Microsoft

Foreword by César Galindo-Legaria, PhD
Manager, Query Optimization Team, Microsoft SQL Server

Inside Microsoft® SQL Server® 2008: T-SQL Querying

Itzik Ben-Gan
Lubor Kollar, Dejan Sarka, Steve Kass
Kalen Delaney–Series Editor

SOLID QUALITY MENTORS

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

Library of Congress Control Number: 2009920791

Printed and bound in the United States of America.

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

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: Denise Bankaitis
Editorial Production: S4Carlisle Publishing Services
Technical Reviewers: Steve Kass and Umachandar Jayachandran; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Cover: Tom Draper Design

Body Part No. X15-45856

To my parents, Mila & Gabi

—Itzik Ben-Gan

Table of Contents

Foreword xiii
Acknowledgments xv
Introduction xix

1 Logical Query Processing 1
Logical Query Processing Phases 2
Logical Query Processing Phases in Brief 3
Sample Query Based on Customers/Orders Scenario 5
Logical Query Processing Phase Details 7
Step 1: The FROM Phase 7
Step 2: The WHERE Phase 11
Step 3: The GROUP BY Phase 12
Step 4: The HAVING Phase 13
Step 5: The SELECT Phase 14
Step 6: The Presentation ORDER BY Phase 16
Further Aspects of Logical Query Processing 20
Table Operators 20
OVER Clause 29
Set Operators 31
Conclusion 33

2 Set Theory and Predicate Logic 35
An Example of English-to-Mathematics Translation 35
Well-Definedness 37
Equality, Identity, and Sameness 39
Mathematical Conventions 39
Numbers 41
Context 41
Functions, Parameters, and Variables 43
Instructions and Algorithms 43

Set Theory 44
Notation for Sets 45
Well-Definedness of Sets 46
Domains of Discourse 46
Faithfulness 49
Russell's Paradox 52
Ordered Pairs, Tuples, and Cartesian Products 53
The Empty Set(s) 54
The Characteristic Function of a Set 55
Cardinality 56
Order 57
Set Operators 61
Set Partitions 63
Generalizations of Set Theory 64
Predicate Logic 65
Logic-Like Features of Programming Languages 65
Propositions and Predicates 66
The Law of Excluded Middle 68
And, Or, and Not 68
Logical Equivalence 70
Logical Implication 70
Quantification 72
Alternatives and Generalizations 73
Relations 75
The Reflexive, Symmetric, and Transitive Properties 75
A Practical Application 77
Conclusion 81

3 The Relational Model 83

Introduction to the Relational Model 83
Relations, Tuples and Types 84
The Relational Model: A Quick Summary 89
Relational Algebra and Relational Calculus 90
Basic Operators 90
Relational Algebra 91
Relational Calculus 102
T-SQL Support 103
Data Integrity 104
Declarative Constraints 105
Other Means of Enforcing Integrity 109

Normalization and Other Design Topics 111
Normal Forms Dealing with Functional Dependencies 112
Higher Normal Forms 119
Denormalization 122
Generalization and Specialization 124
Conclusion 126

4 Query Tuning 127

Sample Data for This Chapter 127
Tuning Methodology 131
Analyze Waits at the Instance Level 134
Correlate Waits with Queues 143
Determine Course of Action 145
Drill Down to the Database/File Level 145
Drill Down to the Process Level 148
Tune Indexes and Queries 169
Tools for Query Tuning 171
Cached Query Execution Plans 171
Clearing the Cache 171
Dynamic Management Objects 172
STATISTICS IO 172
Measuring the Run Time of Queries 173
Analyzing Execution Plans 174
Hints 185
Traces/Profiler 186
Database Engine Tuning Advisor 187
Data Collection and Management Data Warehouse 187
Using SMO to Clone Statistics 187
Index Tuning 187
Table and Index Structures 188
Index Access Methods 197
Analysis of Indexing Strategies 244
Fragmentation 256
Partitioning 258
Preparing Sample Data 259
Data Preparation 259
TABLESAMPLE 265
An Examination of Set-Based vs. Iterative/Procedural Approaches and a Tuning Exercise 268
Conclusion 276

5 Algorithms and Complexity 277
Do You Have a Quarter? 278
How Algorithms Scale 279
An Example of Quadratic Scaling 280
An Algorithm with Linear Complexity 280
Exponential and Superexponential Complexity 281
Sublinear Complexity 282
Constant Complexity 283
Technical Definitions of Complexity 283
Comparing Complexities 285
Classic Algorithms and Algorithmic Strategies 286
Algorithms for Sorting 287
String Searching 289
A Practical Application 290
Identifying Trends in Measurement Data 291
The Algorithmic Complexity of LISLP 291
Solving the Longest Increasing Subsequence Length Problem in T-SQL 292
Conclusion 295

6 Subqueries, Table Expressions, and Ranking Functions 297
Subqueries 298
Self-Contained Subqueries 298
Correlated Subqueries 302
Misbehaving Subqueries 314
Uncommon Predicates 316
Table Expressions 318
Derived Tables 318
Common Table Expressions 321
Analytical Ranking Functions 330
Row Number 332
Rank and Dense Rank 352
Tile Number 354
Auxiliary Table of Numbers 359
Missing and Existing Ranges (Also Known as Gaps and Islands) 363
Missing Ranges (Gaps) 366
Existing Ranges (Islands) 375
Conclusion 387

7 Joins and Set Operations 389
Joins 389
Old Style vs. New Style 389
Fundamental Join Types 390
Further Examples of Joins 402
Sliding Total of Previous Year 417
Join Algorithms 421
Separating Elements 429
Set Operations 435
UNION 436
EXCEPT 437
INTERSECT 439
Precedence of Set Operations 440
Using INTO with Set Operations 441
Circumventing Unsupported Logical Phases 441
Conclusion 443

8 Aggregating and Pivoting Data 445
OVER Clause 445
Tiebreakers 448
Running Aggregations 451
Cumulative Aggregations 453
Sliding Aggregations 457
Year-to-Date (YTD) 459
Pivoting 460
Pivoting Attributes 460
Relational Division 465
Aggregating Data 466
Unpivoting 470
Custom Aggregations 473
Custom Aggregations Using Pivoting 474
User Defined Aggregates (UDA) 476
Specialized Solutions 487
Histograms 499
Grouping Factor 503
Grouping Sets 506
Sample Data 507
The GROUPING SETS Subclause 508

The CUBE Subclause . . . 511
The ROLLUP Subclause . . . 512
Grouping Sets Algebra . . . 514
The GROUPING_ID Function . . . 518
Materialize Grouping Sets . . . 521
Sorting . . . 524
Conclusion . . . 525

9 TOP and APPLY . . . 527

SELECT TOP . . . 527
TOP and Determinism . . . 529
TOP and Input Expressions . . . 530
TOP and Modifications . . . 531
TOP on Steroids . . . 534
APPLY . . . 535
Solutions to Common Problems Using TOP and APPLY . . . 537
TOP *n* for Each Group . . . 537
Matching Current and Previous Occurrences . . . 543
Paging . . . 547
Random Rows . . . 552
Median . . . 554
Logical Transformations . . . 556
Conclusion . . . 559

10 Data Modification . . . 561

Inserting Data . . . 561
Enhanced VALUES Clause . . . 561
SELECT INTO . . . 563
BULK Rowset Provider . . . 565
Minimally Logged Operations . . . 567
INSERT EXEC . . . 590
Sequence Mechanisms . . . 595
GUIDs . . . 600
Deleting Data . . . 601
TRUNCATE vs. DELETE . . . 601
Removing Rows with Duplicate Data . . . 601
DELETE Using Joins . . . 603

Updating Data 606
UPDATE Using Joins 606
Updating Large Value Types 610
SELECT and UPDATE Statement Assignments 611
Merging Data 616
MERGE Fundamentals 617
Adding a Predicate 621
Multiple WHEN Clauses 623
WHEN NOT MATCHED BY SOURCE 624
MERGE Values 626
MERGE and Triggers 627
OUTPUT Clause 628
INSERT with OUTPUT 629
DELETE with OUTPUT 630
UPDATE with OUTPUT 632
MERGE with OUTPUT 634
Composable DML 636
Conclusion 638

11 Querying Partitioned Tables 639
Partitioning in SQL Server 639
Partitioned Views 639
Partitioned Tables 640
Conclusion 657

12 Graphs, Trees, Hierarchies, and Recursive Queries 659
Terminology 659
Graphs 659
Trees 660
Hierarchies 661
Scenarios 661
Employee Organizational Chart 661
Bill of Materials (BOM) 663
Road System 666
Iteration/Recursion 670
Subordinates 671
Ancestors 681

Subgraph/Subtree with Path Enumeration 685
Sorting 688
Cycles 691
Materialized Path 694
Maintaining Data 695
Querying 701
Materialized Path with the HIERARCHYID Data Type 706
Maintaining Data 708
Querying 715
Further Aspects of Working with HIERARCHYID 719
Nested Sets 730
Assigning Left and Right Values 731
Querying 737
Transitive Closure 740
Directed Acyclic Graph 740
Conclusion 755

Appendix A: Logic Puzzles 757

Index 779

Foreword

I had met Itzik Ben-Gan briefly a couple of times and knew of his reputation, so I was looking forward to his afternoon session on avoiding cursors in SQL programming at PASS. I was lucky to get there early, as the large room filled up quickly. Itzik took a couple of SQL programming problems and diced them up in the most skillful and entertaining way, showing the elegance and efficiency of set-oriented thinking. The audience loved it—and so did I, except I had a different angle. Having worked on the internals of SQL Server, I could see Itzik touch the product nerves in his demos, and I admired how he turned features into beautiful solutions. After the session, I asked one of the attendees what had been his main takeaway, curious about which of the many techniques would have stood out for him. He looked at me, mildly surprised, and just said, "The man is a genius!" That pretty much sums it up.

This question of cursors is more fundamental than it may appear at first. It points to a deep dichotomy of tremendous practical importance. Most of us were taught to program by chopping up a task into smaller steps that, when executed in sequence, perform a desired computation. But if you approach SQL programming this way, you will get only mediocre results. Your code will be much larger and harder to maintain. It will be less efficient, less flexible, and less tunable. Using SQL effectively is not about an incremental extension of your procedural programming skills or about a specific collection of tricks. Writing SQL well requires approaching problems with a different mind-set—one that is declarative and set oriented, not procedural. This is the dichotomy.

Inside Microsoft SQL Server 2008: T-SQL Querying puts together all the ingredients you need to understand this declarative and set-oriented way of thinking and become a proficient SQL programmer, thus making an important contribution to the SQL Server development community. Its chapters on formal foundations help you understand the basis for the language philosophy and get a sense for its potential. The language itself is covered thoroughly, from the basic operations to the most advanced features, all of them explained in the context of real problem solving. The many examples show you what good SQL looks like, and they cover common patterns you are likely to find when writing applications. A comprehensive chapter on query tuning explains in detail the factors that impact performance in the system, how to go about identifying issues, and how to address them effectively.

Itzik assembled a strong team of collaborators to write this book. Coming from different backgrounds, all of them share a deep expertise in SQL, a passion for database technology, extensive teaching experience, and a recognized track record of contributions to the SQL Server community. Steve Kass is known for his depth of understanding and clarity of thought. Dejan Sarka contributes an extensive knowledge of the relational model and a breadth of database technologies. As for Lubor Kollar, I've had the pleasure of working with him on the definition, design, and implementation of the Query Processing engine of SQL Server for over a decade, and I deeply respect his insight. They make an outstanding team of guides who can help you improve your skills.

SQL is a very powerful language, but I believe only a minority of developers really know how to get the most out of it. Using SQL well can mean code that is 10 times more efficient, more scalable, and more maintainable. *Inside Microsoft SQL Server 2008: T-SQL Querying* tells you how.

César Galindo-Legaria, PhD

Manager of the Query Optimization Team, Microsoft SQL Server

Acknowledgments

Several people contributed to the T-SQL querying and T-SQL programming books, and I'd like to acknowledge their contributions. Some were involved directly in writing or editing the books, while others were involved indirectly by providing advice, support, and inspiration.

To the coauthors of *Inside Microsoft SQL Server 2008: T-SQL Querying*—Lubor Kollar, Dejan Sarka, and Steve Kass—and to the coauthors of *Inside Microsoft SQL Server 2008: T-SQL Programming*—Dejan Sarka, Roger Wolter, Greg Low, Ed Katibah, and Isaac Kunen—it is a great honor to work with you. It is simply amazing to see the level of mastery that you have over your areas of expertise, and it is pure joy to read your texts. Thanks for agreeing to be part of this project.

To Lubor, besides directly contributing to the books, you provide support, advice, and friendship and are a great source of inspiration. I always look forward to spending time with you—hiking, drinking, and talking about SQL and other things.

To Dejko, your knowledge of the relational model is admirable. Whenever we spend time together, I learn new things and discover new depths. I like the fact that you don't take things for granted and don't follow blindly words of those who are considered experts in the field. You have a healthy mind of your own and see things that very few are capable of seeing. I'd like to thank you for agreeing to contribute texts to the books. I'd also like to thank you for your friendship; I always enjoy spending time with you. We need to do the beer list thing again some time. It's been almost 10 years!

To the technical editor of the books, Steve Kass, your unique mix of strengths in mathematics, SQL, and English are truly extraordinary. I know that editing both books and also writing your own chapters took their toll. Therefore, I'd like you to know how much I appreciate your work. I know you won't like my saying this, but it is quite interesting to see a genius at work. It kept reminding me of Domingo Montoya's work on the sword he prepared for the six-fingered man from William Goldman's *The Princess Bride*.

To Umachandar Jayachandran (UC), many thanks for helping out by editing some of the chapters. Your mastery of T-SQL is remarkable, and I'm so glad you could join the project in any capacity. I'd also like to thank Bob Beauchemin for reviewing the chapter on Spatial Data.

To Cesar Galindo-Legaria, I feel honored that you agreed to write the foreword for the T-SQL querying book. The way you and your team designed SQL Server's optimizer is simply a marvel. I'm constantly trying to figure out and interpret what the optimizer does, and whenever I manage to understand a piece of the puzzle, I find it astonishing what a piece of software is capable of. Your depth of knowledge, your pleasant ways, and your humility are an inspiration.

To the team at Microsoft Press: Ken Jones, the product planner: I appreciate the personal manner in which you handle things and always look forward to Guinness sessions with you. I think that you have an impossible job trying to make everyone happy and keep projects moving, but somehow you still manage to do it.

To Sally Stickney, the development editor, thanks for kicking the project off the ground. I know that the T-SQL querying book was your last project at Microsoft Press before you started your new chosen path in life and am hopeful that it left a good impression on you. I wish you luck and happiness in your new calling.

To Denise Bankaitis, the project editor, you of all people at Microsoft Press probably spent most time working on the books. Thanks for your elegant project management and for making sure things kept flowing. It was a pleasure to work with you.

I'd also like to thank DeAnn Montoya, the project manager for the vendor editorial team, S4Carlisle Publishing Services, and Becka McKay, the copy editor. I know you spent countless hours going over our texts, and I appreciate it a lot.

To Solid Quality Mentors, being part of this amazing company and group of people is by far the best thing that happened to me in my career. It's as if all I did in my professional life led me to this place where I can fulfill my calling, which is teaching people about SQL. To Fernando Guerrero, Brian Moran, and Douglas McDowell: the company grew and matured because of your efforts, and you have a lot to be proud of. Being part of this company, I feel a part of something meaningful and that I'm among family and friends—among people whom I both respect and trust.

I'd like to thank my friends and colleagues from the company: Ron Talmage, Andrew J. Kelly, Eladio Rincón, Dejan Sarka, Herbert Albert, Fritz Lechnitz, Gianluca Hotz, Erik Veerman, Jay Hackney, Daniel A. Seara, Davide Mauri, Andrea Benedetti, Miguel Egea, Adolfo Wiernik, Javier Loria, Rushabh Mehta, Greg Low, Peter Myers, Randy Dyess, and many others. I'd like to thank Jeanne Reeves for making many of my classes possible and all the back-office team for their support. I'd also like to thank Kathy Blomstrom for managing our writing projects and for your excellent edits.

I'd like to thank the members of the SQL Server development team who are working on T-SQL and its optimization: Michael Wang, Michael Rys, Eric Hanson, Umachandar Jayachandran (UC), Tobias Thernström, Jim Hogg, Isaac Kunen, Krzysztof Kozielczyk, Cesar Galindo-Legaria, Craig Freedman, Conor Cunningham, and many others. For better or worse, what you develop is what we have to work with, and so far the results are outstanding! Still, until we get a full implementation of the OVER clause, you know I won't stop bothering you. ;-)

I'd like to thank Dubi Lebel and Assaf Fraenkel from Microsoft Israel and also Ami Levin, who helps me run the Israeli SQL Server users group.

