENSE_RANK* works exactly like *RANK* except that it increments only on distinct rank changes—in other words, unlike in the Olympics, it awards a silver medal when there are two gold medals. Listing 2-28 shows an example.

LISTING 2-28 Ranking with *DENSE_RANK*

```
SELECT
  SalesOrderID,
  CustomerID,
  DENSE_RANK() OVER (ORDER BY CustomerID) AS DenseRank
 FROM Sales.SalesOrderHeader
 WHERE CustomerID > 100
```

The results are shown here:

```
SalesOrderID CustomerID  DenseRank
------------ ----------- --------------------
46950        101         1
47979        101         1
49048        101         1
50200        101         1
51700        101         1
57022        101         1
63138        101         1
69400        101         1
43855        102         2
44498        102         2
45280        102         2
46038        102         2
46951        102         2
47978        102         2
49103        102         2
50199        102         2
51733        103         3
57058        103         3
 :
```

The following example shows the difference between *RANK* and *DENSE_RANK*. We will round the customers' sales to the nearest hundred (because managers always like to look at whole numbers in their reports!) and look at the difference when we run into a tie. The code is shown in Listing 2-29.

LISTING 2-29 *RANK* versus *DENSE_RANK*

```
WITH CustomerSum AS
(
  SELECT
    CustomerID,
    ROUND(CONVERT(int, SUM(TotalDue)) / 100, 8) * 100 AS TotalAmt
   FROM Sales.SalesOrderHeader
   GROUP BY CustomerID
)
SELECT *,
  RANK() OVER (ORDER BY TotalAmt DESC) AS Rank,
  DENSE_RANK() OVER (ORDER BY TotalAmt DESC) AS DenseRank
 FROM CustomerSum
```

And here are the results:

```
CustomerID  TotalAmt    Rank    DenseRank
----------- ----------- ------- --------------------
697         1272500     1       1
678         1179800     2       2
170         1134700     3       3
328         1084400     4       4
 :
87          213300      170     170
667         210600      171     171
196         207700      172     172
451         206100      173     173
672         206100      173     173
27          205200      175     174
687         205200      175     174
163         204000      177     175
102         203900      178     176
 :
```

Notice that customers 451 and 672 are tied, with the same total sales amount. They are ranked 173 by both the *RANK* and the *DENSE_RANK* functions. What happens next is where the difference between the two functions comes into play. Customers 27 and 687 are tied for the next position, and they are both assigned 175 by *RANK* and 174 by *DENSE_RANK*. Customer 163 is the next nontie, and it is assigned 177 by *RANK* and 175 by *DENSE_RANK*.

NTILE divides the returned rows into approximately evenly sized groups, the number of which you specify as a parameter to the function. It assigns each member of a group the same number in the result set. A perfect example of this is the percentile ranking in a college examination or a road race. Listing 2-30 shows an example of using *NTILE*.

LISTING 2-30 Ranking with *NTILE*

```
SELECT
  SalesOrderID,
  CustomerID,
  NTILE(10000) OVER (ORDER BY CustomerID) AS NTile
 FROM Sales.SalesOrderHeader
```

The results are shown here:

```
SalesOrderID    CustomerID    NTile
--------------- ------------- ---------------
43860           1             1
44501           1             1
45283           1             1
46042           1             1
46976           2             2
47997           2             2
49054           2             2
50216           2             2
51728           2             3
57044           2             3
63198           2             3
69488           2             3
44124           3             4
 :
45024           29475         9998
45199           29476         9998
60449           29477         9998
60955           29478         9999
49617           29479         9999
62341           29480         9999
45427           29481         10000
49746           29482         10000
49665           29483         10000

(31465 row(s) affected)
```

Using All the Ranking Functions Together

So far, we have looked at the ranking functions in isolation. The ranking functions are just regular SQL Server expressions, so you can have as many of them as you want in a single *SELECT* statement. We'll look at one last example in Listing 2-31 that brings these all together into one SQL statement and shows the differences between the four ranking functions.

LISTING 2-31 Contrasting SQL Server ranking functions

```
SELECT
  SalesOrderID AS OrderID,
  CustomerID,
  ROW_NUMBER() OVER (ORDER BY CustomerID) AS RowNumber,
  RANK() OVER (ORDER BY CustomerID) AS Rank,
  DENSE_RANK() OVER (ORDER BY CustomerID) AS DenseRank,
  NTILE(10000) OVER (ORDER BY CustomerID) AS NTile
 FROM Sales.SalesOrderHeader
```

The results are shown here:

```
OrderID  CustomerID    RowNumber Rank    DenseRank NTile
-------- ------------- --------- ------- --------- --------
43860    1             1         1       1         1
44501    1             2         1       1         1
45283    1             3         1       1         1
46042    1             4         1       1         1
46976    2             5         5       2         2
47997    2             6         5       2         2
49054    2             7         5       2         2
50216    2             8         5       2         2
51728    2             9         5       2         3
57044    2             10        5       2         3
63198    2             11        5       2         3
69488    2             12        5       2         3
44124    3             13        13      3         4
44791    3             14        13      3         4
 :
```

Ranking over Groups Using *PARTITION BY*

The ranking functions can also be combined with *windowing functions*. A windowing function divides a result set into equal partitions based on the values of your *PARTITION BY* statement in conjunction with the *OVER* clause in your ranking function. This is like applying a *GROUP BY* to your ranking function—you get a separate ranking for each partition. The example in Listing 2-32 uses *ROW_NUMBER* with *PARTITION BY* to count the number of orders by order date by salesperson. We do this by using *PARTITION BY SalesPersonID ORDER BY OrderDate*. You can do this with any of the four ranking functions.

LISTING 2-32 Ranking over groups with *PARTITION BY*

```
SELECT
  SalesOrderID,
  SalesPersonID,
  OrderDate,
  ROW_NUMBER() OVER (PARTITION BY SalesPersonID ORDER BY OrderDate) AS OrderRank
 FROM Sales.SalesOrderHeader
 WHERE SalesPersonID IS NOT NULL
```

The results are shown here. You might find that the order of your rows varies slightly from the results shown.

```
SalesOrderID    SalesPersonID    OrderDate    OrderRank
--------------- ---------------- ------------ --------------
 :
43659           279              2001-07-01 00:00:00.000    1
43660           279              2001-07-01 00:00:00.000    2
43681           279              2001-07-01 00:00:00.000    3
43684           279              2001-07-01 00:00:00.000    4
43685           279              2001-07-01 00:00:00.000    5
43694           279              2001-07-01 00:00:00.000    6
43695           279              2001-07-01 00:00:00.000    7
43696           279              2001-07-01 00:00:00.000    8
43845           279              2001-08-01 00:00:00.000    9
43861           279              2001-08-01 00:00:00.000    10
 :
48079           287              2002-11-01 00:00:00.000    1
48064           287              2002-11-01 00:00:00.000    2
48057           287              2002-11-01 00:00:00.000    3
47998           287              2002-11-01 00:00:00.000    4
48001           287              2002-11-01 00:00:00.000    5
48014           287              2002-11-01 00:00:00.000    6
47982           287              2002-11-01 00:00:00.000    7
47992           287              2002-11-01 00:00:00.000    8
48390           287              2002-12-01 00:00:00.000    9
48308           287              2002-12-01 00:00:00.000    10
 :
```

Let's partition our ranking function by country, as shown in Listing 2-33. We'll create a CTE to aggregate the sales by customer and by country. Then we'll apply the ranking function over the *TotalAmt* field and the *CustomerID* field, partitioned by the *CountryName* field.

LISTING 2-33 Aggregates with *PARTITION BY*

```
WITH CTETerritory AS
(
  SELECT
    cr.Name AS CountryName,
    CustomerID,
    SUM(TotalDue) AS TotalAmt
   FROM
    Sales.SalesOrderHeader AS soh
    INNER JOIN Sales.SalesTerritory AS ter ON soh.TerritoryID = ter.TerritoryID
    INNER JOIN Person.CountryRegion AS cr ON cr.CountryRegionCode = ter.
CountryRegionCode
   GROUP BY
    cr.Name, CustomerID
)
SELECT
  *,
  RANK() OVER(PARTITION BY CountryName ORDER BY TotalAmt, CustomerID DESC) AS Rank
 FROM CTETerritory
```

The results look like this:

```
CountryName    CustomerID    TotalAmt    Rank
-------------- ------------- ----------- --------------
Australia      29083         4.409       1
Australia      29061         4.409       2
Australia      29290         5.514       3
Australia      29287         5.514       4
Australia      28924         5.514       5
 :
Canada         29267         5.514       1
Canada         29230         5.514       2
Canada         28248         5.514       3
Canada         27628         5.514       4
Canada         27414         5.514       5
 :
France         24538         4.409       1
France         24535         4.409       2
France         23623         4.409       3
France         23611         4.409       4
France         20961         4.409       5
 :
```

PARTITION BY supports other SQL Server aggregate functions, including *MIN* and *MAX* as well as your own scalar functions. You can apply your aggregate function in the same way that you apply the ranking functions, with a *PARTITION BY* statement. Let's apply this technique to the current sample by adding a column to our result set using the *AVG* function, as shown in Listing 2-34. We will get the same results but with an additional column showing the average by country.

LISTING 2-34 Using *AVG* with *PARTITION BY*

```
WITH CTETerritory AS
(
  SELECT
    cr.Name AS CountryName,
    CustomerID,
    SUM(TotalDue) AS TotalAmt
   FROM
    Sales.SalesOrderHeader AS soh
    INNER JOIN Sales.SalesTerritory AS ter ON soh.TerritoryID = ter.TerritoryID
    INNER JOIN Person.CountryRegion AS cr ON cr.CountryRegionCode = ter.
CountryRegionCode
   GROUP BY
    cr.Name, CustomerID
)
SELECT
  *,
  RANK() OVER (PARTITION BY CountryName ORDER BY TotalAmt, CustomerID DESC) AS Rank,
  AVG(TotalAmt) OVER(PARTITION BY CountryName) AS Average
 FROM CTETerritory
```

Here are the results:

```
CountryName    CustomerID    TotalAmt    Rank    Average
-------------- ------------- ----------- ------- ------------------
Australia      29083         4.409       1       3364.8318
Australia      29061         4.409       2       3364.8318
Australia      29290         5.514       3       3364.8318
 :
Canada         29267         5.514       1       12824.756
Canada         29230         5.514       2       12824.756
Canada         28248         5.514       3       12824.756
 :
```

Exception Handling in Transactions

SQL Server offers major improvements in error handling inside T-SQL transactions. As of SQL Server 2005, you can catch T-SQL and transaction abort errors using the *TRY/CATCH* model without any loss of the transaction context. The only types of errors that the *TRY/CATCH* construct can't handle are those that cause the termination of your session (usually errors with severity 21 and above, such as hardware errors). The syntax is shown here:

```
BEGIN TRY
  --sql statements
END TRY
BEGIN CATCH
  --sql statements for catching your errors
END CATCH
```

If an error within an explicit transaction occurs inside a *TRY* block, control is passed to the *CATCH* block that immediately follows. If no error occurs, the *CATCH* block is completely skipped.

You can investigate the type of error that was raised and react accordingly. To do so, you can use the *ERROR_xxx* functions to return error information in the *CATCH* block, as shown in Listing 2-35.

LISTING 2-35 T-SQL exception handling example

```
BEGIN TRY
  SELECT 5/0
END TRY
BEGIN CATCH
  SELECT
   ERROR_NUMBER()    AS ErrNumber,
   ERROR_SEVERITY()  AS ErrSeverity,
   ERROR_STATE()     AS ErrState,
   ERROR_PROCEDURE() AS ErrProc,
   ERROR_LINE()      AS ErrLine,
   ERROR_MESSAGE()   AS ErrMessage
END CATCH
```

You can examine the value reported by any of the various *ERROR_xxx* functions to decide what to do with the control flow of your procedure and whether to abort any transactions. In our example in Listing 2-35, which attempts to divide by zero, here are the values returned by the error functions. (The *ERROR_PROCEDURE* function returns *NULL* in this example because the exception did not occur within a stored procedure.)

```
ErrNumber  ErrSeverity  ErrState   ErrProc  ErrLine  ErrMessage
---------- ------------ ---------- -------- -------- ---------------------------------
8134       16           1          NULL     2        Divide by zero error encountered.
```

When you experience a transaction abort error inside a transaction located in the *TRY* block, control is passed to the *CATCH* block. The transaction then enters a failed state in which locks are not released and persisted work is not reversed until you explicitly issue a *ROLLBACK* statement. You're not allowed to initiate any activity that requires opening an implicit or explicit transaction until you issue a *ROLLBACK*.

Certain types of errors are not detected by the *TRY/CATCH* block, and you end up with an unhandled exception even though the error occurred inside your *TRY* block. If this happens, the *CATCH* block is not executed. This is because *CATCH* blocks are invoked by errors that take place in actual executing code, not by compile or syntax errors. Two examples of such errors are syntax errors and statement-level recompile errors (for example, selecting from a nonexistent table). These errors are not caught at the same execution level as the *TRY* block, but at the lower level of execution—when you execute dynamic SQL or when you call a stored procedure from the *TRY* block. For example, if you have a syntax error inside a *TRY* block, you get a compile error and your *CATCH* block will not run, as shown here:

```
-- Syntax error doesn't get caught
BEGIN TRY
   SELECT * * FROM Customer
END TRY
BEGIN CATCH
   PRINT 'Error'
END CATCH
```

The result is an error from SQL Server, not from your *CATCH* block, as follows:

```
Msg 102, Level 15, State 1, Line 2
Incorrect syntax near '*'.
```

Statement-level recompilation errors also don't get caught by *CATCH* blocks. For example, using a nonexistent object in a *SELECT* statement in the *TRY* block forces an error from SQL Server, but your *CATCH* block will not execute, as shown here:

```
-- Statement level recompilation doesn't get caught
BEGIN TRY
  SELECT * FROM NonExistentTable
END TRY
BEGIN CATCH
  PRINT 'Error'
END CATCH
```

The result is an error from SQL Server, as follows:

```
Msg 208, Level 16, State 1, Line 3
Invalid object name 'NonExistentTable'.
```

When you use dynamic SQL or a stored procedure, these types of compile errors do get caught because they are part of the current level of execution. Each of the SQL blocks shown in Listing 2-36 will execute the *CATCH* block.

LISTING 2-36 Catching syntax and recompilation errors in dynamic SQL and stored procedure calls with exception handlers

```
-- Dynamic SQL Example
BEGIN TRY
  EXEC sp_executesql 'SELECT * * FROM Customer'
END TRY
BEGIN CATCH
  PRINT 'Error'
END CATCH
GO

-- Stored Procedure Example
CREATE PROCEDURE MyErrorProc
AS
  SELECT * FROM NonExistentTable
GO

BEGIN TRY
  EXEC MyErrorProc
END TRY
BEGIN CATCH
  PRINT 'Error'
END CATCH
```

The varchar(max) Data Type

The *varchar(max)*, *nvarchar(max)*, and *varbinary(max)* data types are extensions of the *varchar*, *nvarchar*, and *varbinary* data types that can store up to 2 gigabytes (GB) of data. They are alternatives to *text*, *ntext*, and *image* and use the *max* size specifier. Using one of these data types is easy—you just specify it in your *CREATE TABLE* statement (or in any variable declaration) with a *(max)* identifier, as in this example:

```
CREATE TABLE TableWithMaxColumn
 (Customer_Id int, CustomerLifeStory varchar(max))
```

All the standard T-SQL string functions operate on *varchar(max),* including concatenation functions *SUBSTRING, LEN,* and *CONVERT*. For example, you can use the T-SQL *SUBSTRING*

function to read parts of the string (*chunks*), and the *UPDATE* statement has also been enhanced to support the updating of chunks.

This is a vast improvement over the limitations in SQL Server 2000 and earlier, where *text* and *image* fields were used to store this type of data. These data types are not allowed as stored procedure parameters and cannot be updated directly, and many of the string manipulation functions don't work on the *text* data type.

The *WAITFOR* Statement

In SQL Server 2000, *WAITFOR* waited for a specified duration or a supplied *datetime* value. Starting with SQL Server 2005, as with the *TOP* enhancements, you can use *WAITFOR* with a SQL expression. You can essentially use the *WAITFOR* function to wait for a T-SQL statement to affect at least one row. (You can also set a time-out on that SQL expression.) You can specify *WAITFOR* to wait not only in *SELECT* statements but also in *INSERT*, *UPDATE*, *DELETE*, and *RECEIVE* statements. In essence, *SELECT* statements won't complete until at least one row is produced, and data manipulation language (DML) statements won't complete until at least one row is affected.

Here is the syntax:

```
WAITFOR(<statement>) [,TIMEOUT <timeout_value>]
```

This feature provides an alternative to polling. For example, you can use *WAITFOR* to select all the records in a log or a queue table, as shown here:

```
WAITFOR (SELECT * FROM MyQueue)
```

DDL Triggers

SQL Server supports data definitin language (DDL) triggers, allowing you to trap DDL operations and react to them. You can thus roll back the DDL activity. DDL triggers work synchronously, immediately after the triggering event, similar to the way that DML triggers work. DDL triggers can be database-wide and can react to certain types of DDLs or all DDLs.

The cool thing about DDL triggers is that you can get context information from querying the *EVENTDATA* function. Event data is an XML payload of data about what was happening when your DDL trigger ran (including information about the time, connection, and user), the type of event that was fired, and other useful data. To get at *EVENTDATA* data, you have to use the *EVENTDATA* function in your trigger code. If you issue a *ROLLBACK* statement in the trigger, the *EVENTDATA* function will no longer return information. In this situation, you must store the information in a variable before issuing the *ROLLBACK* statement to be accessed later.