To the team at *SQL Server Magazine*: Megan Bearly, Sheila Molnar, Mary Waterloo, Michele Crockett, Mike Otey, Lavon Peters, and Anne Grubb: Being part of this magazine is a great privilege. Congratulations on the 10th anniversary of the magazine! I can't believe that 10 years passed so quickly, but that's what happens when you have fun.

To my fellow SQL Server MVPs: Erland Sommarskog, Alejandro Mesa, Aaron Bertrand, Tibor Karaszi, Steve Kass, Dejan Sarka, Roy Harvey, Tony Rogerson, Marcello Poletti (Marc), Paul Randall, Bob Beauchemin, Adam Machanic, Simon Sabin, Tom Moreau, Hugo Kornelis, David Portas, David Guzman, and many others: Your contribution to the SQL Server community is remarkable. Much of what I know today is thanks to our discussions and exchange of ideas.

To my fellow SQL Server MCTs: Tibor Karaszi, Chris Randall, Ted Malone, and others: We go a long way back, and I'm glad to see that you're all still around in the SQL teaching community. We all share the same passion for teaching. Of anyone, you best understand the kind of fulfillment that teaching can bestow.

To my students: Without you, my work would be meaningless. Teaching is what I like to do best, and the purpose of pretty much everything else that I do with SQL—including writing these books—is to support my teaching. Your questions make me do a lot of research, and therefore I owe much of my knowledge to you.

To my parents, Emilia and Gabriel Ben-Gan, and to my siblings, Ina Aviram and Michael Ben-Gan, thanks for your continuous support. The fact that most of us ended up being teachers is probably not by chance, but for me to fulfill my calling, I end up traveling a lot. I miss you all when I'm away, and I always look forward to our family reunions when I'm back.

To Lilach, you're the one who needs to put up with me all the time and listen to my SQL ideas that you probably couldn't care less about. It's brainwashing, you see—at some point you will start asking for more, and before you know it, you will even start reading my books. Not because I will force you but because you will want to, of course. That's the plan at least. Thanks for giving meaning to what I do and for supporting me through some rough times of writing.

Introduction

This book and its sequel—*Inside Microsoft SQL Server 2008: T-SQL Programming*—cover advanced T-SQL querying, query tuning, and programming in Microsoft SQL Server 2008. They are designed for experienced programmers and DBAs who need to write and optimize code in SQL Server 2008. For brevity, I'll refer to the books as *T-SQL Querying* and *T-SQL Programming*, or just as *these books*.

Those who read the SQL Server 2005 edition of the books will find plenty of new materials covering new subjects, new features, and enhancements in SQL Server 2008, plus revisions and new insights about the existing subjects.

These books focus on practical common problems, discussing several approaches to tackle each. You will be introduced to many polished techniques that will enhance your toolbox and coding vocabulary, allowing you to provide efficient solutions in a natural manner.

These books unveil the power of set-based querying and explain why it's usually superior to procedural programming with cursors and the like. At the same time, they teach you how to identify the few scenarios where cursor-based solutions are superior to set-based ones.

This book—*T-SQL Querying*—focuses on set-based querying and query tuning, and I recommend that you read it first. The second book—*T-SQL Programming*—focuses on procedural programming and assumes that you read the first book or have sufficient querying background.

T-SQL Querying starts with five chapters that lay the foundation of logical and physical query processing required to gain the most from the rest of the chapters in both books.

The first chapter covers logical query processing. It describes in detail the logical phases involved in processing queries, the unique aspects of SQL querying, and the special mind-set you need to adopt to program in a relational, set-oriented environment.

The second chapter covers set theory and predicate logic—the strong mathematical foundations upon which the relational model is built. Understanding these foundations will give you better insights into the model and the language. This chapter was written by Steve Kass, who was also the main technical editor of these books. Steve has a unique combination of strengths in mathematics, computer science, SQL, and English that make him the ideal author for this subject.

The third chapter covers the relational model. Understanding the relational model is essential for good database design and helps in writing good code. The chapter defines relations and tuples and operators of relational algebra. Then it shows the relational model from a different perspective called *relational calculus*. This is more of a business-oriented perspective, as the logical model is described in terms of predicates and propositions. Data integrity is crucial for transactional systems; therefore, the chapter spends time discussing all kinds of constraints. Finally, the chapter introduces normalization—the formal process of improving database design. This chapter was written by Dejan Sarka. Dejan is one of the people with the deepest understanding of the relational model that I know.

The fourth chapter covers query tuning. It introduces a query tuning methodology we developed in our company (Solid Quality Mentors) and have been applying in production systems. The chapter also covers working with indexes and analyzing execution plans. This chapter provides the important background knowledge required for the rest of the chapters in both books, which as a practice discuss working with indexes and analyzing execution plans. These are important aspects of querying and query tuning.

The fifth chapter covers complexity and algorithms and was also written by Steve Kass. This chapter particularly focuses on some of the algorithms used often by the SQL Server engine. It gives attention to considering worst-case behavior as well as average case complexity. By understanding the complexity of algorithms used by the engine, you can anticipate, for example, how the performance of certain queries will degrade when more data is added to the tables involved. Gaining a better understanding of how the engine processes your queries equips you with better tools to tune them.

The chapters that follow delve into advanced querying and query tuning, addressing both logical and physical aspects of your code. These chapters cover the following subjects: subqueries, table expressions, and ranking functions; joins and set operations; aggregating and pivoting data; TOP and APPLY; data modification; querying partitioned tables; and graphs, trees, hierarchies, and recursive queries.

The chapter covering querying partitioned tables was written by Lubor Kollar. Lubor led the development of partitioned tables and indexes when first introduced in the product, and many of the features that we have today are thanks to his efforts. These days Lubor works with customers who have, among other things, large implementations of partitioned tables and indexes as part of his role in the SQL Server Customer Advisory Team (SQL CAT).

Appendix A covers logic puzzles. Here you have a chance to practice logical puzzles to improve your logic skills. SQL querying essentially deals with logic. I find it important to practice pure logic to improve your query problem-solving capabilities. I also find these puzzles fun and challenging, and you can practice them with the entire family. These puzzles

are a compilation of the logic puzzles that I covered in my T-SQL column in *SQL Server Magazine*. I'd like to thank *SQL Server Magazine* for allowing me to share these puzzles with the book's readers.

The second book—*T-SQL Programming*—focuses on programmatic T-SQL constructs and expands its coverage to treatment of XML and XQuery and the CLR integration. The book's chapters cover the following subjects: views; user-defined functions; stored procedures; triggers; transactions and concurrency; exception handling; temporary tables and table variables; cursors; dynamic SQL; working with date and time; CLR user-defined types; temporal support in the relational model; XML and XQuery (including coverage of open schema); spatial data; change data capture, change tracking, and auditing; and Service Broker.

The chapters covering CLR user-defined types, temporal support in the relational model, and XML and XQuery were written by Dejan Sarka. As I mentioned, Dejan is extremely knowledgeable in the relational model and has very interesting insights into the model itself and the way the constructs that he covers in his chapters fit in the model when used sensibly.

The chapter about spatial data was written by Ed Katibah and Isaac Kunen. Ed and Isaac are with the SQL Server development team and led the efforts to implement spatial data support in SQL Server 2008. It is a great privilege to have this chapter written by the designers of the feature. Spatial data support is new to SQL Server 2008 and brings new data types, methods, and indices. This chapter is not intended as an exhaustive treatise on spatial data or as an encyclopedia of every spatial method that SQL Server now supports. Instead, this chapter will introduce core spatial concepts and provide the reader with key programming constructs necessary to successfully navigate this new feature to SQL Server.

The chapter about change data capture, change tracking, and auditing was written by Greg Low. Greg is a SQL Server MVP and the managing director of SolidQ Australia. Greg has many years of experience working with SQL Server—teaching, speaking, and writing about it—and is highly regarded in the SQL Server community. The technologies that are the focus of this chapter track access and changes to data and are new in SQL Server 2008. At first glance, these technologies can appear to be either overlapping or contradictory, and the best-use cases for each might be far from obvious. This chapter explores each technology, discusses the capabilities and limitations of each, and explains how each is intended to be used.

The last chapter, which covers Service Broker (SSB), was written by Roger Wolter. Roger is the program manager with the SQL Server development team and led the initial efforts to introduce SSB in SQL Server. Again, there's nothing like having the designer of a component explain it in his own words. The "sleeper" feature of SQL Server 2005 is now in production in

a wide variety of applications. This chapter covers the architecture of SSB and how to use SSB to build a variety of reliable asynchronous database applications. The SQL 2008 edition adds coverage of the new features added to SSB for the SQL Server 2008 release and includes lessons learned and best practices from SSB applications deployed since the SQL Server 2005 release. The major new features are Queue Priorities, External Activation, and a new SSB troubleshooting application that incorporates lessons the SSB team learned from customers who have already deployed applications.

Hardware and Software Requirements

To practice all the material in these books and run all code samples, it is recommended that you use Microsoft SQL Server 2008 Developer or Enterprise Edition and Microsoft Visual Studio 2008 Professional or Database Edition. If you have a subscription to MSDN, you can download SQL Server 2008 and Visual Studio 2008 from *http://msdn.microsoft.com*. Otherwise, you can download a 180-day free SQL Server 2008 trial software from *http://www.microsoft.com/sqlserver/2008/en/us/trial-software.aspx* and a 90-day free Visual Studio 2008 trial software from *http://msdn.microsoft.com/en-us/vstudio/aa700831.aspx*.

You can find system requirements for SQL Server 2008 at *http://msdn.microsoft.com/en-us/library/ms143506.aspx* and for Visual Studio 2008 at *http://msdn.microsoft.com/en-us/vs2008/products/bb894726.aspx*.

Companion Content and Sample Database

These books feature a companion Web site that makes available to you all the code used in the books, the errata, additional resources, and more. The companion Web site is *http://www.insidetsql.com*.

For each of these books the companion Web site provides a compressed file with the book's source code, a script file to create the books' sample database, and additional files that are required to run some of the code samples.

After downloading the source code, run the script file TSQLFundamentals2008.sql to create the sample database InsideTSQL2008, which is used in many of the books' code samples. The data model of the InsideTSQL2008 database is provided in Figure I-1 for your convenience.

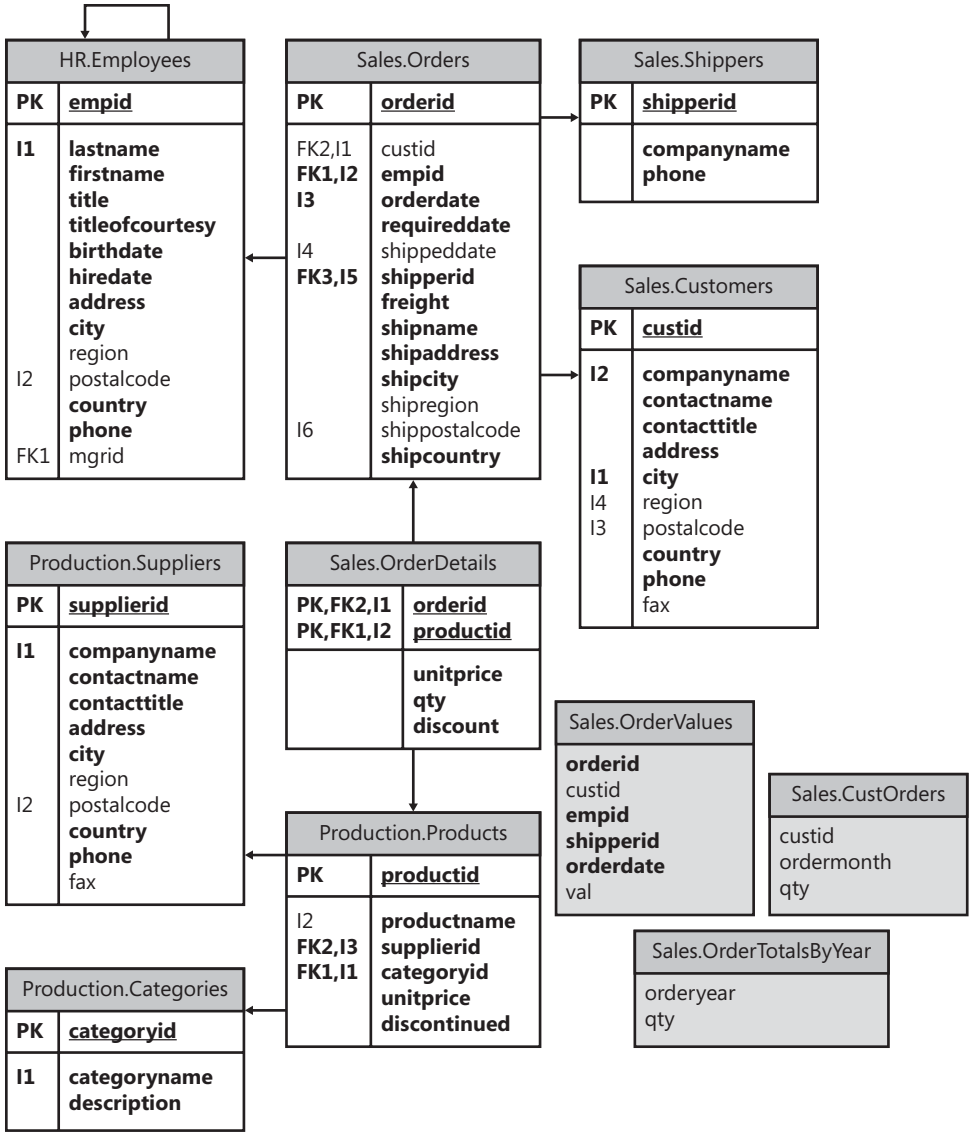


FIGURE I-1 Data model of the TSQLFundamentals2008 database

Find Additional Content Online

As new or updated material becomes available that complements your books, it will be posted online on the Microsoft Press Online Windows Server and Client Web site. The type of material you might find includes updates to books content, articles, links to companion content, errata, sample chapters, and more. This Web site is available at *http://microsoftpresssrv.libredigital.com/serverclient/* and is updated periodically.

Support for These Books

Every effort has been made to ensure the accuracy of these books and the contents of the companion Web site. As corrections or changes are collected, they will be added to a Microsoft Knowledge Base article.

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

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

Questions and Comments

If you have comments, questions, or ideas regarding the books or questions that are not answered by visiting the sites above, please send them to me via e-mail to

itzik@SolidQ.com

or via postal mail to

Microsoft Press

Attn: Inside Microsoft SQL Server 2008: T-SQL Querying and Inside Microsoft SQL Server 2008: T-SQL Programming Editor

One Microsoft Way

Redmond, WA 98052-6399.

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

Chapter 8

Aggregating and Pivoting Data

This chapter covers various data-aggregation techniques, including using the OVER clause with aggregate functions, tiebreakers, running aggregates, pivoting, unpivoting, custom aggregations, histograms, grouping factors, and grouping sets.

In my solutions in this chapter, I'll reuse techniques that I introduced earlier. I'll also introduce new techniques for you to familiarize yourself with.

Logic will naturally be an integral element in the solutions. Remember that at the heart of every querying problem lies a logical puzzle.

OVER Clause

The OVER clause allows you to request window-based calculations—that is, calculations performed over a whole window of rows. In Chapter 6, "Subqueries, Table Expressions, and Ranking Functions," I described in detail how you use the OVER clause with analytical ranking functions. Microsoft SQL Server also supports the OVER clause with scalar aggregate functions; however, currently you can provide only the PARTITION BY clause. Future versions of SQL Server will most likely also support the other ANSI elements of aggregate window functions, including the ORDER BY and ROWS clauses.

The purpose of using the OVER clause with scalar aggregates is to calculate, for each row, an aggregate based on a window of values that extends beyond that row—and to do all this without using a GROUP BY clause in the query. In other words, the OVER clause allows you to add aggregate calculations to the results of an ungrouped query. This capability provides an alternative to requesting aggregates with subqueries in case you need to include both base row attributes and aggregates in your results.