Note SQL Server 2008 introduces new security features that allow you to audit DDL actions, as an alternative to using DDL triggers for auditing the same information. We cover SQL Server Audit in Chapter 5.

The *AdventureWorks2008* trigger, shown in Listing 2-37, is created at the database level and will capture *DROP TABLE* attempts. First we'll create a log table to log all our event data using an XML column, and then we'll create a dummy table that we will attempt to delete for testing our trigger on. Our trigger will issue a *ROLLBACK* (which effectively cancels the attempted *DROP TABLE* operation) and then write the event data to this table.

LISTING 2-37 Catching *DROP TABLE* attempts with a trigger

```
-- Create a log table
CREATE TABLE TriggerLog (LogInfo xml)

-- Create a dummy table to delete later on
CREATE TABLE TableToDelete (Id int PRIMARY KEY)

-- Add some dummy data
INSERT INTO TableToDelete VALUES(1)
GO

-- Create a trigger that will prevent the table from being deleted
CREATE TRIGGER StopTableDrop ON DATABASE AFTER DROP_TABLE
AS
  DECLARE @EventData AS xml
  SET @EventData = EVENTDATA() -- must be captured *before* rollback
  ROLLBACK
  PRINT 'DROP TABLE attempt in database ' + DB_NAME() + '.'
  INSERT INTO TriggerLog VALUES(@EventData)
```

The following example attempts to drop a table (which will fail because of our DDL trigger) and then queries the *TriggerLog* table to examine the details of the attempt:

```
-- The trigger in action...
DROP TABLE TableToDelete
SELECT * FROM TriggerLog
```

The results look like this:

```
DROP TABLE attempt in database AdventureWorks2008.

(1 row(s) affected)
Msg 3609, Level 16, State 2, Line 2
The transaction ended in the trigger. The batch has been aborted.
```

The *EventData* XML recorded to the *TriggerLog* table at the time we ran the script in Listing 2-37 on our system looks like this:

```
<EVENT_INSTANCE>
  <EventType>DROP_TABLE</EventType>
  <PostTime>2008-06-11T22:07:42.910</PostTime>
  <SPID>55</SPID>
  <ServerName>SQL08DEV</ServerName>
  <LoginName>SQL08DEV\Administrator</LoginName>
  <UserName>dbo</UserName>
  <DatabaseName>AdventureWorks2008</DatabaseName>
  <SchemaName>dbo</SchemaName>
  <ObjectName>TableToDelete</ObjectName>
  <ObjectType>TABLE</ObjectType>
  <TSQLCommand>
    <SetOptions ANSI_NULLS="ON" ANSI_NULL_DEFAULT="ON" ANSI_PADDING="ON" QUOTED_
IDENTIFIER="ON" ENCRYPTED="FALSE" />
    <CommandText>DROP TABLE TableToDelete</CommandText>
  </TSQLCommand>
</EVENT_INSTANCE>
```

SNAPSHOT Isolation

For working with T-SQL transactions, SQL Server 2005 introduced a new isolation level called SNAPSHOT that allows you to work in a mode in which writers don't block readers. Readers thus read from a previously committed version of the data they request, rather than being blocked during write transactions. SNAPSHOT isolation works by SQL Server maintaining a linked list in *tempdb* that tracks changes to rows and constructs an older, committed version of data for readers to access. This isolation level is useful for optimistic locking, in which *UPDATE* conflicts are uncommon. For example, if process 1 retrieves data and later attempts to modify it, and if process 2 has modified the same data between the retrieval and modification belonging to process 1, SQL Server produces an error when process 1 attempts to modify data because of the conflict. Process 1 can then try to reissue the transaction. This mode can be efficient in situations where *UPDATE* conflicts are not common.

Look for a more thorough discussion of SQL Server isolation levels and transactions in Chapter 12.

Table-Valued Parameters

We explored common table expressions (CTEs) at the beginning of this chapter and saw how, in SQL Server 2005, they provided a new alternative to using temporary tables and table variables, which have been around for a long time in SQL Server. Table-valued parameters (TVPs) in SQL Server 2008 give us yet another choice for treating a set of rows as a single entity that you can query or join against. It is now also remarkably easy to send an entire set of rows

from our .NET client applications to SQL Server with just one server roundtrip using Microsoft ADO.NET, as you'll see close up toward the end of our discussion of TVPs.

Note TVPs and all the remaining topics in this chapter are all-new features available only in SQL Server 2008.

More than Just Another Temporary Table Solution

A TVP is based on a new user-defined table type in SQL Server 2008 that describes the schema for a set of rows that can be passed to stored procedures or user-defined functions (UDFs). It's helpful to understand TVPs by first comparing them with table variables, temp tables, and CTEs and then contrasting their similarities and differences. All of these techniques provide a different way of querying or joining against a typed temporary result set, enabling you to treat a TVP, table variable, temporary table, or CTE just as you would an ordinary table or view in many scenarios.

Both CTEs and table variables store their row data in memory, assuming reasonably sized sets that don't overflow the RAM cache allocated for them (in which case, they do get pushed into *tempdb*). In contrast, the new TVPs in SQL Server 2008 are always stored in *tempdb*. This means that they incur disk I/O just as regular temp tables do, which is a performance hit evaded by (reasonably sized) memory-resident CTEs and table variables, and a consideration for you to bear in mind when using TVPs. Conversely, this also means that TVPs are better suited than CTEs and table variables when dealing with larger numbers of rows, since TVPs can be indexed but CTEs and table variables cannot.

The real power of TVPs in SQL Server 2008 lies in the ability to pass an entire table (a set of rows) as a single parameter from client to server and between your T-SQL stored procedures and user-defined functions. Table variables and temporary tables, on the other hand, cannot be passed as parameters. CTEs are limited in scope to the statement following their creation (as mentioned at the beginning of this chapter) and are therefore inherently incapable of being passed as parameters.

Reusability is another side benefit of TVPs. The schema of a TVP is centrally maintained, which is not the case with table variables, temporary tables, and CTEs. You define the schema once by creating a new user-defined type (UDT) of type *table*, which you do by applying the new *AS TABLE* clause to the *CREATE TYPE* statement, as shown in Listing 2-38.

LISTING 2-38 Defining the schema for a user-defined table type

```
CREATE TYPE CustomerUdt AS TABLE
  (Id int,
   CustomerName nvarchar(50),
   PostalCode nvarchar(50))
```

This statement creates a new user-defined table type named *CustomerUdt* with three columns. TVP variables of type *CustomerUdt* can now be declared and populated with rows of data that fit this schema, which SQL Server will store in *tempdb* behind the scenes. These variables can be passed freely between stored procedures—unlike regular table variables, which are stored in RAM behind the scenes and cannot be passed as parameters. When TVP variables declared as *CustomerUdt* fall out of scope and are no longer referenced, the underlying data in *tempdb* supporting the TVP is deleted automatically by SQL Server.

Tip The schema of this *CustomerUdt* table type is now embedded in the database (just like any other UDT), which makes it easily reusable throughout your T-SQL code. Even if you don't necessarily need the parameter-passing capability of a TVP, defining one makes it easy to instantly declare a variable of the table type without declaring the entire schema. Compare that with table variables and temporary tables, where you can find yourself duplicating the same schema in your code.

So although it's called a table-valued *parameter*, in fact, we see that a TVP is essentially a new user-defined table *type*. This new table type earns its name by allowing populated instances of itself to be passed on as parameters to stored procedures and user-defined functions (UDFs)—something you still can't do with a regular table variable. In a sense, you can think of TVPs as table variables on steroids: they provide similar functionality (temporary row set storage), add new capabilities (can be passed as parameters), and carry more overhead (use disk storage versus RAM).

Note In this text, the terms *table-valued parameter (TVP)* and *user-defined table type* are used interchangeably.

Once the table type is defined, you can create stored procedures with parameters of that type to pass an entire set of rows using TVPs.

TVP types are displayed in Management Studio Object Explorer in the new *User-Defined Table Types* node beneath *Programmability, Types*, as shown in Figure 2-1.

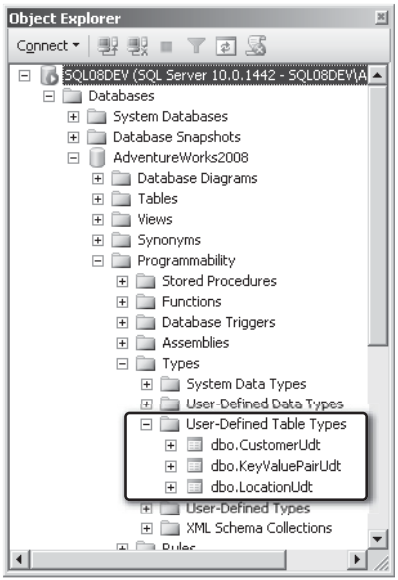


FIGURE 2-1 User-defined table types that can be used for TVPs displayed in Management Studio Object Explorer

There are many practical applications for this newly acquired ability to pass entire sets of data around as parameters, and we'll explore a number of them in the rest of this section. You will be sure to discover more good uses for TVPs on your own, just by setting your imagination loose on your requirements.