Remember that in Chapter 7, "Joins and Set Operations," I presented a problem in which you were required to calculate two aggregates for each order row: the percentage the row contributed to the total value of all orders and the difference between the row's order value and the average value over all orders. In my examples I used a table called MyOrderValues that you create and populate by running the following code:

```
SET NOCOUNT ON;
USE InsideTSQL2008;

IF OBJECT_ID('dbo.MyOrderValues', 'U') IS NOT NULL
  DROP TABLE dbo.MyOrderValues;
GO
```

```
SELECT *
INTO dbo.MyOrderValues
FROM Sales.OrderValues;

ALTER TABLE dbo.MyOrderValues
  ADD CONSTRAINT PK_MyOrderValues PRIMARY KEY(orderid);

CREATE INDEX idx_val ON dbo.MyOrderValues(val);
```

I showed the following optimized query in which I used a cross join between the base table and a derived table of aggregates instead of using multiple subqueries:

```
SELECT orderid, custid, val,
  CAST(val / sumval * 100. AS NUMERIC(5, 2)) AS pct,
  CAST(val - avgval AS NUMERIC(12, 2)) AS diff
FROM dbo.MyOrderValues
  CROSS JOIN (SELECT SUM(val) AS sumval, AVG(val) AS avgval
              FROM dbo.MyOrderValues) AS Aggs;
```

This query produces the following output:

```
orderid  custid  val       pct   diff
-------- ------- --------- ----- -------------
10248    85      440.00    0.03  -1085.05
10249    79      1863.40   0.15  338.35
10250    34      1552.60   0.12  27.55
10251    84      654.06    0.05  -870.99
10252    76      3597.90   0.28  2072.85
10253    34      1444.80   0.11  -80.25
10254    14      556.62    0.04  -968.43
10255    68      2490.50   0.20  965.45
10256    88      517.80    0.04  -1007.25
...
```

The motivation for calculating the two aggregates in a single derived table instead of as two separate subqueries stemmed from the fact that each subquery accessed the base table separately, while the derived table calculated the aggregates using a single scan of the data. SQL Server's query optimizer didn't use the fact that the two subqueries aggregated the same data into the same groups.

When you specify multiple aggregates with identical OVER clauses in the same SELECT list, however, the aggregates refer to the same window, as with a derived table, and SQL Server's query optimizer evaluates them all with one scan of the source data. Here's how you use the OVER clause to answer the same request:

```
SELECT orderid, custid, val,
  CAST(val / SUM(val) OVER() * 100. AS NUMERIC(5, 2)) AS pct,
  CAST(val - AVG(val) OVER() AS NUMERIC(12, 2)) AS diff
FROM dbo.MyOrderValues;
```

Note In Chapter 6, I described the PARTITION BY clause, which is used with window functions, including aggregate window functions. This clause is optional. When not specified, the aggregate is based on the whole input rather than being calculated per partition.

Here, because I didn't specify a PARTITION BY clause, the aggregates were calculated based on the whole input. Logically, *SUM(val) OVER()* is equivalent here to the subquery *(SELECT SUM(val) FROM dbo.MyOrderValues)*. Physically, it's a different story. As an exercise, you can compare the execution plans of the following two queries, each requesting a different number of aggregates using the same OVER clause:

```
SELECT orderid, custid, val,
  SUM(val) OVER() AS sumval
FROM dbo.MyOrderValues;

SELECT orderid, custid, val,
  SUM(val)   OVER() AS sumval,
  COUNT(val) OVER() AS cntval,
  AVG(val)   OVER() AS avgval,
  MIN(val)   OVER() AS minval,
  MAX(val)   OVER() AS maxval
FROM dbo.MyOrderValues;
```

You'll find the two plans nearly identical, with the only difference being that the single Stream Aggregate operator calculates a different number of aggregates. The query costs are identical. On the other hand, compare the execution plans of the following two queries, each requesting a different number of aggregates using subqueries:

```
SELECT orderid, custid, val,
  (SELECT SUM(val) FROM dbo.MyOrderValues) AS sumval
FROM dbo.MyOrderValues;

SELECT orderid, custid, val,
  (SELECT SUM(val)   FROM dbo.MyOrderValues) AS sumval,
  (SELECT COUNT(val) FROM dbo.MyOrderValues) AS cntval,
  (SELECT AVG(val)   FROM dbo.MyOrderValues) AS avgval,
  (SELECT MIN(val)   FROM dbo.MyOrderValues) AS minval,
  (SELECT MAX(val)   FROM dbo.MyOrderValues) AS maxval
FROM dbo.MyOrderValues;
```

You'll find that they have different plans, with the latter being more expensive because it rescans the source data for each aggregate.

Another benefit of the OVER clause is that it allows for shorter and simpler code. This is especially apparent when you need to calculate partitioned aggregates. Using OVER, you simply specify a PARTITION BY clause. Using subqueries, you have to correlate the inner query to the outer, making the query longer and more complex.

As an example of using the PARTITION BY clause, the following query calculates the percentage of the order value out of the customer total and the difference from the customer average:

```
SELECT orderid, custid, val,
  CAST(val / SUM(val) OVER(PARTITION BY custid) * 100.
    AS NUMERIC(5, 2)) AS pct,
  CAST(val - AVG(val) OVER(PARTITION BY custid) AS NUMERIC(12, 2)) AS diff
FROM dbo.MyOrderValues
ORDER BY custid;
```

This query generates the following output:

```
orderid  custid  val     pct    diff
-------- ------- ------- ------ ------------
10643    1       814.50  19.06  102.33
10692    1       878.00  20.55  165.83
10702    1       330.00  7.72   -382.17
10835    1       845.80  19.79  133.63
10952    1       471.20  11.03  -240.97
11011    1       933.50  21.85  221.33
10926    2       514.40  36.67  163.66
10759    2       320.00  22.81  -30.74
10625    2       479.75  34.20  129.01
10308    2       88.80   6.33   -261.94
...
```

In short, the OVER clause allows for more concise and faster-running queries.

When you're done, run the following code for cleanup:

```
IF OBJECT_ID('dbo.MyOrderValues', 'U') IS NOT NULL
  DROP TABLE dbo.MyOrderValues;
```

Tiebreakers

In this section, I want to introduce a new technique based on aggregates to solve tiebreaker problems, which I started discussing in Chapter 6. I'll use the same example as I used there—returning the most recent order for each employee—using different combinations of tiebreaker attributes that uniquely identify an order for each employee. Keep in mind that the performance of the solutions that use subqueries depends very strongly on indexing. That is, you need an index on the partitioning column, sort column, and tiebreaker attributes. But in practice, you don't always have the option of adding as many indexes as you like. The subquery-based solutions will greatly suffer in performance from a lack of appropriate indexes. Using aggregation techniques, you'll see that the solution yields reasonable performance even when an optimal index is not in place—in fact, even when no good index is in place.

Let's start by using *MAX(orderid)* as the tiebreaker. To recap, you're after the most recent order for each employee, and if there's a tie for most recent, choose the order with the largest ID. For each employee's most recent order, you're supposed to return the columns *empid*, *orderdate*, *orderid*, *custid*, and *requireddate*.

The aggregate technique to solve the problem applies the following logical idea, given here in pseudocode:

```
SELECT empid, MAX(orderdate, orderid, custid, requireddate)
FROM Sales.Orders
GROUP BY empid;
```

This idea can't be expressed directly in T-SQL, so don't try to run this query. The idea here is to select for each *empid*, the row with largest *orderdate* (most recent), then largest *orderid*—the tiebreaker—among orders with the most recent *orderdate*. Because the combination *empid*, *orderdate*, *orderid* is already unique, there will be no further ties to break, and the other attributes (*custid* and *requireddate*) are simply returned from the selected row. Because a MAX of more than one attribute does not exist in T-SQL, you must mimic it somehow. One way is by merging the attributes into a single input to the MAX function, then extracting back the individual elements in an outer query.

The question is this: What technique should you use to merge the attributes? The trick is to convert each attribute to a fixed-width string and concatenate the strings. You must convert the attributes to strings in a way that doesn't change the sorting order. When dealing exclusively with nonnegative numbers, you can get by with an arithmetic calculation instead of concatenation. For example, say you have the numbers *m* and *n*, each with a valid range of 1 through 999. To merge *m* and *n*, use the following formula: *m*1000 + n AS r*. You can easily extract the individual pieces later: *r* divided by 1000 is *m*, and *r* modulo 1000 is *n*. However, in many cases you may have nonnumeric data to concatenate, so arithmetic wouldn't be possible. You might want to consider converting all values to fixed-width character strings (*CHAR(n)* or *NCHAR(n)*) or to fixed-width binary strings (*BINARY(n)*).

Here's an example of returning the most recent order for each employee, where *MAX(orderid)* is the tiebreaker, using binary concatenation:

```
SELECT empid,
  CAST(SUBSTRING(binstr, 1,  8) AS DATETIME) AS orderdate,
  CAST(SUBSTRING(binstr, 9,  4) AS INT)      AS orderid,
  CAST(SUBSTRING(binstr, 13, 4) AS INT)      AS custid,
  CAST(SUBSTRING(binstr, 17, 8) AS DATETIME) AS requireddate
FROM (SELECT empid,
        MAX(CAST(orderdate        AS BINARY(8))
              + CAST(orderid      AS BINARY(4))
              + CAST(custid       AS BINARY(4))
              + CAST(requireddate AS BINARY(8))) AS binstr
      FROM Sales.Orders
      GROUP BY empid) AS D;
```

The derived table D contains the maximum concatenated string for each employee. Notice that each value was converted to the appropriate fixed-size string before concatenation based on its data type (DATETIME—8 bytes, INT—4 bytes, and so on).

Note When you convert numbers to binary strings, only nonnegative values preserve their original sort order. As for DATETIME values, as long as they are not earlier than the base date January 1st, 1900, when converted to binary, the values preserve the original sort behavior. Values of the new DATE data type, however, do not preserve their sort behavior when converted to binary. As for character strings, converting them to binary values changes their sort order to one like a binary collation would define. Also note that preserving the original sort order is required only up to the point where uniqueness of a row per group is guaranteed (*orderdate* + *orderid* in our case).

The outer query uses *SUBSTRING* to extract the individual elements, and it converts them back to their original data types.

The real benefit in this solution is that it scans the data only once regardless of whether you have a good index. If you do, you'll probably get an ordered scan of the index and a sort-based aggregate (a stream aggregate). If you don't have a good index—as is the case here—you'll probably get a hash-based aggregate, as you can see in Figure 8-1.

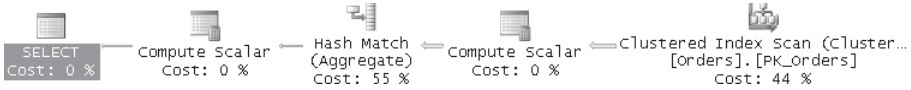


FIGURE 8-1 Execution plan for a tiebreaker query

Things get trickier when the sort columns and tiebreaker attributes have different sort directions within them. For example, suppose the tiebreaker was *MIN(orderid)*. In that case, you would need to apply *MAX* to *orderdate* and *MIN* to *orderid*. There is a logical solution when the attribute with the opposite direction is numeric. Say you need to calculate the *MIN* value of a nonnegative integer column *n*, using only *MAX*, and you need to use binary concatenation. You can get the minimum by using *<maxint> - MAX(<maxint> - n)*.

The following query incorporates this logical technique:

```
SELECT empid,
  CAST(SUBSTRING(binstr, 1, 8) AS DATETIME)         AS orderdate,
  2147483647 - CAST(SUBSTRING(binstr, 9, 4) AS INT) AS orderid,
  CAST(SUBSTRING(binstr, 13, 4) AS INT)             AS custid,
  CAST(SUBSTRING(binstr, 17, 8) AS DATETIME)        AS requireddate
FROM (SELECT empid,
        MAX(CAST(orderdate               AS BINARY(8))
              + CAST(2147483647 - orderid AS BINARY(4))
              + CAST(custid              AS BINARY(4))
              + CAST(requireddate        AS BINARY(8))) AS binstr
      FROM Sales.Orders
      GROUP BY empid) AS D;
```

Another technique to calculate the minimum by using the MAX function is based on bitwise manipulation and works with nonnegative integers. The minimum value of a column *n* is equal to *~MAX(~n)*, where ~ is the bitwise NOT operator.

The following query incorporates this technique:

```
SELECT empid,
  CAST(SUBSTRING(binstr, 1, 8) AS DATETIME)  AS orderdate,
  ~CAST(SUBSTRING(binstr, 9, 4) AS INT)      AS orderid,
  CAST(SUBSTRING(binstr, 13, 4) AS INT)      AS custid,
  CAST(SUBSTRING(binstr, 17, 8) AS DATETIME) AS requireddate
FROM (SELECT empid,
        MAX(CAST(orderdate        AS BINARY(8))
              + CAST(~orderid     AS BINARY(4))
              + CAST(custid       AS BINARY(4))
              + CAST(requireddate AS BINARY(8))) AS binstr
      FROM Sales.Orders
      GROUP BY empid) AS D;
```

Of course, you can play with the tiebreakers you're using in any way you like. For example, the following query returns the most recent order for each employee, using *MAX(requireddate), MAX(orderid)* as the tiebreaker:

```
SELECT empid,
  CAST(SUBSTRING(binstr, 1, 8)   AS DATETIME) AS orderdate,
  CAST(SUBSTRING(binstr, 9, 8)   AS DATETIME) AS requireddate,
  CAST(SUBSTRING(binstr, 17, 4)  AS INT)      AS orderid,
  CAST(SUBSTRING(binstr, 21, 4)  AS INT)      AS custid
FROM (SELECT empid,
        MAX(CAST(orderdate        AS BINARY(8))
              + CAST(requireddate AS BINARY(8))
              + CAST(orderid      AS BINARY(4))
              + CAST(custid       AS BINARY(4))
              ) AS binstr
      FROM Sales.Orders
      GROUP BY empid) AS D;
```

Running Aggregations

Running aggregations are aggregations of data over a sequence (typically temporal). Running aggregate problems have many variations, and I'll describe several important ones here.

In my examples, I'll use a summary table called EmpOrders that contains one row for each employee and month, with the total quantity of orders made by that employee in that month. Run the following code to create the EmpOrders table and populate it with sample data:

```
USE tempdb;

IF OBJECT_ID('dbo.EmpOrders') IS NOT NULL DROP TABLE dbo.EmpOrders;

CREATE TABLE dbo.EmpOrders
(
  empid    INT  NOT NULL,
  ordmonth DATE NOT NULL,
```

```
  qty      INT  NOT NULL,
  PRIMARY KEY(empid, ordmonth)
);
GO

INSERT INTO dbo.EmpOrders(empid, ordmonth, qty)
  SELECT O.empid,
    DATEADD(month, DATEDIFF(month, 0, O.orderdate), 0) AS ordmonth,
    SUM(qty) AS qty
  FROM InsideTSQL2008.Sales.Orders AS O
    JOIN InsideTSQL2008.Sales.OrderDetails AS OD
      ON O.orderid = OD.orderid
  GROUP BY empid,
    DATEADD(month, DATEDIFF(month, 0, O.orderdate), 0);
```

Tip I will represent each month by its start date stored as a DATE. This allows flexible manipulation of the data using date-related functions. Of course, I'll ignore the day part of the value in my calculations.

Run the following query to get the contents of the EmpOrders table:

```
SELECT empid, CONVERT(VARCHAR(7), ordmonth, 121) AS ordmonth, qty
FROM dbo.EmpOrders
ORDER BY empid, ordmonth;
```

This query generates the following output, shown here in abbreviated form:

```
empid       ordmonth qty
----------- -------- -----------
1           2006-07  121
1           2006-08  247
1           2006-09  255
1           2006-10  143
1           2006-11  318
1           2006-12  536
1           2007-01  304
1           2007-02  168
1           2007-03  275
1           2007-04  20
...
2           2006-07  50
2           2006-08  94
2           2006-09  137
2           2006-10  248
2           2006-11  237
2           2006-12  319
2           2007-01  230
2           2007-02  36
2           2007-03  151
2           2007-04  468
...
```

I'll discuss three types of running aggregation problems: cumulative, sliding, and year-to-date (YTD).

Cumulative Aggregations

Cumulative aggregations accumulate data from the first element within the sequence up to the current point. For example, imagine the following request: for each employee and month, return the total quantity and average monthly quantity from the beginning of the employee's activity through the month in question.

Recall the techniques for calculating row numbers without using the built-in ROW_NUMBER function; using these techniques, you scan the same rows we need here to calculate the total quantities. The difference is that for row numbers you used the aggregate COUNT, and here you'll use the aggregates SUM and AVG. I demonstrated two set-based solutions to calculate row numbers without the ROW_NUMBER function—one using subqueries and one using joins. In the solution using joins, I applied what I called an *expand-collapse technique*. To me, the subquery solution is much more intuitive than the join solution, with its artificial expand-collapse technique. So, when there's no performance difference, I'd rather use subqueries. Typically, you won't see a performance difference when only one aggregate is involved because the plans would be similar. However, when you request multiple aggregates, the subquery solution might result in a plan that scans the data separately for each aggregate. Compare this to the plan for the join solution, which typically calculates all aggregates during a single scan of the source data.

So my choice is usually simple—use a subquery for one aggregate and use a join for multiple aggregates. The following query applies the expand-collapse approach to produce the desired result:

```
SELECT O1.empid, CONVERT(VARCHAR(7), O1.ordmonth, 121) AS ordmonth,
  O1.qty AS qtythismonth, SUM(O2.qty) AS totalqty,
  CAST(AVG(1.*O2.qty) AS NUMERIC(12, 2)) AS avgqty
FROM dbo.EmpOrders AS O1
  JOIN dbo.EmpOrders AS O2
    ON O2.empid = O1.empid
    AND O2.ordmonth <= O1.ordmonth
GROUP BY O1.empid, O1.ordmonth, O1.qty
ORDER BY O1.empid, O1.ordmonth;
```

This query generates the following output, shown here in abbreviated form:

```
empid       ordmonth qtythismonth totalqty    avgqty
----------- -------- ------------ ----------- ----------
1           2006-07  121          121         121.00
1           2006-08  247          368         184.00
1           2006-09  255          623         207.67
1           2006-10  143          766         191.50
1           2006-11  318          1084        216.80
1           2006-12  536          1620        270.00
1           2007-01  304          1924        274.86
1           2007-02  168          2092        261.50
1           2007-03  275          2367        263.00
1           2007-04  20           2387        238.70
...
```

```
2           2006-07  50          50          50.00
2           2006-08  94          144         72.00
2           2006-09  137         281         93.67
2           2006-10  248         529         132.25
2           2006-11  237         766         153.20
2           2006-12  319         1085        180.83
2           2007-01  230         1315        187.86
2           2007-02  36          1351        168.88
2           2007-03  151         1502        166.89
2           2007-04  468         1970        197.00
...
```

Now let's say that you are asked to return only one aggregate (say, total quantity). You can safely use the subquery approach:

```
SELECT O1.empid, CONVERT(VARCHAR(7), O1.ordmonth, 121) AS ordmonth,
  O1.qty AS qtythismonth,
  (SELECT SUM(O2.qty)
   FROM dbo.EmpOrders AS O2
   WHERE O2.empid = O1.empid
     AND O2.ordmonth <= O1.ordmonth) AS totalqty
FROM dbo.EmpOrders AS O1
GROUP BY O1.empid, O1.ordmonth, O1.qty;
```

As was the case for calculating row numbers based on subqueries or joins, when calculating running aggregates based on similar techniques, the N^2 performance issues I discussed before apply once again. Because running aggregates typically are calculated on a fairly small number of rows per group, you won't be adversely affected by performance issues, assuming you have appropriate indexes (keyed on grouping columns, then sort columns, and including covering columns).

Let p be the number of partitions involved (employees in our case), let n be the average number of rows per partition (months in our case), and let a be the number of aggregates involved. The total number of rows scanned using the join approach can be expressed as $pn + p(n+n^2)/2$ and as $pn + ap(n+n^2)/2$ using the subquery approach because with subqueries the optimizer uses a separate scan per subquery. It's important to note that the N^2 complexity is relevant to the partition size and not the table size. If the number of rows in the table grows by a factor of f but the partition size doesn't change, the run time increases by a factor of f as well. If, on the other hand, the average partition size grows by a factor of f, the run time increases by a factor of f^2. With small partitions (say, up to several dozen rows), this set-based solution provides reasonable performance. With large partitions, a cursor solution would be faster despite the overhead associated with row-by-row manipulation because a cursor scans the rows only once, and the per-row overhead is constant.

Note ANSI SQL provides support for running aggregates by means of aggregate window functions. SQL Server 2005 introduced the OVER clause for aggregate functions only with the PARTITION BY clause, and unfortunately SQL Server 2008 didn't enhance the OVER clause further. Further enhancements are currently planned for the next major release of SQL Server—SQL

Server 11. Per ANSI SQL—and I hope in future versions of SQL Server—you could provide a solution relying exclusively on window functions, like so:

```
SELECT empid, CONVERT(VARCHAR(7), ordmonth, 121) AS ordmonth, qty,
  SUM(O2.qty) OVER(PARTITION BY empid
                   ORDER BY ordmonth
                   ROWS BETWEEN UNBOUNDED PRECEDING
                            AND CURRENT ROW) AS totalqty,
  CAST(AVG(1.*O2.qty) OVER(PARTITION BY empid
                           ORDER BY ordmonth
                           ROWS BETWEEN UNBOUNDED PRECEDING
                                    AND CURRENT ROW)
  AS NUMERIC(12, 2)) AS avgqty
FROM dbo.EmpOrders;
```

When this code is finally supported in SQL Server, you can expect dramatic performance improvements and obviously much simpler queries. Being familiar with the way ranking calculations based on the OVER clause are currently optimized, you should expect running aggregates based on the OVER clause to be optimized similarly. That is, given a good index to support the request, you should expect the plan to involve a single ordered scan of the data. Then the total number of rows scanned would simply be the number of rows in the table (*pn*).

You might also be requested to filter the data—for example, return monthly aggregates for each employee only for months before the employee reached a certain target. Typically, you'll have a target for each employee stored in a Targets table that you'll need to join to. To make this example simple, I'll assume that all employees have the same target total quantity—1,000. In practice, you'll use the target attribute from the Targets table. Because you need to filter an aggregate, not an attribute, you must specify the filter expression (in this case, *SUM(O2.qty) < 1000*) in the HAVING clause, not the WHERE clause. The solution is as follows:

```
SELECT O1.empid, CONVERT(VARCHAR(7), O1.ordmonth, 121) AS ordmonth,
  O1.qty AS qtythismonth, SUM(O2.qty) AS totalqty,
  CAST(AVG(1.*O2.qty) AS NUMERIC(12, 2)) AS avgqty
FROM dbo.EmpOrders AS O1
  JOIN dbo.EmpOrders AS O2
    ON O2.empid = O1.empid
    AND O2.ordmonth <= O1.ordmonth
GROUP BY O1.empid, O1.ordmonth, O1.qty
HAVING SUM(O2.qty) < 1000
ORDER BY O1.empid, O1.ordmonth;
```

This query generates the following output, shown here in abbreviated form:

```
empid       ordmonth qtythismonth totalqty    avgqty
----------- -------- ------------ ----------- ----------
1           2006-07  121          121         121.00
1           2006-08  247          368         184.00
1           2006-09  255          623         207.67
1           2006-10  143          766         191.50
2           2006-07  50           50          50.00
2           2006-08  94           144         72.00
2           2006-09  137          281         93.67
2           2006-10  248          529         132.25
```

```
2           2006-11  237          766         153.20
3           2006-07  182          182         182.00
3           2006-08  228          410         205.00
3           2006-09  75           485         161.67
3           2006-10  151          636         159.00
3           2006-11  204          840         168.00
3           2006-12  100          940         156.67
...
```

Things get a bit tricky if you also need to include the rows for those months in which the employees reached their target. If you specify *SUM(O2.qty) <= 1000* (that is, write <= instead of <), you still won't get the row in which the employee reached the target unless the total through that month is exactly 1,000. But remember that you have access to both the cumulative total and the current month's quantity, and using these two values together, you can solve this problem. If you change the HAVING filter to *SUM(O2.qty) – O1.qty < 1000*, you get the months in which the employee's total quantity, *excluding the current month's orders*, had not reached the target. In particular, the first month in which an employee reached or exceeded the target satisfies this new criterion, and that month will appear in the results. The complete solution follows:

```
SELECT O1.empid, CONVERT(VARCHAR(7), O1.ordmonth, 121) AS ordmonth,
  O1.qty AS qtythismonth, SUM(O2.qty) AS totalqty,
  CAST(AVG(1.*O2.qty) AS NUMERIC(12, 2)) AS avgqty
FROM dbo.EmpOrders AS O1
  JOIN dbo.EmpOrders AS O2
    ON O2.empid = O1.empid
    AND O2.ordmonth <= O1.ordmonth
GROUP BY O1.empid, O1.ordmonth, O1.qty
HAVING SUM(O2.qty) - O1.qty < 1000
ORDER BY O1.empid, O1.ordmonth;
```

This query generates the following output, shown here in abbreviated form:

```
empid       ordmonth qtythismonth totalqty    avgqty
----------- -------- ------------ ----------- ----------
1           2006-07  121          121         121.00
1           2006-08  247          368         184.00
1           2006-09  255          623         207.67
1           2006-10  143          766         191.50
1           2006-11  318          1084        216.80
2           2006-07  50           50          50.00
2           2006-08  94           144         72.00
2           2006-09  137          281         93.67
2           2006-10  248          529         132.25
2           2006-11  237          766         153.20
2           2006-12  319          1085        180.83
3           2006-07  182          182         182.00
3           2006-08  228          410         205.00
3           2006-09  75           485         161.67
3           2006-10  151          636         159.00
3           2006-11  204          840         168.00
3           2006-12  100          940         156.67
3           2007-01  364          1304        186.29
...
```

Note You might have another solution in mind that seems like a plausible and simpler alternative—to leave the SUM condition alone but change the join condition to *O2.ordmonth* < *O1.ordmonth*. This way, the query would select rows where the total through the previous month did not meet the target. However, in the end, this solution is not any easier (the AVG is hard to generate, for example); what's worse is that you might come up with a solution that does not work for employees who reach the target in their first month.

Tip If you want to return no fewer than a certain number of rows per partition, simply add the criterion *OR COUNT(*) <= <min_num_of_rows>* to the HAVING clause. This technique works well in our case since the base table contains one row per result row/group.

Suppose you're interested in seeing results only for the specific month in which the employee reached the target of 1,000, without seeing results for preceding months. What's true for only those rows in the output of the last query? You're looking for rows where the total quantity is greater than or equal to 1,000. Simply add this criterion to the HAVING filter. Here's the query followed by its output:

```
SELECT O1.empid, CONVERT(VARCHAR(7), O1.ordmonth, 121) AS ordmonth,
  O1.qty AS qtythismonth, SUM(O2.qty) AS totalqty,
  CAST(AVG(1.*O2.qty) AS NUMERIC(12, 2)) AS avgqty
FROM dbo.EmpOrders AS O1
  JOIN dbo.EmpOrders AS O2
    ON O2.empid = O1.empid
    AND O2.ordmonth <= O1.ordmonth
GROUP BY O1.empid, O1.ordmonth, O1.qty
HAVING SUM(O2.qty) - O1.qty < 1000
  AND SUM(O2.qty) >= 1000
ORDER BY O1.empid, O1.ordmonth;

empid       ordmonth qtythismonth totalqty    avgqty
----------- -------- ------------ ----------- ----------
1           2006-11  318          1084        216.80
2           2006-12  319          1085        180.83
3           2007-01  364          1304        186.29
4           2006-10  613          1439        359.75
5           2007-05  247          1213        173.29
6           2007-01  64           1027        171.17
7           2007-03  191          1069        152.71
8           2007-01  305          1228        175.43
9           2007-06  161          1007        125.88
```

Sliding Aggregations

Sliding aggregates are calculated over a sliding window in a sequence (again, typically temporal), as opposed to being calculated from the beginning of the sequence until the current point. A *moving average*—such as the employee's average quantity over the last three months—is one example of a sliding aggregate.

Note Without clarification, expressions such as "last three months" are ambiguous. The last three months could mean the previous three months (*not including this month*), or it could mean the previous two months *along with this month*. When you get a problem like this, be sure you know precisely what window of time you are using for aggregation—for a particular row, exactly when does the window begin and end?

In our example, the window of time is this: greater than the point in time starting three months ago and smaller than or equal to the current point in time. Note that this definition works well even in cases where you track finer time granularities than a month (including day, hour, minute, second, millisecond, microsecond, and nanosecond). This definition also addresses implicit conversion issues resulting from the accuracy level supported by SQL Server for the *DATETIME* data type—3.33 milliseconds. To avoid implicit conversion issues, it's wiser to use > and <= predicates than the BETWEEN predicate.

The main difference between the solution for cumulative aggregates and the solution for sliding aggregates is in the join condition (or in the subquery's filter in the case of the alternate solution using subqueries). Instead of using *O2.ordmonth* <= *O1.current_month*, you use *O2.ordmonth* > *three_months_before_current AND O2.ordmonth* <= *O1.current_month*. In T-SQL, this translates to the following query:

```
SELECT O1.empid,
  CONVERT(VARCHAR(7), O1.ordmonth, 121) AS tomonth,
  O1.qty AS qtythismonth,
  SUM(O2.qty) AS totalqty,
  CAST(AVG(1.*O2.qty) AS NUMERIC(12, 2)) AS avgqty
FROM dbo.EmpOrders AS O1
  JOIN dbo.EmpOrders AS O2
    ON O2.empid = O1.empid
    AND (O2.ordmonth > DATEADD(month, -3, O1.ordmonth)
         AND O2.ordmonth <=  O1.ordmonth)
GROUP BY O1.empid, O1.ordmonth, O1.qty
ORDER BY O1.empid, O1.ordmonth;
```

This query generates the following output, shown here in abbreviated form:

```
empid       tomonth qtythismonth totalqty    avgqty
----------- ------- ------------ ----------- ----------
1           2006-07 121          121         121.00
1           2006-08 247          368         184.00
1           2006-09 255          623         207.67
1           2006-10 143          645         215.00
1           2006-11 318          716         238.67
1           2006-12 536          997         332.33
1           2007-01 304          1158        386.00
1           2007-02 168          1008        336.00
1           2007-03 275          747         249.00
1           2007-04 20           463         154.33
...
2           2006-07 50           50          50.00
2           2006-08 94           144         72.00
2           2006-09 137          281         93.67
2           2006-10 248          479         159.67
2           2006-11 237          622         207.33
```

```
2           2006-12 319          804         268.00
2           2007-01 230          786         262.00
2           2007-02 36           585         195.00
2           2007-03 151          417         139.00
2           2007-04 468          655         218.33
...
```

Note that this solution includes aggregates for three-month periods that don't include three months of actual data. If you want to return only periods with three full months accumulated, without the first two periods that do not cover three months, you can add the criterion *MIN(O2.ordmonth) = DATEADD(month, –2, O1.ordmonth)* to the HAVING filter.

Note Per ANSI SQL, you can use the ORDER BY and ROWS subclauses of the OVER clause—which are currently missing in SQL Server—to address sliding aggregates. You would use the following query to return the desired result for the last sliding aggregates request (assuming the data has exactly one row per month):

```
SELECT empid, CONVERT(VARCHAR(7), ordmonth, 121) AS ordmonth,
  qty AS qtythismonth,
  SUM(O2.qty) OVER(PARTITION BY empid
                   ORDER BY ordmonth
                   ROWS BETWEEN 2 PRECEDING
                            AND CURRENT ROW) AS totalqty,
  CAST(AVG(1.*O2.qty) OVER(PARTITION BY empid
                           ORDER BY ordmonth
                           ROWS BETWEEN 2 PRECEDING
                                    AND CURRENT ROW)
    AS NUMERIC(12, 2)) AS avgqty
FROM dbo.EmpOrders;
```

Year-to-Date (YTD)

YTD aggregates accumulate values from the beginning of a period based on some date and time unit (say, a year) until the current point. The calculation is very similar to the sliding aggregates solution. The only difference is the lower bound provided in the query's filter, which is the calculation of the beginning of the year. For example, the following query returns YTD aggregates for each employee and month:

```
SELECT O1.empid,
  CONVERT(VARCHAR(7), O1.ordmonth, 121) AS ordmonth,
  O1.qty AS qtythismonth,
  SUM(O2.qty) AS totalqty,
  CAST(AVG(1.*O2.qty) AS NUMERIC(12, 2)) AS avgqty
FROM dbo.EmpOrders AS O1
  JOIN dbo.EmpOrders AS O2
    ON O2.empid = O1.empid
    AND (O2.ordmonth >= CAST(CAST(YEAR(O1.ordmonth) AS CHAR(4))
                             + '0101' AS DATETIME)
         AND O2.ordmonth <= O1.ordmonth)
GROUP BY O1.empid, O1.ordmonth, O1.qty
ORDER BY O1.empid, O1.ordmonth;
```

This query generates the following output, shown here in abbreviated form:

```
empid       ordmonth qtythismonth totalqty    avgqty
----------- -------- ------------ ----------- ----------
1           2006-07  121          121         121.00
1           2006-08  247          368         184.00
1           2006-09  255          623         207.67
1           2006-10  143          766         191.50
1           2006-11  318          1084        216.80
1           2006-12  536          1620        270.00
1           2007-01  304          304         304.00
1           2007-02  168          472         236.00
1           2007-03  275          747         249.00
1           2007-04  20           767         191.75
...
2           2006-07  50           50          50.00
2           2006-08  94           144         72.00
2           2006-09  137          281         93.67
2           2006-10  248          529         132.25
2           2006-11  237          766         153.20
2           2006-12  319          1085        180.83
2           2007-01  230          230         230.00
2           2007-02  36           266         133.00
2           2007-03  151          417         139.00
2           2007-04  468          885         221.25
...
```

Pivoting

Pivoting is a technique that allows you to rotate rows to columns, possibly performing aggregations along the way. The number of applications for pivoting is simply astounding. In this section, I'll present a few, including pivoting attributes in an open schema environment, solving relational division problems, and formatting aggregated data. Later in the chapter and also in later chapters in the book, I'll show additional applications.