Working with a Multiple-Row Set

One common scenario in which TVPs can be applied is a typical order entry system. When a customer places an order, a new order row and any number of new order detail rows must be created in the database. Traditionally, this might be accomplished by creating two stored procedures—one for inserting an order row and one for inserting an order detail row. The application would invoke a stored procedure call for each individual row, so for an order with 20 details, there would be a total of 21 stored procedure calls (1 for the order and 20 for the details). There could of course be even larger orders with many more than 20 details. As a result, numerous roundtrips are made between the application and the database, each one carrying only a single row of data. Additional programming would also be required to wrap the entire set of insert operations within a single atomic transaction. The transaction would be needed to ensure that an order is completely inserted into the database—or not inserted at all—in the event that an error occurs at any point during the process.

Enter TVPs. Now you can create a single stored procedure with just two TVPs, one for the order row and one for the order details row. The client can now issue a single call to this stored procedure, passing to it both the order and all of the order details at one time, as shown in

Listing 2-39. Furthermore, the entire operation is guaranteed to succeed or fail as a whole, eliminating the need to program a transaction around the process.

LISTING 2-39 Creating a stored procedure that accepts TVPs

```
CREATE PROCEDURE uspInsertNewOrder
 (@OrderTvp AS OrderUdt READONLY,
  @OrderDetailsTvp AS OrderDetailUdt READONLY)
AS
     INSERT INTO [Order]
      SELECT * FROM @OrderTvp

     INSERT INTO [OrderDetail]
      SELECT * FROM @OrderDetailsTvp
```

As you can see, this code is inserted into the *Order* and *OrderDetail* tables directly from the rows passed in through the two TVPs. We have essentially performed a bulk insert with a single call, rather than individual inserts across multiple calls wrapped in a transaction. (Note that the code in Listing 2-39 assumes that the *Order* and *OrderDetail* tables already exist and that the *OrderUdt* and *OrderDetailUdt* table types have already been created with a column schema that matches the tables.)

We'll now take a closer look at the bulk insert possibilities for TVPs and how to create, declare, populate, and pass TVPs in T-SQL. Then we'll look at how TVPs can be used as a powerful T-SQL parameter-passing mechanism. In the last part of this section, we'll demonstrate how to populate TVPs and pass them across the network from .NET client application code to stored procedures using ADO.NET.

Using TVPs for Bulk Inserts and Updates

Here's an example of a stored procedure that you can create in the *AdventureWorks2008* database that accepts a TVP and inserts all of the rows that get passed in through it into the *Product.Location* table. By creating a user-defined table type named *LocationUdt* that describes the schema for each row passed to the stored procedure, any code can call the stored procedure and pass to it a set of rows for insertion into *Product.Location* using a single parameter typed as *LocationUdt*.

First we'll create the user-defined table data type *LocationUdt*, as shown in Listing 2-40.

LISTING 2-40 Creating the *LocationUdt* table type to be used for bulk operations with TVPs

```
CREATE TYPE LocationUdt AS TABLE(
  LocationName varchar(50),
  CostRate int)
```

Now a TVP variable of this type can be declared to hold a set of rows with the two columns *LocationName* and *CostRate*. These rows can be fed to a stored procedure by passing the TVP variable into it. The stored procedure can then select from the TVP just like a regular table or view and thus use it as the source for an *INSERT INTO…SELECT* statement that appends each row to the *Product.Location* table.

Rows added to *Product.Location* require more than just the two fields for the location name and cost rate. The table also needs values for the availability and modified date fields, so we'll let the stored procedure handle that. What we're doing here is defining a schema that can provide a *subset* of the required *Product.Location* fields (*Name* and *CostRate*), for passing multiple rows of data to a stored procedure that provides values for the remaining required fields (*Availability* and *ModifiedDate*). In our example, the stored procedure sets *Availability* to *0* and *ModifiedDate* to the *GETDATE* function on each row of data inserted from the TVP (passed in as the only parameter) that provides the values for *Name* and *CostRate*, as shown in Listing 2-41.

LISTING 2-41 Creating a stored procedure to perform a bulk insert using a TVP declared as the *LocationUdt* table type

```
CREATE PROCEDURE uspInsertProductionLocation
 (@TVP LocationUdt READONLY)
AS
     INSERT INTO [Production].[Location]
       ([Name], [CostRate], [Availability], [ModifiedDate])
      SELECT *, 0, GETDATE() FROM @TVP
```

We now have a stored procedure that can accept a TVP containing a set of rows with location names and cost rates to be inserted into the *Production.Location* table and that sets the availability quantity and modified date on each inserted row—all achieved with a single parameter and a single *INSERT INTO…SELECT* statement! The procedure doesn't know or care *how* the caller populates the TVP before it is used as the source for the *INSERT INTO…SELECT* statement. For example, the caller could manually add one row at a time, as follows:

```
DECLARE @LocationTvp AS LocationUdt

INSERT INTO @LocationTvp VALUES('UK', 122.4)
INSERT INTO @LocationTvp VALUES('Paris', 359.73)

EXEC uspInsertProductionLocation @LocationTvp
```

Or the caller could bulk insert into the TVP from another source table using *INSERT INTO…SELECT*, as in our next example. We will fill the TVP from the existing *Person.StateProvince* table using the table's *Name* column for *LocationName* and the value *0* for *CostRate*. Passing this TVP to our stored procedure will result in a new set of rows added to *Production.Location* with their *Name* fields set according to the names in the *Person.StateProvince* table, their

CostRate and *Availability* values set to *0*, and their *ModifiedDate* values set by *GETDATE*, as shown here:

```
DECLARE @LocationTVP AS LocationUdt

INSERT INTO @LocationTVP
 SELECT [Name], 0.00 FROM [Person].[StateProvince]

EXEC uspInsertProductionLocation @LocationTVP
```

The TVP could also be populated on the client using ADO.NET, which we cover at the end of this section.

Bulk updates (and deletes) using TVPs are possible as well. You can create an *UPDATE* statement by joining a TVP (which you must alias) to the table you want to update. The rows updated in the table are determined by the matches joined to by the TVP and can be set to new values that are also contained in the TVP. For example, you can pass a TVP populated with category IDs and names for updating the *Categories* table in the database, as shown in Listing 2-42. By joining the TVP to the *Categories* table on the category ID, all the matching rows in the *Categories* table can be updated with the new category names in the TVP.

LISTING 2-42 Bulk updates using TVPs

```
UPDATE Category
 SET Category.Name = ec.Name
 FROM Category INNER JOIN @EditedCategoriesTVP AS ec ON Category.Id = ec.Id
```

Working with a Single Row of Data

You don't need to be working with multiple rows of data in a set to derive benefit from TVPs. Because they encapsulate a schema, TVPs can come in handy even when dealing with only a single row that holds a collection of column values to be used as a typed parameter list.

It is now easy to pass these multiple values along from one stored procedure to another by using a single parameter rather than by authoring and maintaining duplicate signatures with multiple parameters in your T-SQL code. As you've seen, the schema of a TVP is defined just once, using the *CREATE TYPE…AS TABLE* statement shown earlier, and is thus maintained centrally in one location for the entire database. This makes TVPs easy to reuse as structures for wrapping a typed parameter list passed around as a single parameter in code.

By creating the preceding *CustomerUdt* TVP that encapsulates three columns, we can now pass the three values from one stored procedure to another using one TVP, rather than using three separate parameters for each column. For longer parameter lists, of course, this can mean the difference between one parameter versus a dozen of more. So without using the TVP, our stored procedure might look like the code in Listing 2-43.

LISTING 2-43 Creating a stored procedure with a typical parameter list

```
CREATE PROCEDURE uspPassWithoutUdt(
  @Id int,
  @CustomerName nvarchar(50),
  @PostalCode nvarchar(50))
AS
 BEGIN
    -- Use the parameters
    SELECT @Id, @CustomerName, @PostalCode
 END
```

A call to this stored procedure would typically look like this:

```
EXEC uspPassWithoutUdt 1, 'Christian Hess', '23911'
```

Now imagine many more stored procedures with the same parameter list. These would all have the same three parameters, which would lead you to copy and paste, duplicating and maintaining them in each stored procedure. Using a TVP instead alleviates this burden, since the schema is centrally defined only once in the definition of the user-defined table type (see Listing 2-38).

With this table type defined, our stored procedures now have a much simplified signature that no longer requires us to duplicate the parameter list. The same stored procedure can now be implemented as shown in Listing 2-44.