Pivoting Attributes

I'll use *open schema* as the scenario for pivoting attributes. Open schema is a design problem describing an environment that needs to deal with frequent schema changes. The relational model and SQL were conceived to handle frequent changes and requests for data via SQL's data manipulation language (DML). However, SQL's data definition language (DDL) was not conceived to support frequent schema changes. Whenever you need to add new entities, you must create new tables; whenever existing entities change their structures, you must add, alter, or drop columns. Such changes usually require downtime of the affected objects, and they also bring about substantial revisions to the application.

You can choose from several ways to model an open schema environment, each of which has advantages and disadvantages. One of those models is known as Entity Attribute

Value (EAV) and also as the *narrow* representation of data. In this model, you store all data in a single table, where each attribute value resides in its own row along with the entity or object ID and the attribute name or ID. You represent the attribute values using the data type SQL_VARIANT to accommodate multiple attribute types in a single column.

In my examples, I'll use the OpenSchema table, which you can create and populate by running the following code:

```
USE tempdb;

IF OBJECT_ID('dbo.OpenSchema') IS NOT NULL DROP TABLE dbo.OpenSchema;

CREATE TABLE dbo.OpenSchema
(
  objectid  INT          NOT NULL,
  attribute NVARCHAR(30) NOT NULL,
  value     SQL_VARIANT  NOT NULL,
  PRIMARY KEY (objectid, attribute)
);
GO

INSERT INTO dbo.OpenSchema(objectid, attribute, value) VALUES
  (1, N'attr1', CAST(CAST('ABC'      AS VARCHAR(10))   AS SQL_VARIANT)),
  (1, N'attr2', CAST(CAST(10         AS INT)           AS SQL_VARIANT)),
  (1, N'attr3', CAST(CAST('20070101' AS SMALLDATETIME) AS SQL_VARIANT)),
  (2, N'attr2', CAST(CAST(12         AS INT)           AS SQL_VARIANT)),
  (2, N'attr3', CAST(CAST('20090101' AS SMALLDATETIME) AS SQL_VARIANT)),
  (2, N'attr4', CAST(CAST('Y'        AS CHAR(1))       AS SQL_VARIANT)),
  (2, N'attr5', CAST(CAST(13.7       AS NUMERIC(9,3))  AS SQL_VARIANT)),
  (3, N'attr1', CAST(CAST('XYZ'      AS VARCHAR(10))   AS SQL_VARIANT)),
  (3, N'attr2', CAST(CAST(20         AS INT)           AS SQL_VARIANT)),
  (3, N'attr3', CAST(CAST('20080101' AS SMALLDATETIME) AS SQL_VARIANT));

-- show the contents of the table
SELECT * FROM dbo.OpenSchema;
```

This generates the following output:

```
objectid    attribute  value
----------- ---------- ------------------------
1           attr1      ABC
1           attr2      10
1           attr3      2007-01-01 00:00:00.000
2           attr2      12
2           attr3      2009-01-01 00:00:00.000
2           attr4      Y
2           attr5      13.700
3           attr1      XYZ
3           attr2      20
3           attr3      2008-01-01 00:00:00.000
```

Representing data this way allows logical schema changes to be implemented without adding, altering, or dropping tables and columns—you use DML INSERTs, UPDATEs, and DELETEs instead.

Of course, other aspects of working with the data (such as enforcing integrity, tuning, and querying) become more complex and expensive with such a representation. As mentioned, there are other approaches to dealing with open schema environments—for example, storing the data in XML format, using a *wide* representation of data, using CLR types, and others. However, when you weigh the advantages and disadvantages of each representation, you might find the EAV approach demonstrated here more favorable in some scenarios.

Keep in mind that this representation of the data requires very complex queries even for simple requests because different attributes of the same entity instance are spread over multiple rows. Before you query such data, you might want to rotate it to a traditional form with one column for each attribute—perhaps store the result in a temporary table, index it, query it, and then get rid of the temporary table. To rotate the data from its open schema form into a traditional form, you need to use a pivoting technique.

In the following section, I'll describe the steps involved in solving pivoting problems. I'd like to point out that to understand the steps of the solution, it can be very helpful if you think about query logical processing phases, which I described in detail in Chapter 1, "Logical Query Processing." I discussed the query processing phases involved with the native PIVOT table operator, but those phases apply just as well to the standard solution that does not use this proprietary operator. Moreover, in the standard solution the phases are more apparent in the code, while using the PIVOT operator they are implicit.

The first step you might want to try when solving pivoting problems is to figure out how the number of rows in the result correlates to the number of rows in the source data. Here, you need to create a single result row out of the multiple base rows for each object. In SQL, this translates to grouping rows. So our first logical processing phase in pivoting is a *grouping* phase, and the associated element (the element you need to group by) is the *objectid* column.

As the next step in a pivoting problem, you can think in terms of the result columns. You need a result column for each unique attribute. Because the data contains five unique attributes (*attr1*, *attr2*, *attr3*, *attr4*, and *attr5*), you need five expressions in the SELECT list. Each expression is supposed to extract, out of the rows belonging to the grouped object, the value corresponding to a specific attribute. You can think of this logical phase as a *spreading* phase—you need to spread the values, or shift them, from the source column (value in our case) to the corresponding target column. As for the element that dictates where to spread the values, or the *spread by* element, in our case it is the attribute column. This spreading activity can be done with the following CASE expression, which in this example is applied to the attribute *attr2*:

```
CASE WHEN attribute = 'attr2' THEN value END
```

Remember that with no ELSE clause, CASE assumes an implicit ELSE NULL. The CASE expression just shown yields NULL for rows where *attribute* does not equal *attr2* and yields *value* when *attribute* does equal *attr2*. This means that among the rows with a given value of *objectid* (say, 1), the CASE expression would yield several NULLs and, at most, one known value

(10 in our example), which represents the value of the target attribute (*attr2* in our example) for the given *objectid*.

The third phase in pivoting attributes is to extract the known value (if it exists) out of the set of NULLs and the known value. You have to use an aggregate for this purpose because, as you'll recall, the query involves grouping. The trick to extracting the one known value is to use *MAX* or *MIN*. Both ignore NULLs and will return the one non-NULL value present because both the minimum and the maximum of a set containing one value is that value. So our third logical processing phase in pivoting is an *aggregation* phase. The aggregation element is the value column, and the aggregate function is *MAX*. Using the previous expression implementing the second phase with *attr2*, here's the revised expression including the aggregation as well:

```
MAX(CASE WHEN attribute = 'attr2' THEN value END) AS attr2
```

Here's the complete query that pivots the attributes from OpenSchema:

```
SELECT objectid,
  MAX(CASE WHEN attribute = 'attr1' THEN value END) AS attr1,
  MAX(CASE WHEN attribute = 'attr2' THEN value END) AS attr2,
  MAX(CASE WHEN attribute = 'attr3' THEN value END) AS attr3,
  MAX(CASE WHEN attribute = 'attr4' THEN value END) AS attr4,
  MAX(CASE WHEN attribute = 'attr5' THEN value END) AS attr5
FROM dbo.OpenSchema
GROUP BY objectid;
```

This query generates the following output:

```
objectid    attr1      attr2      attr3                   attr4      attr5
----------- ---------- ---------- ----------------------- ---------- ----------
1           ABC        10         2007-01-01 00:00:00.000 NULL       NULL
2           NULL       12         2009-01-01 00:00:00.000 Y          13.700
3           XYZ        20         2008-01-01 00:00:00.000 NULL       NULL
```

Note To write this query, you have to know the names of the attributes. If you don't, you'll need to construct the query string dynamically. I'll provide an example later in the chapter.

This technique for pivoting data is very efficient because it scans the base table only once.

SQL Server supports a native specialized table operator for pivoting called PIVOT. This operator does not provide any special advantages over the technique I just showed, except that it allows for shorter code. It doesn't support dynamic pivoting, and underneath the covers, it applies very similar logic to the one I presented in the last solution. So you probably won't even find noticeable performance differences. At any rate, here's how you would pivot the OpenSchema data using the PIVOT operator:

```
SELECT objectid, attr1, attr2, attr3, attr4, attr5
FROM dbo.OpenSchema
  PIVOT(MAX(value) FOR attribute
    IN([attr1],[attr2],[attr3],[attr4],[attr5])) AS P;
```

Within this solution, you can identify all the elements I used in the previous solution. The inputs to the PIVOT operator are as follows:

- The aggregate function applied to the aggregation element. In our case, it's *MAX(value)*, which extracts the single non-NULL value corresponding to the target attribute. In other cases, you might have more than one non-NULL value per group and want a different aggregate (for example, SUM or AVG).
- Following the FOR keyword, the name of the *spread by* element (*attribute*, in our case). This is the source column holding the values that become the target column names.
- The list of actual target column names in parentheses following the keyword IN.

As you can see, in the parentheses of the PIVOT operator, you specify the aggregate function and aggregation element and the *spread by* element and spreading values but not the *group by* elements. This is a problematic aspect of the syntax of the PIVOT operator—the grouping elements are implicitly derived from what was not specified. The grouping elements are the list of all columns from the input table to the PIVOT operator that were not mentioned as either the aggregation or the spreading elements. In our case, *objectid* is the only column left. If you unintentionally query the base table directly, you might end up with undesired grouping. If new columns will be added to the table in the future, those columns will be implicitly added to PIVOT's grouping list. Therefore, it is strongly recommended that you apply the PIVOT operator not to the base table directly but rather to a table expression (derived table or CTE) that includes only the elements relevant to the pivoting activity. This way, you can control exactly which columns remain besides the aggregation and spreading elements. Future column additions to the table won't have any impact on what PIVOT ends up operating on. The following query demonstrates applying this approach to our previous query, using a derived table:

```
SELECT objectid, attr1, attr2, attr3, attr4, attr5
FROM (SELECT objectid, attribute, value FROM dbo.OpenSchema) AS D
  PIVOT(MAX(value) FOR attribute
    IN([attr1],[attr2],[attr3],[attr4],[attr5])) AS P;
```

Tip The input to the aggregate function must be a base column from the PIVOT operator's input table with no manipulation—it cannot be an expression (for example: *SUM(qty * price)*). If you want to provide the aggregate function with an expression as input, have the PIVOT operator operate on a derived table or CTE (as suggested for other reasons as well), and in the derived table query assign the expression with a column alias (*qty * price AS value*). Then, as far as the PIVOT operator is concerned, that alias is the name of a base column in its input table, so it is valid to use that column name as input to PIVOT's aggregate function (*SUM(value)*).

Also, you cannot spread attributes from more than one column (the column that appears after the FOR keyword). If you need to pivot more than one column's attributes (say, *empid* and *YEAR(orderdate)*), you can use a similar approach to the previous suggestion: in the derived table or CTE used as the input to the PIVOT operator, concatenate the values from all columns you want to use as the spreading elements and assign the expression with a column alias (*CAST(empid AS VARCHAR(10)) + '_' + CAST(YEAR(orderdate) AS CHAR(4)) AS emp_year*). Then, in the outer query, specify that column after PIVOT's FOR keyword (*FOR emp_year IN([1_2007], [1_2008], [1_2009], [2_2007], . . .)*).

Relational Division

You can also use pivoting to solve relational division problems when the number of elements in the divisor set is fairly small. In my examples, I'll use the OrderDetails table, which you create and populate by running the following code:

```
USE tempdb;

IF OBJECT_ID('dbo.OrderDetails') IS NOT NULL
  DROP TABLE dbo.OrderDetails;

CREATE TABLE dbo.OrderDetails
(
  orderid   VARCHAR(10) NOT NULL,
  productid INT         NOT NULL,
  PRIMARY KEY(orderid, productid)
  /* other colums */
);
GO

INSERT INTO dbo.OrderDetails(orderid, productid) VALUES
  ('A', 1),
  ('A', 2),
  ('A', 3),
  ('A', 4),
  ('B', 2),
  ('B', 3),
  ('B', 4),
  ('C', 3),
  ('C', 4),
  ('D', 4);
```

A classic relational division problem is to return orders that contain a certain basket of products—say, products 2, 3, and 4. You use a pivoting technique to rotate only the relevant products into separate columns for each order. Instead of returning an actual attribute value, you produce a 1 if the product exists in the order and a 0 otherwise. Create a derived table out of the pivot query, and in the outer query filter only orders that contain a 1 in all product columns. Here's the full query, which correctly returns orders A and B:

```
SELECT orderid
FROM (SELECT
        orderid,
        MAX(CASE WHEN productid = 2 THEN 1 END) AS P2,
        MAX(CASE WHEN productid = 3 THEN 1 END) AS P3,
        MAX(CASE WHEN productid = 4 THEN 1 END) AS P4
      FROM dbo.OrderDetails
      GROUP BY orderid) AS P
WHERE P2 = 1 AND P3 = 1 AND P4 = 1;
```

If you run only the derived table query, you get the following output with the pivoted products for each order:

```
orderid    P2          P3          P4
---------- ----------- ----------- -----------
A          1           1           1
B          1           1           1
C          NULL        1           1
D          NULL        NULL        1
```

To answer the request at hand using the new PIVOT operator, use the following query:

```
SELECT orderid
FROM (SELECT orderid, productid FROM dbo.OrderDetails) AS D
  PIVOT(MAX(productid) FOR productid IN([2],[3],[4])) AS P
WHERE [2] = 2 AND [3] = 3 AND [4] = 4;
```

The aggregate function must accept a column as input, so I provided the *productid* itself. This means that if the product exists within an order, the corresponding value will contain the actual *productid* and not 1. That's why the filter looks a bit different here.

Note that you can make both queries more intuitive and similar to each other in their logic by using the *COUNT* aggregate instead of *MAX*. This way, both queries would produce a 1 where the product exists and a 0 where it doesn't (instead of NULL). Here's what the query that does not use the PIVOT operator looks like:

```
SELECT orderid
FROM (SELECT
        orderid,
        COUNT(CASE WHEN productid = 2 THEN productid END) AS P2,
        COUNT(CASE WHEN productid = 3 THEN productid END) AS P3,
        COUNT(CASE WHEN productid = 4 THEN productid END) AS P4
      FROM dbo.OrderDetails
      GROUP BY orderid) AS P
WHERE P2 = 1 AND P3 = 1 AND P4 = 1;
```

And here's the query you would use based on the PIVOT operator:

```
SELECT orderid
FROM (SELECT orderid, productid FROM dbo.OrderDetails) AS D
  PIVOT(COUNT(productid) FOR productid IN([2],[3],[4])) AS P
WHERE [2] = 1 AND [3] = 1 AND [4] = 1;
```

Aggregating Data

You can also use a pivoting technique to format aggregated data, typically for reporting purposes. In my examples, I'll use the Orders table, which you create and populate by running the code in Listing 8-1.

LISTING 8-1 Creating and populating the Orders table

```
SET NOCOUNT ON;
USE tempdb;

IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL DROP TABLE dbo.Orders;

CREATE TABLE dbo.Orders
(
  orderid   INT        NOT NULL,
  orderdate DATETIME   NOT NULL,
  empid     INT        NOT NULL,
  custid    VARCHAR(5) NOT NULL,
  qty       INT        NOT NULL,
  CONSTRAINT PK_Orders PRIMARY KEY(orderid)
);
GO

INSERT INTO dbo.Orders
  (orderid, orderdate, empid, custid, qty)
VALUES
  (30001, '20060802', 3, 'A', 10),
  (10001, '20061224', 1, 'A', 12),
  (10005, '20061224', 1, 'B', 20),
  (40001, '20070109', 4, 'A', 40),
  (10006, '20070118', 1, 'C', 14),
  (20001, '20070212', 2, 'B', 12),
  (40005, '20080212', 4, 'A', 10),
  (20002, '20080216', 2, 'C', 20),
  (30003, '20080418', 3, 'B', 15),
  (30004, '20060418', 3, 'C', 22),
  (30007, '20060907', 3, 'D', 30);

-- show the contents of the table
SELECT * FROM dbo.Orders;
```

This generates the following output:

```
orderid     orderdate               empid       custid qty
----------- ----------------------- ----------- ------ -----------
10001       2006-12-24 00:00:00.000 1           A      12
10005       2006-12-24 00:00:00.000 1           B      20
10006       2007-01-18 00:00:00.000 1           C      14
20001       2007-02-12 00:00:00.000 2           B      12
20002       2008-02-16 00:00:00.000 2           C      20
30001       2006-08-02 00:00:00.000 3           A      10
30003       2008-04-18 00:00:00.000 3           B      15
30004       2006-04-18 00:00:00.000 3           C      22
30007       2006-09-07 00:00:00.000 3           D      30
40001       2007-01-09 00:00:00.000 4           A      40
40005       2008-02-12 00:00:00.000 4           A      10
```

Suppose you want to return a row for each customer, with the total yearly quantities in a different column for each year. As with all pivoting problems, it boils down to identifying the grouping, spreading, and aggregation elements. In this case, the grouping element is the *custid* column, the spreading element is the expression *YEAR(orderdate),* and the aggregate function and element is *SUM(qty)*. What remains is simply to use the solution templates I provided previously. Here's the solution that does not use the PIVOT operator, followed by its output:

```
SELECT custid,
  SUM(CASE WHEN orderyear = 2006 THEN qty END) AS [2006],
  SUM(CASE WHEN orderyear = 2007 THEN qty END) AS [2007],
  SUM(CASE WHEN orderyear = 2008 THEN qty END) AS [2008]
FROM (SELECT custid, YEAR(orderdate) AS orderyear, qty
      FROM dbo.Orders) AS D
GROUP BY custid;

custid 2006        2007        2008
------ ----------- ----------- -----------
A      22          40          10
B      20          12          15
C      22          14          20
D      30          NULL        NULL
```

Here you can see the use of a derived table to isolate only the relevant elements for the pivoting activity (*custid*, *orderyear*, *qty*).

One of the main issues with this pivoting solution is that you might end up with lengthy query strings when the number of elements you need to rotate is large. It's not a problem in this case because we are dealing with order years, and there usually aren't that many, but it could be a problem in other cases when the spreading column has a large number of values. In an effort to shorten the query string, you can use a matrix table that contains a column and a row for each attribute that you need to rotate (*orderyear*, in this case). Only column values in the intersections of corresponding rows and columns contain the value 1, and the other column values are populated with a NULL or a 0, depending on your needs. Run the following code to create and populate the Matrix table:

```
USE tempdb;
GO

IF OBJECTPROPERTY(OBJECT_ID('dbo.Matrix'), 'IsUserTable') = 1
  DROP TABLE dbo.Matrix;
GO

CREATE TABLE dbo.Matrix
(
  orderyear INT NOT NULL PRIMARY KEY,
  y2006 INT NULL,
  y2007 INT NULL,
  y2008 INT NULL
);

INSERT INTO dbo.Matrix(orderyear, y2006) VALUES(2006, 1);
INSERT INTO dbo.Matrix(orderyear, y2007) VALUES(2007, 1);
INSERT INTO dbo.Matrix(orderyear, y2008) VALUES(2008, 1);
```

```
-- show the contents of the table
SELECT * FROM dbo.Matrix;
```

This generates the following output:

```
orderyear   y2006       y2007       y2008
----------- ----------- ----------- -----------
2006        1           NULL        NULL
2007        NULL        1           NULL
2008        NULL        NULL        1
```

You join the base table (or table expression) with the Matrix table based on a match in *orderyear*. This means that each row from the base table will be matched with one row from Matrix—the one with the same *orderyear*. In that row, only the corresponding *orderyear*'s column value will contain a 1. So you can substitute the expression

```
SUM(CASE WHEN orderyear = <some_year> THEN qty END) AS [<some_year>]
```

with the logically equivalent expression

```
SUM(qty*y<some_year>) AS [<some_year>]
```

Here's what the full query looks like:

```
SELECT custid,
  SUM(qty*y2006) AS [2006],
  SUM(qty*y2007) AS [2007],
  SUM(qty*y2008) AS [2008]
FROM (SELECT custid, YEAR(orderdate) AS orderyear, qty
      FROM dbo.Orders) AS D
  JOIN dbo.Matrix AS M ON D.orderyear = M.orderyear
GROUP BY custid;
```

If you need the number of orders instead of the sum of *qty*, in the original solution you produce a 1 instead of the *qty* column for each order and use the COUNT aggregate function, like so:

```
SELECT custid,
  COUNT(CASE WHEN orderyear = 2006 THEN 1 END) AS [2006],
  COUNT(CASE WHEN orderyear = 2007 THEN 1 END) AS [2007],
  COUNT(CASE WHEN orderyear = 2008 THEN 1 END) AS [2008]
FROM (SELECT custid, YEAR(orderdate) AS orderyear
      FROM dbo.Orders) AS D
GROUP BY custid;
```

This code generates the following output:

```
custid 2006        2007        2008
------ ----------- ----------- -----------
A      2           1           1
B      1           1           1
C      1           1           1
D      1           0           0
```

With the Matrix table, simply specify the column corresponding to the target year:

```
SELECT custid,
  COUNT(y2006) AS [2006],
  COUNT(y2007) AS [2007],
  COUNT(y2008) AS [2008]
FROM (SELECT custid, YEAR(orderdate) AS orderyear
      FROM dbo.Orders) AS D
  JOIN dbo.Matrix AS M ON D.orderyear = M.orderyear
GROUP BY custid;
```

Of course, using the PIVOT operator, the query strings are pretty much as short as they can get. You don't explicitly specify the CASE expressions: those are constructed behind the scenes for you (you can actually see them by looking at the properties of the aggregate operator in the plan). In short, you don't need to use the Matrix table approach with the PIVOT operator. Here's the query using the PIVOT operator to calculate total yearly quantities per customer:

```
SELECT *
FROM (SELECT custid, YEAR(orderdate) AS orderyear, qty
      FROM dbo.Orders) AS D
  PIVOT(SUM(qty) FOR orderyear IN([2006],[2007],[2008])) AS P;
```

And here's a query that counts the orders:

```
SELECT *
FROM (SELECT custid, YEAR(orderdate) AS orderyear
      FROM dbo.Orders) AS D
  PIVOT(COUNT(orderyear) FOR orderyear IN([2006],[2007],[2008])) AS P;
```

Remember that static queries performing pivoting require you to know ahead of time the list of attributes you're going to rotate. For dynamic pivoting, you need to construct the query string dynamically.

Unpivoting

Unpivoting is the opposite of pivoting—namely, rotating columns to rows. Unpivoting is usually used to normalize data, but it has other applications as well.

Note Unpivoting is not an exact inverse of pivoting—it won't necessarily allow you to regenerate source rows that were pivoted. However, for the sake of simplicity, think of it as the opposite of pivoting.

In my examples, I'll use the PvtCustOrders table, which you create and populate by running the following code:

```
USE tempdb;

IF OBJECT_ID('dbo.PvtCustOrders') IS NOT NULL
  DROP TABLE dbo.PvtCustOrders;
GO
```

```
SELECT custid,
  COALESCE([2006], 0) AS [2006],
  COALESCE([2007], 0) AS [2007],
  COALESCE([2008], 0) AS [2008]
INTO dbo.PvtCustOrders
FROM (SELECT custid, YEAR(orderdate) AS orderyear, qty
      FROM dbo.Orders) AS D
  PIVOT(SUM(qty) FOR orderyear IN([2006],[2007],[2008])) AS P;

UPDATE dbo.PvtCustOrders
  SET [2007] = NULL, [2008] = NULL
WHERE custid = 'D';

-- Show the contents of the table
SELECT * FROM dbo.PvtCustOrders;
```

This generates the following output:

```
custid 2006        2007        2008
------ ----------- ----------- -----------
A      22          40          10
B      20          12          15
C      22          14          20
D      30          NULL        NULL
```

The goal in this case is to generate a result row for each customer and year, containing the customer ID (*custid*), order year (*orderyear*), and quantity (*qty*).

I'll start with a solution that does not use the native UNPIVOT operator. Here as well, try to think in terms of logical query processing as described in Chapter 1.

The first step in the solution is to generate three copies of each base row—one for each year. You can achieve this by performing a cross join between the base table and a virtual auxiliary table that has one row per year. The SELECT list can then return *custid* and *orderyear* and also calculate the target year's *qty* with the following CASE expression:

```
CASE orderyear
  WHEN 2006 THEN [2006]
  WHEN 2007 THEN [2007]
  WHEN 2008 THEN [2008]
END AS qty
```

You achieve unpivoting this way, but you also get rows corresponding to NULL values in the source table (for example, for customer D in years 2007 and 2008). To eliminate those rows, create a derived table out of the solution query and, in the outer query, eliminate the rows with the NULL in the *qty* column.

Note In practice, you'd typically store a 0 and not a NULL as the quantity for a customer with no orders in a certain year; the order quantity is known to be zero and not unknown. However, I used NULLs here to demonstrate the treatment of NULLs, which is a very common need in unpivoting problems.

Here's the complete solution, followed by its output:

```
SELECT custid, orderyear, qty
FROM (SELECT custid, orderyear,
        CASE orderyear
          WHEN 2006 THEN [2006]
          WHEN 2007 THEN [2007]
          WHEN 2008 THEN [2008]
        END AS qty
      FROM dbo.PvtCustOrders
        CROSS JOIN
          (SELECT 2006 AS orderyear
           UNION ALL SELECT 2007
           UNION ALL SELECT 2008) AS OrderYears) AS D
WHERE qty IS NOT NULL;

custid orderyear   qty
------ ----------- -----------
A      2006        22
A      2007        40
A      2008        10
B      2006        20
B      2007        12
B      2008        15
C      2006        22
C      2007        14
C      2008        20
D      2006        30
D      2007        0
D      2008        0
```

As of SQL Server 2008, you can replace the current definition of the derived table D with a table value constructor based on the VALUES clause, like so:

```
SELECT custid, orderyear, qty
FROM (SELECT custid, orderyear,
        CASE orderyear
          WHEN 2006 THEN [2006]
          WHEN 2007 THEN [2007]
          WHEN 2008 THEN [2008]
        END AS qty
      FROM dbo.PvtCustOrders
        CROSS JOIN
          (VALUES(2006),(2007),(2008)) AS OrderYears(orderyear)) AS D
WHERE qty IS NOT NULL;
```

Either way, using the native proprietary UNPIVOT table operator is dramatically simpler, as the following query shows:

```
SELECT custid, orderyear, qty
FROM dbo.PvtCustOrders
  UNPIVOT(qty FOR orderyear IN([2006],[2007],[2008])) AS U;
```

Unlike the PIVOT operator, I find the UNPIVOT operator simple and intuitive, and obviously it requires significantly less code than the alternative solutions. UNPIVOT's first input is the target column name to hold the source column values (*qty*). Then, following the FOR keyword,

you specify the target column name to hold the source column names (*orderyear*). Finally, in the parentheses of the IN clause, you specify the source column names that you want to unpivot ([2006],[2007],[2008]).

Tip All source attributes that are unpivoted must share the same data type. If you want to unpivot attributes defined with different data types, create a derived table or CTE where you first convert all those attributes to SQL_VARIANT. The target column that will hold unpivoted values will also be defined as SQL_VARIANT, and within that column, the values will preserve their original types.

Note Like PIVOT, UNPIVOT requires a static list of column names to be rotated. Also, the UNPIVOT operator applies a logical phase that removes NULL rows. However, unlike in the other solutions where the removal of NULL rows is an optional phase, with the UNPIVOT operator it is not optional.

Custom Aggregations

Custom aggregations are aggregations that are not provided as built-in aggregate functions—for example, concatenating strings, calculating products, performing bitwise manipulations, calculating medians, and others. In this section, I'll provide solutions to several custom aggregate requests. Some techniques that I'll cover are generic, in the sense that you can use similar logic for other aggregate requests; other techniques are specific to one kind of aggregate request.

More Info One of the generic custom aggregate techniques uses cursors. For details about cursors, including handling of custom aggregates with cursors, please refer to *Inside Microsoft SQL Server 2008: T-SQL Programming* (Microsoft Press, 2009).

In my examples, I'll use the generic Groups table, which you create and populate by running the following code:

```
USE tempdb;

IF OBJECT_ID('dbo.Groups') IS NOT NULL DROP TABLE dbo.Groups;

CREATE TABLE dbo.Groups
(
  groupid  VARCHAR(10) NOT NULL,
  memberid INT         NOT NULL,
  string   VARCHAR(10) NOT NULL,
  val      INT         NOT NULL,
  PRIMARY KEY (groupid, memberid)
);
GO
```

```
INSERT INTO dbo.Groups(groupid, memberid, string, val) VALUES
  ('a', 3, 'stra1', 6),
  ('a', 9, 'stra2', 7),
  ('b', 2, 'strb1', 3),
  ('b', 4, 'strb2', 7),
  ('b', 5, 'strb3', 3),
  ('b', 9, 'strb4', 11),
  ('c', 3, 'strc1', 8),
  ('c', 7, 'strc2', 10),
  ('c', 9, 'strc3', 12);

-- Show the contents of the table
SELECT * FROM dbo.Groups;
```

This generates the following output:

```
groupid    memberid    string     val
---------- ----------- ---------- -----------
a          3           stra1      6
a          9           stra2      7
b          2           strb1      3
b          4           strb2      7
b          5           strb3      3
b          9           strb4      11
c          3           strc1      8
c          7           strc2      10
c          9           strc3      12
```

The Groups table has a column representing the group (*groupid*), a column representing a unique identifier within the group (*memberid*), and some value columns (*string* and *val*) that need to be aggregated. I like to use such a generic form of data because it allows you to focus on the techniques and not on the data. Note that this is merely a generic form of a table containing data that you want to aggregate. For example, it could represent a Sales table where *groupid* stands for *empid*, *val* stands for *qty*, and so on.

Custom Aggregations Using Pivoting

One technique for solving custom aggregate problems is pivoting. You pivot the values that need to participate in the aggregate calculation; when they all appear in the same result row, you perform the calculation as a linear one across the columns. With a large number of elements you'll end up with very lengthy query strings; therefore, this pivoting technique is limited to situations where each group has a small number of elements. Note that unless you have a sequencing column within the group, you need to calculate row numbers that will be used to identify the position of elements within the group. For example, if you need to concatenate all values from the string column per group, what do you specify as the pivoted attribute list (the spreading values)? The values in the *memberid* column are not known ahead of time, plus they differ in each group. Row numbers representing positions within the group solve this problem.

String Concatenation Using Pivoting

As the first example, the following query calculates an aggregate string concatenation over the column string for each group with a pivoting technique:

```
SELECT groupid,
    [1]
  + COALESCE(',' + [2], '')
  + COALESCE(',' + [3], '')
  + COALESCE(',' + [4], '') AS string
FROM (SELECT groupid, string,
        ROW_NUMBER() OVER(PARTITION BY groupid ORDER BY memberid) AS rn
      FROM dbo.Groups AS A) AS D
  PIVOT(MAX(string) FOR rn IN([1],[2],[3],[4])) AS P;
```

This query generates the following output:

```
groupid    string
---------- -------------------------
a          stra1,stra2
b          strb1,strb2,strb3,strb4
c          strc1,strc2,strc3
```

The query that generates the derived table D calculates a row number within the group based on *memberid* order. The outer query pivots the values based on the row numbers, and it performs linear concatenation. I'm assuming here that each group has at most four rows, so I specified four row numbers. You need as many row numbers as the maximum number of elements you anticipate.

The COALESCE function is used to replace a NULL representing a nonexistent element with an empty string so as not to cause the result to become NULL. You don't need the COALESCE function with the first element (*[1]*) because at least one element must exist in the group; otherwise, the group won't appear in the table.

Aggregate Product Using Pivoting

In a similar manner, you can calculate the product of the values in the *val* column for each group:

```
SELECT groupid,
    [1]
  * COALESCE([2], 1)
  * COALESCE([3], 1)
  * COALESCE([4], 1) AS product
FROM (SELECT groupid, val,
        ROW_NUMBER() OVER(PARTITION BY groupid ORDER BY memberid) AS rn
      FROM dbo.Groups AS A) AS D
  PIVOT(MAX(val) FOR rn IN([1],[2],[3],[4])) AS P;
```

This query generates the following output:

```
groupid    product
---------- -----------
a          42
b          693
c          960
```

The need for an aggregate product is common in financial applications—for example, to calculate compound interest rates.

User Defined Aggregates (UDA)

SQL Server allows you to create your own user-defined aggregates (UDAs). You write UDAs in a .NET language of your choice (for example, C# or Visual Basic), and you use them in T-SQL. This book is dedicated to T-SQL and not to the common language runtime (CLR), so I won't explain CLR UDAs at great length. Rather, I'll provide you with a couple of examples with step-by-step instructions and, of course, the T-SQL interfaces involved. Examples are provided in both C# and Visual Basic.