LISTING 2-44 Creating a stored procedure with parameter values embedded in a TVP

```
CREATE PROCEDURE uspPassWithUdt(@CustomerTvp CustomerUdt READONLY)
AS
 BEGIN
    -- Extract values from the TVP
    SELECT Id, CustomerName, PostalCode FROM @CustomerTvp
 END
```

To call this stored procedure, we declare a TVP variable of type *CustomerUdt*, insert into it a single row of data containing the three values, and then pass just the single variable on to the stored procedure, as shown here:

```
DECLARE @CustomerTvp CustomerUdt

INSERT INTO @CustomerTvp(Id, CustomerName, PostalCode)
 VALUES(1, 'Christian Hess', '23911')

EXEC uspPassWithUdt @CustomerTvp
```

This is logically analogous in the .NET world to creating a class or struct instance, populating its properties or fields, and passing the instance on to a method call that then retrieves the

values it wants from those properties or fields. A "package the parameters" approach such as this usually results in neater and more maintainable code—whether that code is written in .NET or T-SQL—particularly for lengthy and frequently reused parameter lists.

If you take proper care up front, you can design these parameter lists to tolerate the addition of new columns to the schema of the user-defined table type in the future without disturbing existing T-SQL code that references preexisting columns. To completely insulate code from extensions to the parameter list schema, you'll need to pay some special attention to your coding style when populating values—namely, avoiding *INSERT* statements that have no explicit column list. *INSERT* statements that populate a TVP should always use an explicit column list so that new columns added in the future do not affect their behavior and cause errors.

Tip In general, it is best practice to use explicit column lists in your *INSERT* statements.

INSERT statements with no explicit column list adapt dynamically with respect to the schema of the table being inserted into. As the result of adding new columns to the table, an *INSERT* statement without an explicit column list that worked before will now certainly fail because the number of values specified in its *VALUES* clause will no longer match the number of columns in the table. To ensure that these statements continue to function despite future extensions to the schema, they *must* provide an explicit column list for mapping the specified *VALUES*. This will allow you to insert new columns anywhere in the schema (not just appended to the end), since providing an explicit column list means that you are not tied to the number (or order) of the columns in the table type. You'll also need to make sure that new columns added to the schema are either nullable or have default values assigned to them so that existing *INSERT* statements will continue to work properly without the concern of providing values for newly added columns.

This strategy puts you in the best position, since you are now in control and can make your own decisions about how, when, and where to use the values newly added to your typed parameter lists. You can choose to use the newer values only in new T-SQL code that needs to support them, without ever revisiting or modifying existing code that continues to work just fine. You can also incrementally and selectively update any existing code as you need or want to.

If you've coded defensively as we've just discussed, you're ready to safely extend the *CustomerUdt* table definition by adding a fourth column. There is no *ALTER TYPE…AS TABLE* statement available, so the table type must be dropped and then re-created. Because SQL Server tracks and enforces integrity on object dependencies in the database, it will also be necessary to first drop all the stored procedures that reference the table type, drop the table type itself, re-create the table type with the new schema, and then re-create the dropped stored procedures. This might (and perhaps should) discourage you from too frequently

extending the parameter lists in your TVP schemas, although the burden is alleviated by the ability to easily generate all the necessary scripts that re-create all the programmability objects (stored procedures, TVPs, and so on) in the database. The code in Listing 2-45 drops and re-creates the table type and stored procedure.

LISTING 2-45 Revising the schema of the *CustomerUdt* table type

```
DROP PROCEDURE uspPassWithUdt
DROP TYPE CustomerUdt
GO

CREATE TYPE CustomerUdt AS TABLE
  (Id int,
   CustomerName nvarchar(50),
   HomePhone nvarchar(50), -- inserted new column
   PostalCode nvarchar(50))
GO

CREATE PROCEDURE uspPassWithUdt(@CustomerTvp CustomerUdt READONLY)
AS
BEGIN
    -- Extract values from the TVP
    SELECT Id, CustomerName, PostalCode FROM @CustomerTvp
 END
```

The *SELECT* statement in the stored procedure has not been modified and therefore continues to use only the original three columns. Had the *SELECT* statement been coded instead as *SELECT* * (that is, without an explicit column list), it would now be selecting the new *HomePhone* column as well, without any modification on your part. Understanding this, it's up to you to determine on a case-by-case basis whether to use *SELECT* * or *SELECT* with an explicit column list against your TVPs (although *SELECT* * is generally regarded as poor practice in production code). For *INSERT* statements, as we've explained, you should always use an explicit column list to code defensively against extensions to the TVP schema.

Creating Dictionary-Style TVPs

You can use TVPs to implement a generic dictionary-like structure to use as another flexible parameter-passing mechanism. This technique is roughly analogous to populating a *Dictionary* object in .NET that uses a string for the key and an object (which can hold any data type) for the value and then passing the dictionary object to another method. Achieving this with T-SQL in the past would typically require composing a string containing all of the data, with one delimiter separating each pair and another delimiter separating the key and value within each pair. The stored procedure would accept this string and then decompose it by parsing the delimiters to extract the values and casting them to their appropriate types. Using the intrinsic T-SQL string manipulation functions to implement this is both tedious and awkward. With SQL Server 2005, the situation could be improved marginally by leveraging

the *xml* data type and composing an "array" in XML to be passed on to a stored procedure, which could then decompose the XML using XQuery. Here too, TVPs can provide a much easier solution natively.

First create a simple user-defined table type named *KeyValuePairUdt*, as shown in Listing 2-46.

LISTING 2-46 Creating a table type to be used for storing and passing key-value pairs

```
CREATE TYPE KeyValuePairUdt AS TABLE
  (K nvarchar(50),
   V sql_variant)
```

This table type defines the two columns *K* and *V*, which can hold a string key and a value of any data type. (The *sql_variant* data type is a "one size fits all" type that can hold almost any SQL Server data type.) Then declare a TVP, populate it with several key-value pairs, and pass it on to your stored procedure, as shown here:

```
DECLARE @Username AS nvarchar(50)
DECLARE @DOB AS date
DECLARE @IsActive AS bit
SET @Username = 'Admin'
SET @DOB = '1/2/2006'
SET @IsActive = 1

DECLARE @ParamsPackage AS KeyValuePairUdt

INSERT INTO @ParamsPackage(K, V) VALUES('Username', @Username)
INSERT INTO @ParamsPackage(K, V) VALUES('DOB', @DOB)
INSERT INTO @ParamsPackage(K, V) VALUES('IsActive', @IsActive)

EXEC uspProcessEntry 392, @ParamsPackage
```

Here we have three values assigned into variables for username, date of birth, and an active flag, of types *nvarchar(50)*, *date*, and *bit*. We then declare a TVP named *@ParamsPackage* typed as our new *KeyValuePairUdt* table type, into which we insert each of the three differently typed values with the corresponding string keys *Username*, *DOB*, and *IsActive*. Because the key column (*K*) in *KeyValuePairUdt* is *nvarchar*, the keys must be strings. However, by declaring the value column (*V*) as *sql_variant*, we are able to store a mix of data types as values in each row of our *@ParamsPackage* TVP.

We then call the *uspProcessEntry* stored procedure, as shown in Listing 2-47, passing two parameters: a regular integer for *EntryId* and the TVP holding our dictionary of key-value pairs. The stored procedure picks up the first parameter as a normal integer value. To extract elements from the dictionary passed in by the second parameter, the stored procedure selects them one at a time from the TVP by the string keys it expects the caller to have set (*Username*, *DOB*, and *IsActive*). Again, because of the *sql_variant* data type, the stored procedure is able to extract the values in their various different native data types.

LISTING 2-47 Creating a stored procedure that accepts a dictionary of values using a TVP

```
CREATE PROCEDURE uspProcessEntry(
 @EntryId AS int,
 @KeyValuePairTvp AS KeyValuePairUdt READONLY)
AS
 BEGIN
   SET NOCOUNT ON
   DECLARE @Username AS nvarchar(max)
   DECLARE @DOB AS date
   DECLARE @IsActive AS bit

   SELECT @Username = CONVERT(nvarchar, V)
    FROM @KeyValuePairTvp WHERE K = 'Username'

   SELECT @DOB = CONVERT(date, V)
    FROM @KeyValuePairTvp WHERE K = 'DOB'

   SELECT @IsActive = CONVERT(bit, V)
    FROM @KeyValuePairTvp WHERE K = 'IsActive'

   PRINT 'EntryId: ' + CAST(@EntryId AS nvarchar)
   PRINT 'Username: ' + @Username
   PRINT 'DOB: ' + CAST(@DOB AS nvarchar)
   PRINT 'IsActive: ' + CAST(@IsActive as nvarchar)

 END
```

Passing TVPs Using ADO.NET

Arguably the most compelling facet of TVPs is the ability to marshal multiple rows of data from a client application to SQL Server 2008 without requiring multiple roundtrips or implementing special logic on the server for processing the data. We'll conclude our discussion of TVPs with the new *SqlDbType.Structured* enumeration in ADO.NET, which makes doing this both possible and easy.