CLR Code in a Database

This section discusses .NET common language runtime (CLR) integration in SQL Server; therefore, it's appropriate to spend a couple of words explaining the reasoning behind CLR integration in a database. It is also important to identify the scenarios where using CLR objects is more appropriate than using T-SQL.

Developing in .NET languages such as C# and Visual Basic gives you an incredibly rich programming model. The .NET Framework includes literally thousands of prepared classes, and it is up to you to make astute use of them. .NET languages are not just data oriented like SQL, so you are not as limited. For example, regular expressions are extremely useful for validating data, and they are fully supported in .NET. SQL languages are set oriented and slow to perform row-oriented (row-by-row or one-row-at-a-time) operations. Sometimes you need row-oriented operations inside the database; moving away from cursors to CLR code should improve the performance. Another benefit of CLR code is that it can be much faster than T-SQL code for operations such as string manipulation and iterations and in computationally intensive calculations.

SQL Server 2005 introduced CLR integration, and SQL Server 2008 enhances this integration in a number of ways. Later in this section I'll describe the enhancements that are applicable to UDAs. Although SQL Server supported programmatic extensions even before CLR integration was introduced, CLR integration in .NET code is superior in a number of ways.

For example, you could add functionality to earlier versions of SQL Server (before 2005) using extended stored procedures. However, such procedures can compromise the integrity of SQL Server processes because their memory and thread management is not integrated well enough with SQL Server's resource management. .NET code is managed by the CLR inside SQL Server, and because the CLR itself is managed by SQL Server, it is much safer to use than extended procedure code.

T-SQL—a set-oriented language—was designed to deal mainly with data and is optimized for data manipulation. You should not rush to translate all your T-SQL code to CLR code. T-SQL is still SQL Server's primary language. Data access can be achieved through T-SQL only. If an operation can be expressed as a set-oriented one, you should program it in T-SQL.

You need to make another important decision before you start using CLR code inside SQL Server. You need to decide where your CLR code is going to run—at the server or at the client. CLR code is typically faster and more flexible than T-SQL for computations, and thus it extends the opportunities for server-side computations. However, the server side is typically a single working box, and load balancing at the data tier is still in its infancy. Therefore, you should consider whether it would be more sensible to process those computations at the client side.

With CLR code, you can write stored procedures, triggers, user-defined functions, user-defined types, and user-defined aggregate functions. The last two objects can't be written with declarative T-SQL; rather, they can be written only with CLR code. A user-defined type (UDT) is the most complex CLR object type and demands extensive coverage.

More Info For details about programming CLR UDTs, as well as programming CLR routines, please refer to *Inside Microsoft SQL Server 2008: T-SQL Programming*.

Let's start with a concrete implementation of two UDAs. The steps involved in creating a CLR-based UDA are as follows:

1. Define the UDA as a class in a .NET language.
2. Compile the class you defined to build a CLR assembly.
3. Register the assembly in SQL Server using the CREATE ASSEMBLY command in T-SQL.
4. Use the CREATE AGGREGATE command in T-SQL to create the UDA that references the registered assembly.

Note You can register an assembly and create a CLR object from Microsoft Visual Studio 2008 directly, using the project deployment option (from the Build menu item, choose the Deploy option). Direct deployment from Visual Studio is supported only with the Professional edition or higher; if you're using the Standard edition, your only option is explicit deployment in SQL Server.

This section will provide examples for creating aggregate string concatenation and aggregate product functions in both C# and Visual Basic. You can find the code for the C# classes in Listing 8-2 and the code for the Visual Basic classes in Listing 8-3. You'll be provided with the requirements for a CLR UDA alongside the development of a UDA.

LISTING 8-2 C# code for UDAs

```
using System;
using System.Data;
using Microsoft.SqlServer.Server;
using System.Data.SqlTypes;
using System.IO;
using System.Text;
using System.Runtime.InteropServices;

[Serializable]
[SqlUserDefinedAggregate(
   Format.UserDefined,              // use user defined serialization
   IsInvariantToNulls = true,       // NULLs don't matter
   IsInvariantToDuplicates = false, // duplicates matter
   IsInvariantToOrder = false,      // order matters
   IsNullIfEmpty = false,           // do not yield a NULL for a set of zero strings
   MaxByteSize = -1)                // max size unlimited
]
public struct StringConcat : IBinarySerialize
{
  private StringBuilder sb;

  public void Init()
  {
    this.sb = new StringBuilder();
  }

  //two arguments
  public void Accumulate(SqlString v, SqlString separator)
  {
    if (v.IsNull)
    {
      return; // ignore NULLs approach
    }

    this.sb.Append(v.Value).Append(separator.Value);
  }

  public void Merge(StringConcat other)
  {
    this.sb.Append(other.sb);
  }

  public SqlString Terminate()
  {
    string output = string.Empty;
    if (this.sb != null && this.sb.Length > 0)
    {
      // remove last separator
      output = this.sb.ToString(0, this.sb.Length - 1);
    }

    return new SqlString(output);
  }
```

```
  public void Read(BinaryReader r)
  {
    sb = new StringBuilder(r.ReadString());
  }

  public void Write(BinaryWriter w)
  {
    w.Write(this.sb.ToString());
  }
} // end StringConcat

[Serializable]
[StructLayout(LayoutKind.Sequential)]
[SqlUserDefinedAggregate(
   Format.Native,                   // use native serialization
   IsInvariantToNulls = true,       // NULLs don't matter
   IsInvariantToDuplicates = false, // duplicates matter
   IsInvariantToOrder = false)]     // order matters
public class Product
{
  private SqlInt64 si;

  public void Init()
  {
    si = 1;
  }

  public void Accumulate(SqlInt64 v)
  {
    if (v.IsNull || si.IsNull)  // NULL input = NULL output approach
    {
      si = SqlInt64.Null;
      return;
    }
    if (v == 0 || si == 0)      // to prevent an exception in next if
    {
      si = 0;
      return;
    }
    // stop before we reach max v
    if (Math.Abs(v.Value) <= SqlInt64.MaxValue / Math.Abs(si.Value))
    {
      si = si * v;
    }
    else
    {
      si = 0;                 // if we reach too big v, return 0
    }

  }

  public void Merge(Product Group)
  {
    Accumulate(Group.Terminate());
  }
```

```
  public SqlInt64 Terminate()
  {
    return (si);
  }

} // end Product
```

LISTING 8-3 Visual Basic code for UDAs

```
Imports System
Imports System.Data
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server
Imports System.Text
Imports System.IO
Imports System.Runtime.InteropServices

<Serializable(), _
 SqlUserDefinedAggregate( _
               Format.UserDefined, _
               IsInvariantToDuplicates:=False, _
               IsInvariantToNulls:=True, _
               IsInvariantToOrder:=False, _
               IsNullIfEmpty:=False, _
               MaxByteSize:=-1)> _
Public Structure StringConcat
  Implements IBinarySerialize

  Private sb As StringBuilder

  Public Sub Init()
    Me.sb = New StringBuilder()
  End Sub

  Public Sub Accumulate(ByVal v As SqlString, ByVal separator As SqlString)
    If v.IsNull Then
      Return
    End If

    Me.sb.Append(v.Value).Append(separator.Value)
  End Sub

  Public Sub Merge(ByVal other As StringConcat)
    Me.sb.Append(other.sb)
  End Sub

  Public Function Terminate() As SqlString
    Dim output As String = String.Empty

    If Not (Me.sb Is Nothing) AndAlso Me.sb.Length > 0 Then
      output = Me.sb.ToString(0, Me.sb.Length - 1)
    End If

    Return New SqlString(output)
  End Function
```

```
  Public Sub Read(ByVal r As BinaryReader) _
    Implements IBinarySerialize.Read
    sb = New StringBuilder(r.ReadString())
  End Sub

  Public Sub Write(ByVal w As BinaryWriter) _
    Implements IBinarySerialize.Write
    w.Write(Me.sb.ToString())
  End Sub

End Structure

<Serializable(), _
 StructLayout(LayoutKind.Sequential), _
 SqlUserDefinedAggregate( _
               Format.Native, _
               IsInvariantToOrder:=False, _
               IsInvariantToNulls:=True, _
               IsInvariantToDuplicates:=False)> _
Public Class Product

  Private si As SqlInt64

  Public Sub Init()
    si = 1
  End Sub

  Public Sub Accumulate(ByVal v As SqlInt64)
    If v.IsNull = True Or si.IsNull = True Then
      si = SqlInt64.Null
      Return
    End If
    If v = 0 Or si = 0 Then
      si = 0
      Return
    End If
    If (Math.Abs(v.Value) <= SqlInt64.MaxValue / Math.Abs(si.Value)) _
      Then
      si = si * v
    Else
      si = 0
    End If
  End Sub

  Public Sub Merge(ByVal Group As Product)
    Accumulate(Group.Terminate())
  End Sub

  Public Function Terminate() As SqlInt64
    If si.IsNull = True Then
      Return SqlInt64.Null
    Else
      Return si
    End If
  End Function

End Class
```

Use the following step-by-step instructions to create and deploy the assemblies in Visual Studio 2008.

Creating and Deploying an Assembly in Visual Studio 2008

1. In Visual Studio 2008, create a new C# or Visual Basic project based on your language preference. Use the Database folder and the SQL Server Project template.

> **Note** This template is not available in Visual Studio 2008, Standard edition. If you're working with the Standard edition, use the Class Library template and manually write all the code.

2. In the New Project dialog box, specify the following information:
   - Name UDAs
   - Location C:\
   - Solution Name UDAs

   When you're done entering the information, confirm that it is correct.
3. At this point, you'll be requested to specify a database reference. Create a new database reference to the tempdb database in the SQL Server instance you're working with and choose it. The database reference you choose tells Visual Studio where to deploy the UDAs that you develop.
4. After confirming the choice of database reference, in the Solution Explorer window, right-click the UDAs project, select the menu items Add and Aggregate, and then choose the Aggregate template. If you're using C#, rename the class Aggregate1.cs to **UDAClasses.cs**. If you're using Visual Basic, rename Aggregate1.vb to **UDAClasses.vb**. Confirm.
5. Examine the code of the template. You'll find that a UDA is implemented as a structure (*struct* in C#, *Structure* in Visual Basic). It can be implemented as a class as well. The first block of code in the template includes namespaces that are used in the assembly (lines of code starting with *using* in C# and with *Imports* in Visual Basic). Add three more statements to include the following namespaces: *System.Text*, *System.IO*, and *System.Runtime.InteropServices*. (You can copy those from Listing 8-2 or Listing 8-3.) You'll use the *StringBuilder* class from the *System.Text* namespace, the *BinaryReader* and *BinaryWriter* classes from the *System.IO* namespace, and the *StructLayout* attribute from the *System.Runtime.InteropServices* namespace (in the second UDA).
6. Rename the default name of the UDA—which is currently the same name as the name of the class (*UDAClasses*)—to **StringConcat**.
7. You'll find four methods that are already provided by the template. These are the methods that every UDA must implement. However, if you use the Class Library template for your

project, you have to write them manually. Using the Aggregate template, all you have to do is fill them with your code. Following is a description of the four methods:

- *Init* This method is used to initialize the computation. It is invoked once for each group that the query processor is aggregating.
- *Accumulate* The name of the method gives you a hint at its purpose—accumulating the aggregate values, of course. This method is invoked once for each value (that is, for every single row) in the group that is being aggregated. It uses input parameters, and the parameters have to be of the data types corresponding to the native SQL Server data types of the columns you are going to aggregate. The data type of the input can also be a CLR UDT. In SQL Server 2005, UDAs supported no more than one input parameter. In SQL Server 2008, UDAs support multiple input parameters.
- *Merge* Notice that this method uses an input parameter with the type that is the aggregate class. The method is used to merge multiple partial computations of an aggregation.
- *Terminate* This method finishes the aggregation and returns the result.

8. Add an internal (private) variable—*sb*—to the class just before the *Init* method. You can do so by simply copying the code that declares it from Listing 8-2 or Listing 8-3, depending on your choice of language. The variable *sb* is of type *StringBuilder* and will hold the intermediate aggregate value.
9. Override the current code for the four methods with the code implementing them from Listing 8-2 or Listing 8-3. Keep in mind the following points for each method:
   - In the *Init* method, you initialize *sb* with an empty string.
   - The *Accumulate* method accepts two input parameters (new in SQL Server 2008)—*v* and *separator*. The parameter *v* represents the value to be concatenated, and the parameter *separator* is obviously the separator. If *v* is NULL, it is simply ignored, similar to the way built-in aggregates handle NULLs. If *v* is not NULL, the value in *v* and a separator are appended to *sb*.
   - In the *Merge* method, you are simply adding a partial aggregation to the current one. You do so by calling the *Accumulate* method of the current aggregation and adding the termination (final value) of the other partial aggregation. The input of the Merge function refers to the class name, which you revised earlier to *StringConcat*.
   - The *Terminate* method is very simple as well; it just returns the string representation of the aggregated value minus the superfluous separator at the end.
10. Delete the last two rows of the code in the class from the template; these are a placeholder for a member field. You already defined the member field you need at the beginning of the UDA.

11. Next, go back to the top of the UDA, right after the inclusion of the namespaces. You'll find attribute names that you want to include. Attributes help Visual Studio in deployment, and they help SQL Server to optimize the usage of the UDA. UDAs have to include the *Serializable* attribute. Serialization in .NET means saving the values of the fields of a class persistently. UDAs need serialization for intermediate results. The format of the serialization can be native, meaning they are left to SQL Server or defined by the user. Serialization can be native if you use only .NET value types; it has to be user defined if you use .NET reference types. Unfortunately, the *string* type is a reference type in .NET. Therefore, you have to prepare your own serialization. You have to implement the *IBinarySerialize* interface, which defines just two methods: *Read* and *Write*. The implementation of these methods in our UDA is very simple. The *Read* method uses the *ReadString* method of the *StringBuilder* class. The *Write* method uses the default *ToString* method. The *ToString* method is inherited by all .NET classes from the topmost class, called *System.Object*.

Continue implementing the UDA by following these steps:

11.1. Specify that you are going to implement the *IBinarySerialize* interface in the structure. If you're using C#, you do so by adding a colon and the name of the interface right after the name of the structure (the UDA name). If you're using Visual Basic, you do so by adding *Implements IBinarySerialize* after the name of the structure.

11.2. Copy the *Read* and *Write* methods from Listing 8-2 or Listing 8-3 to the end of your UDA.

11.3. Change the *Format.Native* property of the *SqlUserDefinedAggregate* attribute to **Format.UserDefined**. In SQL Server 2005, with user-defined serialization, your aggregate was limited to 8,000 bytes only. You had to specify how many bytes your UDA could return at maximum with the *MaxByteSize* property of the *SqlUserDefinedAggregate* attribute. SQL Server 2008 lifts this restriction and supports unlimited size (or more accurately, the maximum size supported by large object types like VARCHAR(MAX), which is currently 2 GB). A value of –1 in the *MaxByteSize* property indicates unlimited size.

12. You'll find some other interesting properties of the *SqlUserDefinedAggregate* attribute in Listings 8-2 and 8-3. Let's explore them:

- *IsInvariantToDuplicates* This is an optional property. For example, the *MAX* aggregate is invariant to duplicates, while *SUM* is not.
- *IsInvariantToNulls* This is another optional property. It specifies whether the aggregate is invariant to NULLs.
- *IsInvariantToOrder* This property is reserved for future use. It is currently ignored by the query processor. Therefore, order is currently not guaranteed. If you want to concatenate elements in a certain order, you have to implement your own sorting logic either in the *Accumulate* or the *Terminate* methods. This naturally incurs extra cost and unfortunately cannot benefit from index ordering.

- *IsNullIfEmpty* This property indicates whether the aggregate returns a NULL if no values have been accumulated.

13. Add the aforementioned properties to your UDA by copying them from Listing 8-2 or Listing 8-3. Your first UDA is now complete!
14. Listings 8-2 and 8-3 also have the code to implement a product UDA (*Product*). Copy the complete code implementing *Product* to your script. Note that this UDA involves handling of big integers only. Because the UDA internally deals only with value types, it can use native serialization. Native serialization requires that the *StructLayoutAttribute* be specified as *StructLayout.LayoutKind.Sequential* if the UDA is defined in a class and not a structure. Otherwise, the UDA implements the same four methods as your previous UDA. An additional check in the *Accumulate* method prevents out-of-range values.
15. Save all files by choosing the File menu item and then choosing Save All.
16. Create the assembly file in the project folder by building the solution. You do this by choosing the Build menu item and then choosing Build Solution.
17. Deploy the assembly in SQL Server.