Simply prepare a *SqlCommand* object as you did in the past, setting its *CommandType* property to *CommandType.StoredProcedure* and populating its *Parameters* collection with *SqlParameter* objects. All you do to mark a *SqlParameter* as a TVP is to set its *SqlDbType* property to *SqlDbType.Structured*. You will then be able to specify any *DataTable*, *DbDataReader*, or *IList<SqlDataRecord>* object as the parameter value to be passed to the stored procedure in a single call to the server.

In Listing 2-48, a new customer order is stored in separate *Order* and *OrderDetail DataTable* objects within a *DataSet*. The two tables are passed to the SQL Server stored procedure we saw earlier, which accepts them as TVPs for insertion into the *Order* and *OrderDetail* database tables.

LISTING 2-48 Passing TVPs to a SQL Server stored procedure from ADO.NET

```
// Assumes conn is an open SqlConnection object and ds is
// a DataSet with an Order and OrderDetails table
using(conn)
{
    // Create the command object to call the stored procedure
    SqlCommand cmd = new SqlCommand("uspInsertNewOrder", conn);
    cmd.CommandType = CommandType.StoredProcedure;

    // Create the parameter for passing the Order TVP
    SqlParameter headerParam = cmd.Parameters.AddWithValue
     ("@OrderTvp", ds.Tables["Order"]);

    // Create the parameter for passing the OrderDetails TVP
    SqlParameter detailsParam = cmd.Parameters.AddWithValue
     ("@OrderDetailsTvp", ds.Tables["OrderDetail"]);

    // Set the SqlDbType of the parameters to Structured
    headerParam.SqlDbType = SqlDbType.Structured;
    detailsParam.SqlDbType = SqlDbType.Structured;

    // Execute the stored procedure
    cmd.ExecuteNonQuery();
}
```

This code calls a SQL Server 2008 stored procedure and passes to it an order header and the complete set of order details with a single roundtrip in a single implicit transaction. Remarkably, it's just as simple as that.

You can also send a set of rows directly to a parameterized SQL statement without creating a stored procedure. Because the SQL statement is dynamically constructed on the client, there is no stored procedure signature that specifies the name of the user-defined table type for the TVP. Therefore, you need to tell ADO.NET what the type is by setting the *TypeName* property to the name of the table type defined on the server. For example, the code in Listing 2-49 passes a *DataTable* to a parameterized SQL statement.

LISTING 2-49 Passing TVPs to a parameterized SQL statement from ADO.NET

```
// Define the INSERT INTO...SELECT statement to insert into Categories
const string TSqlStatement =
 "INSERT INTO Categories (CategoryID, CategoryName)" +
 " SELECT nc.CategoryID, nc.CategoryName" +
 " FROM @NewCategoriesTvp AS nc";

// Assumes conn is an open SqlConnection object and ds is
// a DataSet with a Category table
using(conn)
{
    // Set up the command object for the statement
    SqlCommand cmd = new SqlCommand(TSqlStatement, conn);
```

```
    // Add a TVP specifying the DataTable as the parameter value
    SqlParameter catParam = cmd.Parameters.AddWithValue
     ("@NewCategoriesTvp", ds.Tables["Category"]);

    catParam.SqlDbType = SqlDbType.Structured;
    catParam.TypeName = "dbo.CategoriesUdt";

    // Execute the command
    cmd.ExecuteNonQuery();
}
```

Setting the *TypeName* property to *dbo.CategoriesUdt* in this code means that you have a user-defined table type by that name on the server, created using the *CREATE TYPE…AS TABLE* statement that defines the *CategoryID* and *CategoryName* columns.

You can also use any object derived from *DbDataReader* to stream rows of data to a TVP. In the example shown in Listing 2-50, we first call an Oracle stored procedure to select from an Oracle database into a connected *OracleDataReader*. The reader object gets passed as a single table-valued input parameter to a SQL Server 2008 stored procedure, which can then use the Oracle data in the reader as the source for adding new rows into the *Category* table in the SQL Server database.

LISTING 2-50 Passing a connected *OracleDataReader* source as a TVP to SQL Server

```
// Set up command object to select from Oracle
OracleCommand selCmd = new OracleCommand
 ("SELECT CategoryID, CategoryName FROM Categories;", oracleConn);

// Execute the command and return the results in a connected
// reader that will automatically close the connection when done
OracleDataReader rdr = selCmd.ExecuteReader
 (CommandBehavior.CloseConnection);

// Set up command object to insert into SQL Server
SqlCommand insCmd = new SqlCommand
 ("uspInsertCategories", connection);

insCmd.CommandType = CommandType.StoredProcedure;

// Add a TVP specifying the reader as the parameter value
SqlParameter catParam = cmd.Parameters.AddWithValue
 ("@NewCategoriesTvp", rdr);

catParam.SqlDbType = SqlDbType.Structured;

// Execute the stored procedure
insertCommand.ExecuteNonQuery();
```

TVP Limitations

There are a number of limitations to TVPs that you should be aware of. First and foremost, TVPs are read-only after they are initially populated and passed; they cannot be used to return data. The *READONLY* keyword must be applied to TVPs in the signatures of your stored procedures, or they will not compile. Similarly, the *OUTPUT* keyword cannot be used. You cannot update the column values in the rows of a TVP, and you cannot insert or delete rows. If you must modify the data in a TVP, one workaround is to insert the data from the TVP into a temporary table or into a table variable to which you can then apply changes.

There is no *ALTER TABLE…AS TYPE* statement that supports changing the schema of a TVP table type. Instead, you must first drop all stored procedures that reference the type before dropping the type, re-creating it with a new schema, and then re-creating the stored procedures. Indexing is limited as well, with support only for *PRIMARY KEY* and *UNIQUE* constraints. Also, statistics on TVPs are not maintained by SQL Server.

New Date and Time Data Types

The *date*, *time*, *datetime2*, and *datetimeoffset* types are four new data types in SQL Server 2008 for storing dates and times, which you should now begin using for new database development in lieu of the traditional *datetime* and *smalldatetime* data types. The new types are now much better aligned with the .NET Framework, Microsoft Windows, and the SQL standard—unlike *datetime* and *smalldatetime*—and have important advantages over those types, including improvements in range, precision, and storage.

In addition, SQL Server 2008 delivers full Open Database Connectivity (ODBC), OLE-DB, and ADO.NET client provider support for all four data types. They are compatible for use across all other SQL Server components, including client tools, Integration Services, Replication, Reporting Services, and Analysis Services.

Separation of Dates and Times

We'll begin by looking at the new *date* and *time* types. Database developers have long been clamoring for the ability to store dates and times as separate types, and SQL Server 2008 now finally delivers it to us with these two new types. If you need to store only a date value (for example, a date of birth), use the new *date* type. Similarly, use the new *time* type for storing just a time value (for example, a daily medication time), as shown here:

```
DECLARE @DOB date
DECLARE @MedsAt time
```

The *datetime* and *smalldatetime* types, which were the only previously available options, each include both a date and a time portion. In cases where only the date or only the time is

needed, the extraneous portion consumes storage needlessly, which results in wasted space in the database. In addition to saving storage, using *date* rather than *datetime* yields better performance for date-only manipulations and calculations, since there is no time portion to be handled or considered.

Note Separate *date* and *time* data types are planned for a future version of the .NET Framework. Until then, the .NET data types that map to the newly separated SQL Server 2008 *date* and *time* types are *System.DateTime* and *System.TimeSpan*.

More Portable Dates and Times

To continue storing both a date and a time as a single value, use the new *datetime2* data type. This new type supports the same range of values as the *DateTime* data type in the .NET Framework, so it can store dates from 1/1/0001 (*DateTime.MinValue* in .NET) to 12/31/9999 (*DateTime.MaxValue* in .NET) in the Gregorian calendar. Contrast this with the allowable date values for the regular *datetime* type, which range only from 1/1/1753 to 12/31/9999. This means that dates in .NET from 1/1/0001 through 12/31/1752 can't be stored at all in SQL Server's *datetime* type, a problem solved by using either the *date*, *datetime2*, or *datetimeoffset* type in SQL Server 2008. Since the supported range of dates is now the same in both .NET and SQL Server, any date can be safely passed between these client and server platforms with no special considerations. You are strongly encouraged to discontinue using the older *datetime* and *smalldatetime* data types and to use only *date* and *datetime2* types for new development (or the new *datetimeoffset* type for time zone awareness, which is discussed next).

Note The SQL standard calls the *datetime2* data type a *timestamp*. Unfortunately, Microsoft has already used the name *timestamp* for a special data type that generates unique binary values often used for row versioning but does not, in fact, represent either a date or a time. For this reason, SQL Server 2008 again deviates from the SQL standard by naming this new type *datetime2* and simultaneously introduces a new synonym for the poorly chosen *timestamp* type, now more aptly named *rowversion*. This discourages continued use of the name *timestamp* as originally defined by Microsoft.

You're not alone if you feel that *datetime2* was yet another poor naming choice for this new type designed as a replacement for *datetime*. Being unable to use the name *timestamp*, Microsoft clearly had a difficult time coming up with a new name and finally just settled on *datetime2*. Don't be surprised to see a new synonym for this type appear in the future, when someone in Redmond finally has better luck at choosing good names!

There has also been a need for greater precision of fractional seconds in time values. The *datetime* type is accurate only within roughly 3.33 milliseconds, whereas time values in Windows and .NET have a significantly greater, 100-nanosecond (10-millionth of a second),

accuracy. (The *smalldatetime* type doesn't even support seconds and is accurate only to the minute.) Storing times in the database therefore results in a loss of precision.

Like the expanded range of supported dates, the new *time*, *datetime2*, and *datetimeoffset* types are now more aligned with .NET and other platforms by also providing the same 100-nanosecond accuracy. As a result, we no longer incur any data loss of fractional second accuracy between platforms when recording time values to the database. Of course, there are storage implications that come with greater time precision, and we'll discuss those momentarily.

Time Zone Awareness

The fourth and last new data type in this category introduced in SQL Server 2008 is *datetimeoffset*. This type defines a date and time with the same range and precision that *datetime2* provides but also includes an offset value with a range of –14:00 to +14:00 that identifies the time zone. In the past, the only practical approach for globalization of dates and times in the database has been to store them in Coordinated Universal Time (UTC) format. Doing this requires back-and-forth conversion between UTC and local time that must be handled at the application level, and that means writing code.

Using the new *datetimeoffset* type, you can now store values that represent the local date and time in different regions of the world and include the appropriate time zone offset for the region in each value. Because the time zone offset embedded in the date and time value is specific to a particular locale, SQL Server is able to perform date and time comparisons between different locales without any conversion efforts required on your part. While *datetimeoffset* values appear to go in and come out as dates and times local to a particular region, they are internally converted, stored, and treated in UTC format for comparisons, sorting, and indexing.

Calculations and comparisons are therefore performed correctly and consistently across all dates and times in the database regardless of the different time zones in various regions. By simply appending the time zone offset to *datetimeoffset* values, SQL Server handles the conversions to and from UTC for you automatically in the background. Even better, you can obtain a *datetimeoffset* value either as UTC or local time. For those of us building databases that need to store various local times (or even just dates) in different regions of the world, this is welcomed as an extremely convenient new feature in SQL Server 2008. The database now handles all the details, so the application developer doesn't have to. Time zone functionality is now simply available for free right at the database level.

For example, the database knows that 9:15 AM in New York is in fact later than 10:30 AM in Los Angeles if you store the values in a *datetimeoffset* data type with appropriate time zone offsets. Because the New York time specifies a time zone offset of –5:00 and the Los Angeles time has an offset of –8:00, SQL Server is aware of the three-hour difference between the

two time zones and accounts for that difference in all date/time manipulations and calculations. This behavior is demonstrated by the code in Listing 2-51.

LISTING 2-51 Time zone calculations using *datetimeoffset*

```
DECLARE @Time1 datetimeoffset
DECLARE @Time2 datetimeoffset
DECLARE @MinutesDiff int

SET @Time1 = '2007-11-10 09:15:00-05:00'  -- NY time is UTC -05:00
SET @Time2 = '2007-11-10 10:30:00-08:00'  -- LA time is UTC -08:00

SET @MinutesDiff = DATEDIFF(minute, @Time1, @Time2)

SELECT @MinutesDiff
```

The output result from this code is:

```
-----------
255

(1 row(s) affected)
```

SQL Server was clearly able to account for the three-hour difference in time zones. Because 10:30 AM in Los Angeles is actually 1:30 PM in New York, a difference of 255 minutes (4 hours and 15 minutes) between that time and 9:15 AM New York time was calculated correctly.

Note Time zone *names* are not supported, nor is there support for daylight savings time. Unfortunately, these features did not make it into the final release of the product, but they are on the list for the next version of SQL Server. Time zones can be expressed only by hour/minute offsets, and you must continue to handle daylight savings time considerations on your own.

The .NET Framework also now provides the same functionality in a new type by the same name, *System.DateTimeOffset*. This means that .NET client applications and SQL Server can seamlessly pass time zone–aware values back and forth to each other.

Note SQL Server 2008 also provides time zone support for the *xsd:dateTime* type. We cover this Extensible Schema Definition (XSD) enhancement in Chapter 6.

Date and Time Accuracy, Storage, and Format

Date values stored in *date*, *datetime2*, and *datetimeoffset* types are compacted into a fixed storage space of 3 bytes. They use an optimized format that is 1 byte less than the 4 bytes

consumed by the date portion of the older *datetime* type (supporting a greater range in a smaller space).

Time values stored in *time*, *datetime2*, and *datetimeoffset* types, by default, consume five bytes of storage to support the same 100-nanosecond accuracy as Windows and .NET. However, the folks at Microsoft did not cast aside the storage concerns of developers and database administrators who don't require the highest degree of fractional-second precision. You can specify a lower degree of precision in order to benefit from further compacted storage by providing an optional scale parameter when declaring *time*, *datetime2*, and *datetimeoffset* variables. The scale can range from 0 to 7, with 0 offering no fractional-second precision at all being contained in the smallest space (3 bytes) and 7 (the default) offering the greatest fractional-second precision (100 nanoseconds) in the largest space (5 bytes). The scale essentially dictates the number of digits supported after the decimal point of the seconds value, where a scale value of 7 supports a fractional precision of 100 nanoseconds (each 100 nanoseconds being 0.0000001 second).

Table 2-1 shows the storage requirement and precision of *time*, *datetime2*, and *datetimeoffset* values for the different scale values from 0 through 7. The value in the precision column refers to the number of characters contained in an International Organization for Standardization (ISO)–formatted string representation of the three types for each possible scale value.

TABLE 2-1 Storage Requirements and Precision of the New Date and Time Data Types

	time		*datetime2*		*datetimeoffset*	
Scale Value	**Bytes**	**Precision**	**Bytes**	**Precision**	**Bytes**	**Precision**
0	3	8	6	19	8	26
1	3	10	6	21	8	28
2	3	11	6	22	8	29
3	4	12	7	23	9	30
4	4	13	7	24	9	31
5	5	14	8	25	10	32
6	5	15	8	26	10	33
7	5	16	8	27	10	34

The default scale is 7, which offers the greatest precision (to 100 nanoseconds) in the largest space. This means that declaring a variable as *time*, *datetime2*, or *datetimeoffset* is the same as declaring it as *time(7)*, *datetime2(7)*, or *datetimeoffset(7)*, making the following two statements equivalent:

```
DECLARE @StartDateTime datetime2
DECLARE @StartDateTime datetime2(7)
```

If you don't require any fractional precision at all, use a scale of 0, as in this example:

```
DECLARE @FeedingTime time(0)
```

As shown in the preceding table, only 3 bytes are required to store a time in *@FeedingTime*, which is accurate only to the second.

Two time values with differing scales are perfectly compatible with each other for comparison. SQL Server automatically converts the value with the lower scale to match the value with the greater scale and compares the two safely.

Almost all industry-standard string literal formats are supported for conveniently representing dates and times. For example, the date May 15, 2008, can be expressed in any of the formats shown in Table 2-2.

TABLE 2-2 Common Valid Date and Time String Literal Formats

Format	Example
Numeric	5/15/2008, 15-05-2008, 05.15.2008
Alphabetical	May 15, 2008
ISO8601	2008-05-15, 200805153
ODBC	{d'2008-05-15'}
W3C XML	2008-05-15Z

You have similar flexibility for representing times. For example, the same time value can be expressed as 23:30, 23:30:00, 23:30:00.0000, or 11:30:00 PM. Time zone offsets are expressed merely by appending a plus or minus sign followed by the UTC hours and minutes for the zone—for example, +02:00 for Jerusalem.

Note There are even more possible formatting variations than those we've indicated here. The purpose of Table 2-2 is to convey how accommodating SQL Server is with respect to variations in date and time syntax. Refer to SQL Server Books Online for a complete specification of supported date and time literal formats.

You can use *CAST* or *CONVERT* to extract just the date or time portion of a *datetime2* column for searching. When you perform such a conversion on a *datetime2* column that is indexed, SQL Server does not need to resort to a sequential table scan and is able to perform the much faster index seek to locate the specific date or time. For example, the following code defines a table with a *datetime2* type that has a clustered index. Selecting by date or time only can be achieved using *CONVERT*, while still using the clustered index for efficient searching, as shown in Listing 2-52.

LISTING 2-52 Using *CONVERT* to extract the date and time portion from a *datetime2* column

```
CREATE TABLE DateList(MyDate datetime2)
CREATE CLUSTERED INDEX idx1 ON DateList (MyDate);

-- Insert some rows into DateList...

SELECT MyDate FROM DateList WHERE CONVERT(date, MyDate) = '2005-04-07';
SELECT MyDate FROM DateList WHERE CONVERT(time(0), MyDate) = '09:00:00';
```

New and Changed Functions

All of the traditional date-related and time-related functions, including *DATEADD*, *DATEDIFF*, *DATEPART*, and *DATENAME* now of course fully support the new date and time data types introduced in SQL Server 2008, and several new functions have been added as well. We conclude our discussion of the new date and time data types by exploring the T-SQL extensions added to support them.

The new *SYSDATETIME* and *SYSUTCDATETIME* functions return the date and time on the server as *datetime2* types (with full seven-scale precision accuracy within 100 nanoseconds), just as the *GETDATE* and *GETUTCDATE* functions continue to return the current date and time as *datetime* types. Another new function, *SYSDATETIMEOFFSET*, returns the date and time on the server as a *datetimeoffset* type, with a time zone offset reflecting the regional settings established on the server, which includes awareness of local daylight savings time. The code in Listing 2-53 shows the contrast between the various similar server date and time functions.

LISTING 2-53 Comparing server date and time functions

```
SET NOCOUNT ON
SELECT GETDATE() AS 'GETDATE() datetime'
SELECT GETUTCDATE() AS 'GETUTCDATE() datetime'
SELECT SYSDATETIME() AS 'SYSDATETIME() datetime2'
SELECT SYSUTCDATETIME() AS 'SYSUTCDATETIME() datetime2'
SELECT SYSDATETIMEOFFSET() AS 'SYSDATETIMEOFFSET() datetimeoffset'
```

Running this code just after 8:20 PM on November 10, 2007, in New York results in the following output:

```
GETDATE() datetime
-----------------------
2007-11-10 20:21:19.380

GETUTCDATE() datetime
-----------------------
2007-11-11 01:21:19.380
```

```
SYSDATETIME() datetime2
---------------------------
2007-11-10 20:21:19.3807984

SYSUTCDATETIME() datetime2
---------------------------
2007-11-11 01:21:19.3807984

SYSDATETIMEOFFSET() datetimeoffset
----------------------------------
2007-11-10 20:21:19.3807984 -05:00
```

There are also new *TODATETIMEOFFSET* and *SWITCHOFFSET* functions that allow you to perform time zone offset manipulations. *TODATETIMEOFFSET* will convert any date or time type (that has no time zone offset) to a *datetimeoffset* type by applying whatever time zone offset you provide. *SWITCHOFFSET* makes it easy to find out what the same time is in two different time zones. You provide the *datetimeoffset* for a source location and a time zone offset for a target location, and *SWITCHOFFSET* returns a *datetimeoffset* representing the equivalent date and time in the target location, as shown in Listing 2-54.

LISTING 2-54 Performing time zone offset manipulations using *TODATETIMEOFFSET* and *SWITCHOFFSET*

```
DECLARE @TheTime datetime2
DECLARE @TheTimeInNY datetimeoffset
DECLARE @TheTimeInLA datetimeoffset

-- Hold a time that doesn't specify a time zone
SET @TheTime = '2007-11-10 7:35PM'

-- Convert it to one that specifies time zone for New York
SET @TheTimeInNY = TODATETIMEOFFSET(@TheTime, '-05:00')

-- Calculate the equivalent time in Los Angeles
SET @TheTimeInLA = SWITCHOFFSET(@TheTimeInNY , '-08:00')

SELECT @TheTime AS 'Any Time'
SELECT @TheTimeInNY AS 'NY Time'
SELECT @TheTimeInLA AS 'LA Time'
```

Here is the output result:

```
Any Time
---------------------------
2007-11-10 19:35:00.0000000

NY Time
----------------------------------
2007-11-10 19:35:00.0000000 -05:00

LA Time
----------------------------------
2007-11-10 16:35:00.0000000 -08:00
```

You can use *TODATETIMEOFFSET* with *INSERT INTO…SELECT* to bulk-insert date and time values with no time zone information from a source table into a target table and to apply a time zone offset to produce *datetimeoffset* values in the target table. For example, the following code copies all the row values from the *dt2* column in table *test1* (of type *datetime2*, which has no time zone information) into the *dto* column in *test2* (of type *datetimeoffset*) and applies a time zone offset of –05:00 to each copied value:

```
INSERT INTO test2(dto)
 SELECT TODATETIMEOFFSET(dt2, '-05:00') FROM test1
```

The next example retrieves all the *datetimeoffset* values from the *dto* column in the *test2* table, which can include values across a variety of different time zones. Using a *SWITCHOFFSET* function that specifies an offset of –05:00, the values are automatically converted to New York time from whatever time zone is stored in the *test2* table:

```
SELECT SWITCHOFFSET(dto, '-05:00') FROM test2
```

Last, both the existing *DATEPART* and *DATENAME* functions have been extended to add support for microseconds (*mcs*), nanoseconds (*ns*), and time zone offsets (*tz*) in the new types, as shown in Listing 2-55.

LISTING 2-55 Using the new date portions in SQL Server 2008 with *DATEPART* and *DATENAME*

```
SET NOCOUNT ON
DECLARE @TimeInNY datetimeoffset
SET @TimeInNY = SYSDATETIMEOFFSET()

-- Show the current time in NY
SELECT @TimeInNY AS 'Time in NY'

-- DATEPART with tz gets the time zone value
SELECT DATEPART(tz, @TimeInNY) AS 'NY Time Zone Value'

-- DATENAME with tz gets the time zone string
SELECT DATENAME(tz, @TimeInNY) AS 'NY Time Zone String'

-- Both DATEPART and DATENAME with mcs gets the microseconds
SELECT DATEPART(mcs, @TimeInNY) AS 'NY Time Microseconds'

-- Both DATEPART and DATENAME with ns gets the nanoseconds
SELECT DATEPART(ns, @TimeInNY) AS 'NY Time Nanoseconds'
```

Running this code returns the following output:

```
Time in NY
----------------------------------
2007-11-10 20:50:55.7851424 -05:00
```

```
NY Time Zone Value
------------------
-300

NY Time Zone String
-------------------------------
-05:00

NY Time Microseconds
--------------------
785142

NY Time Nanoseconds
-------------------
785142400
```

The *MERGE* Statement

The new *MERGE* statement in SQL Server 2008 does just what its name says. It combines the normal insert, update, and delete operations involved in a typical merge scenario, along with the select operation that provides the source and target data for the merge. That's right—it combines four statements into one. In fact, you can combine *five* statements into one using the *OUTPUT* clause, and even more than that with INSERT OVER DML (a new T-SQL feature), as you'll see later in this chapter.

In earlier versions of SQL Server, separate multiple statements were required to achieve what can now be accomplished with a single *MERGE* statement. This new statement has a flexible syntax that allows us to exercise fine control over source and target matching, as well as the various set-based DML actions carried out on the target. The result is simpler code that's easier to write and maintain (and also runs faster) than the equivalent code using separate statements to achieve the same result.

More Info We cover the most pertinent *MERGE* statement keywords in our discussions. You can and should refer to Books Online for the complete *MERGE* syntax.

More Info The *MERGE* statement is particularly suited to data warehousing scenarios. We cover the concepts and techniques behind data warehousing in Chapter 14.

Let's look at our first example, which uses *MERGE* to efficiently manage our stocks and trades. We begin by creating the two tables to hold stocks that we own and daily trades that we make, as shown in Listing 2-56.

LISTING 2-56 Creating the *Stock* and *Trade* tables

```
CREATE TABLE Stock(Symbol varchar(10) PRIMARY KEY, Qty int CHECK (Qty > 0))
CREATE TABLE Trade(Symbol varchar(10) PRIMARY KEY, Delta int CHECK (Delta <> 0))
```

We start off with 10 shares of Adventure Works stock and 5 shares of Blue Yonder Airlines stock. These are stored in our *Stock* table:

```
INSERT INTO Stock VALUES ('ADVW', 10)
INSERT INTO Stock VALUES ('BYA', 5)
```

During the day, we conduct three trades. We buy 5 new shares for Adventure Works, sell 5 shares of Blue Yonder Airlines, and buy 3 shares for our new investment in Northwind Traders. These are stored in our *Trade* table, as follows:

```
INSERT INTO Trade VALUES('ADVW', 5)
INSERT INTO Trade VALUES('BYA', -5)
INSERT INTO Trade VALUES('NWT', 3)
```

Here are the contents of the two tables:

```
SELECT * FROM Stock
GO

Symbol     Qty
---------- -----------
ADVW       10
BYA        5

(2 row(s) affected)


SELECT * FROM Trade
GO

Symbol     Delta
---------- -----------
ADVW       5
BYA        -5
NWT        3

(3 row(s) affected)
```

At the closing of the day, we want to update the quantities in our *Stock* table to reflect the trades of the day we recorded in the *Trade* table. Our Adventure Works stock quantity has risen to 15, we no longer own any Blue Yonder Airlines (having sold the only 5 shares we owned), and we now own 3 new shares of Northwind Traders stock. That's going to involve joining the *Stock* and *Trade* tables to detect changes in stock quantities resulting from our trades, as well as insert, update, and delete operations to apply those changes back to the