Note To automatically deploy the solution in SQL Server, you normally choose the Build menu item and then choose Deploy Solution. However, at the time of this writing, automatic deployment in Visual Studio 2008 with Service Pack 1 fails if you use any of the new UDA features in SQL Server 2008 (multiple input parameters or the unlimited maximum size). Therefore, I'll provide instructions here to do explicit deployment.

18. Explicit deployment of the UDAs in SQL Server involves running the CREATE ASSEMBLY command to import the intermediate language code from the assembly file into the target database (tempdb in our case) and the CREATE AGGREGATE command to register each aggregate. If you used C# to define the UDAs, run the following code while connected to the tempdb database:

```
CREATE ASSEMBLY UDAs
  FROM 'C:\UDAs\UDAs\bin\Debug\UDAs.dll';

CREATE AGGREGATE dbo.StringConcat
(
  @value     AS NVARCHAR(MAX),
  @separator AS NCHAR(1)
)
RETURNS NVARCHAR(MAX)
EXTERNAL NAME UDAs.StringConcat;

CREATE AGGREGATE dbo.Product
(
  @value     AS BIGINT
)
RETURNS BIGINT
EXTERNAL NAME UDAs.Product;
```

If you used Visual Basic, run the following code:

```
CREATE ASSEMBLY UDAs
  FROM 'C:\UDAs\UDAs\bin\UDAs.dll';

CREATE AGGREGATE dbo.StringConcat
(
  @value     AS NVARCHAR(MAX),
  @separator AS NCHAR(1)
)
RETURNS NVARCHAR(MAX)
EXTERNAL NAME UDAs.[UDAs.StringConcat];

CREATE AGGREGATE dbo.Product
(
  @value     AS BIGINT
)
RETURNS BIGINT
EXTERNAL NAME UDAs.[UDAs.Product];
```

The assembly should be cataloged at this point, and both UDAs should be created.

You can check whether the deployment was successful by browsing the *sys.assemblies* and *sys.assembly_modules* catalog views, which are in the tempdb database in our case. Run the following code to query those views:

```
SELECT * FROM sys.assemblies;
SELECT * FROM sys.assembly_modules;
```

Note that to run user-defined assemblies in SQL Server, you need to enable the server configuration option *'clr enabled'* (which is disabled by default). You do so by running the following code:

```
EXEC sp_configure 'clr enabled', 1;
RECONFIGURE WITH OVERRIDE;
```

This requirement is applicable only if you want to run user-defined assemblies; this option is not required to be turned on if you want to run system-supplied assemblies.

That's basically it. You use UDAs just like you use any built-in aggregate function—and that's one of their great advantages compared to other solutions to custom aggregates. To test the new functions, run the following code, and you'll get the same results returned by the other solutions to custom aggregates I presented earlier:

```
SELECT groupid, dbo.StringConcat(string, N',') AS string
FROM dbo.Groups
GROUP BY groupid;

SELECT groupid, dbo.Product(val) AS product
FROM dbo.Groups
GROUP BY groupid;
```

Note that the *StringConcat* function expects a non-NULL separator as input and will fail if provided with a NULL. Of course, you can add logic to the function's definition to use some default separator when a NULL is specified.

Specialized Solutions

Another type of solution for custom aggregates is developing a specialized, optimized solution for each aggregate. The advantage is usually the improved performance of the solution. The disadvantage is that you probably won't be able to use similar logic for other aggregate calculations.

Specialized Solution for Aggregate String Concatenation

A specialized solution for aggregate string concatenation uses the PATH mode of the FOR XML query option. This beautiful (and extremely fast) technique was devised by Michael Rys, a program manager with the Microsoft SQL Server development team, and Eugene Kogan, a technical lead on the Microsoft SQL Server Engine team. The PATH mode provides an easier way to mix elements and attributes than the EXPLICIT directive. Here's the specialized solution for aggregate string concatenation:

```
SELECT groupid,
  STUFF((SELECT ',' + string AS [text()]
         FROM dbo.Groups AS G2
         WHERE G2.groupid = G1.groupid
         ORDER BY memberid
         FOR XML PATH('')), 1, 1, '') AS string
FROM dbo.Groups AS G1
GROUP BY groupid;
```

The subquery basically returns an ordered path of all strings within the current group. Because an empty string is provided to the PATH clause as input, a wrapper element is not generated. An expression with no alias (for example, *',' + string*) or one aliased as *[text()]* is inlined, and its contents are inserted as a text node. The purpose of the STUFF function is simply to remove the first comma (by substituting it with an empty string).

Dynamic Pivoting Now that you are familiar with a fast, specialized solution to string concatenation, you can put it to use to achieve dynamic pivoting. Recall from the "Pivoting" section that the static solutions for pivoting in SQL Server require you to explicitly list the spreading values (the values in the spreading element). Consider the following static query, which I covered earlier in the "Pivoting" section:

```
SELECT *
FROM (SELECT custid, YEAR(orderdate) AS orderyear, qty
      FROM dbo.Orders) AS D
  PIVOT(SUM(qty) FOR orderyear IN([2006],[2007],[2008])) AS P;
```

Note that this query is against the dbo.Orders table that you created and populated earlier by running the code in Listing 8-1. Here you have to explicitly list the order years in the IN clause. If you want to make this solution more dynamic, query the distinct order years from the table and use the FOR XML PATH technique to construct the comma-separated list of years. You can use the QUOTENAME function to convert the integer years to Unicode character strings and add brackets around them. Also, using QUOTENAME is critical to prevent SQL Injection if this technique is used for a nonnumeric spreading column. The query that produces the comma-separated list of years looks like this:

```
SELECT
  STUFF(
    (SELECT N',' + QUOTENAME(orderyear) AS [text()]
     FROM (SELECT DISTINCT YEAR(orderdate) AS orderyear
           FROM dbo.Orders) AS Years
     ORDER BY orderyear
     FOR XML PATH('')), 1, 1, '');
```

Note that this useful technique has some limitations, though not serious ones, because it's XML based. For example, characters that have special meaning in XML, like '<', will be converted to codes (like <), yielding the wrong pivot statement.

What's left is to construct the whole query string, store it in a variable and use the *sp_executesql* stored procedure to execute it dynamically, like so:

```
DECLARE @sql AS NVARCHAR(1000);

SET @sql = N'SELECT *
FROM (SELECT custid, YEAR(orderdate) AS orderyear, qty
      FROM dbo.Orders) AS D
  PIVOT(SUM(qty) FOR orderyear IN(' +

STUFF(
  (SELECT N',' + QUOTENAME(orderyear) AS [text()]
   FROM (SELECT DISTINCT YEAR(orderdate) AS orderyear
         FROM dbo.Orders) AS Years
   ORDER BY orderyear
   FOR XML PATH('')), 1, 1, '') + N')) AS P;';

EXEC sp_executesql @stmt = @sql;
```

Specialized Solution for Aggregate Product

Keep in mind that to calculate an aggregate product, you have to scan all values in the group. So the performance potential your solution can reach is to achieve the calculation by scanning the data only once, using a set-based query. In the case of an aggregate product, this can be achieved using mathematical manipulation based on logarithms. I'll rely on the following logarithmic equations:

Equation 1: $log_a(b) = x$ *if and only if* $a^x = b$

Equation 2: $log_a(v1 * v2 * \ldots * vn) = log_a(v1) + log_a(v2) + \ldots + log_a(vn)$

Basically, what you're going to do here is a transformation of calculations. You have support in T-SQL for the LOG, POWER, and SUM functions. Using those, you can generate the missing product. Group the data by the *groupid* column, as you would with any built-in aggregate. The expression *SUM(LOG10(val))* corresponds to the right side of Equation 2, where the base *a* is equal to 10 in our case, because you used the LOG10 function. To get the product of the elements, all you have left to do is raise the base (10) to the power of the right side of the equation. In other words, the expression *POWER(10., SUM(LOG10(val)))* gives you the product of elements within the group. Here's what the full query looks like:

```
SELECT groupid, POWER(10., SUM(LOG10(val))) AS product
FROM dbo.Groups
GROUP BY groupid;
```

This is the final solution if you're dealing only with positive values. However, the logarithm function is undefined for zero and negative numbers. You can use pivoting techniques to identify and deal with zeros and negatives as follows:

```
SELECT groupid,
  CASE
    WHEN MAX(CASE WHEN val = 0 THEN 1 END) = 1 THEN 0
    ELSE
      CASE WHEN COUNT(CASE WHEN val < 0 THEN 1 END) % 2 = 0
        THEN 1 ELSE -1
      END * POWER(10., SUM(LOG10(NULLIF(ABS(val), 0))))
  END AS product
FROM dbo.Groups
GROUP BY groupid;
```

The outer CASE expression first uses a pivoting technique to check whether a 0 value appears in the group, in which case it returns a 0 as the result. The ELSE clause invokes another CASE expression, which also uses a pivoting technique to count the number of negative values in the group. If that number is even, it produces a +1; if it's odd, it produces a –1. The purpose of this calculation is to determine the numerical sign of the result. The sign (–1 or +1) is then multiplied by the product of the absolute values of the numbers in the group to give the desired product.

Note that NULLIF is used here to substitute zeros with NULLs. You might expect this part of the expression not to be evaluated at all if a zero is found. But remember that the optimizer can consider many different physical plans to execute your query. As a result, you can't be certain of the actual order in which parts of an expression will be evaluated. By substituting zeros with NULLs, you ensure that you'll never get a domain error if the LOG10 function ends up being invoked with a zero as an input. This use of NULLIF, together with the use of ABS, allows this solution to accommodate inputs of any sign (negative, zero, and positive).

You could also use a pure mathematical approach to handle zeros and negative values using the following query:

```
SELECT groupid,
  CAST(ROUND(EXP(SUM(LOG(ABS(NULLIF(val,0))))*
    (1-SUM(1-SIGN(val))%4)*(1-SUM(1-SQUARE(SIGN(val)))),0) AS INT)
 AS product
FROM dbo.Groups
GROUP BY groupid;
```

This example shows that you should never lose hope when searching for an efficient solution. If you invest the time and think outside the box, in most cases you'll find a solution.

Specialized Solutions for Aggregate Bitwise Operations

In this section, I'll introduce specialized solutions for aggregating the T-SQL bitwise operations—bitwise OR (|), bitwise AND (&), and bitwise XOR (^). I'll assume that you're familiar with the basics of bitwise operators and their uses and provide only a brief overview. If you're not, please refer first to the section "Bitwise Operators" in SQL Server Books Online.

Bitwise operations are operations performed on the individual bits of integer data. Each bit has two possible values, 1 and 0. Integers can be used to store *bitmaps,* or strings of bits, and in fact they are used internally by SQL Server to store metadata information—for example, properties of indexes (clustered, unique, and so on) and properties of databases (readonly, restrict access, autoshrink, and so on). You might also choose to store bitmaps yourself to represent sets of binary attributes—for example, a set of permissions where each bit represents a different permission.

Some experts advise against using such a design because it violates 1NF (first normal form, which requires attributes to be atomic). You might well prefer to design your data in a more normalized form, where attributes like this are stored in separate columns. I don't want to get into a debate about which design is better. Here I'll assume a given design that does store bitmaps with sets of flags, and I'll assume that you need to perform aggregate bitwise activities on these bitmaps. I just want to introduce the techniques for cases where you do find the need to use them.

Bitwise OR (|) is usually used to construct bitmaps or to generate a result bitmap that accumulates all bits that are turned on. In the result of bitwise OR, bits are turned on (that is, have value 1) if they are turned on in at least one of the separate bitmaps.

Bitwise AND (&) is usually used to check whether a certain bit (or a set of bits) is turned on by ANDing the source bitmap and a mask. It's also used to accumulate only bits that are turned on in all bitmaps. It generates a result bit that is turned on if that bit is turned on in all the individual bitmaps.

Bitwise XOR (^) is usually used to calculate parity or as part of a scheme to encrypt data. For each bit position, the result bit is turned on if it is on in an odd number of the individual bitmaps.

Note Bitwise XOR is the only bitwise operator that is reversible. That's why it's used for parity calculations and encryption.

Aggregate versions of the bitwise operators are not provided in SQL Server, and I'll provide solutions here to perform aggregate bitwise operations. I'll use the same Groups table that I used in my other custom aggregate examples. Assume that the integer column *val* represents a bitmap. To see the bit representation of each integer, first create the function *DecToBase* by running the following code:

```
IF OBJECT_ID('dbo.DecToBase') IS NOT NULL
  DROP FUNCTION dbo.DecToBase;
GO
CREATE FUNCTION dbo.DecToBase(@val AS BIGINT, @base AS INT)
  RETURNS VARCHAR(63)
AS
BEGIN
  DECLARE @r AS VARCHAR(63), @alldigits AS VARCHAR(36);

  SET @alldigits = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';

  SET @r = '';
  WHILE @val > 0
  BEGIN
    SET @r = SUBSTRING(@alldigits, @val % @base + 1, 1) + @r;
    SET @val = @val / @base;
  END

  RETURN @r;
END
GO
```

The function accepts two inputs: a 64-bit integer holding the source bitmap and a base in which you want to represent the data. Use the following query to return the bit representation of the integers in the *val* column of Groups:

```
SELECT groupid, val,
  RIGHT(REPLICATE('0', 32) + CAST(dbo.DecToBase(val, 2) AS VARCHAR(64)),
        32) AS binval
FROM dbo.Groups;
```

This code generates the following output (only the 10 rightmost digits of *binval* are shown):

```
groupid    val         binval
---------- ----------- ---------
a          6           00000110
a          7           00000111
b          3           00000011
b          7           00000111
b          3           00000011
b          11          00001011
c          8           00001000
c          10          00001010
c          12          00001100
```

The *binval* column shows the *val* column in base 2 representation, with leading zeros to create a string with a fixed number of digits. Of course, you can adjust the number of leading zeros according to your needs, which I did to produce the outputs I'll show. To avoid distracting you from the techniques I want to focus on, however, the code for that adjustment is not in my code samples.

Aggregate Bitwise OR Without further ado, let's start with calculating an aggregate bitwise OR. To give tangible context to the problem, imagine that you're maintaining application security in the database. The *groupid* column represents a user, and the *val* column represents a bitmap with permission states (either 1 for granted or 0 for not granted) of a role the user is a member of. You're after the effective permissions bitmap for each user (group), which should be calculated as the aggregate bitwise OR between all bitmaps of roles the user is a member of.

The main aspect of a bitwise OR operation that I'll rely on in my solutions is the fact that it's equivalent to the arithmetic sum of the values represented by each distinct bit value that is turned on in the individual bitmaps. Within an integer, a bit represents the value $2^{(bit\_pos-1)}$. For example, the bit value of the third bit is $2^2 = 4$. Take, for example, the bitmaps for user c: 8 (1000), 10 (1010), and 12 (1100). The bitmap 8 has only one bit turned on—the bit value representing 8; 10 has the bits representing 8 and 2 turned on; and 12 has the 8 and 4 bits turned on. The distinct bits turned on in any of the integers 8, 10, and 12 are the 2, 4, and 8 bits, so the aggregate bitwise OR of 8, 10, and 12 is equal to *2 + 4 + 8 = 14 (1110)*.

The following solution relies on the aforementioned logic by extracting the individual bit values that are turned on in any of the participating bitmaps. The extraction is achieved using the expression *MAX(val & <bitval>)*. The query then performs an arithmetic sum of the individual bit values:

```
SELECT groupid,
    MAX(val & 1)
  + MAX(val & 2)
  + MAX(val & 4)
  + MAX(val & 8)
-- ...
  + MAX(val & 1073741824) AS agg_or
FROM dbo.Groups
GROUP BY groupid;
```

This query generates the following output:

```
groupid    agg_or      binval
---------- ----------- --------
a          7           00000111
b          15          00001111
c          14          00001110
```

Note that I added a third column (*binval*) to the output showing the 10 rightmost digits of the binary representation of the result value. I'll continue to do so with the rest of the queries that apply aggregate bitwise operations.

Similarly, you can use *SUM(DISTINCT val & <bitval>)* instead of *MAX(val & <bitval>)* because the only possible results are *<bitval>* and *0*:

```
SELECT groupid,
    SUM(DISTINCT val & 1)
  + SUM(DISTINCT val & 2)
  + SUM(DISTINCT val & 4)
  + SUM(DISTINCT val & 8)
-- ...
  + SUM(DISTINCT val & 1073741824) AS agg_or
FROM dbo.Groups
GROUP BY groupid;
```

Both solutions suffer from the same limitation—lengthy query strings—because of the need for a different expression for each bit value. In an effort to shorten the query strings, you can use an auxiliary table. You join the Groups table with an auxiliary table that contains all relevant bit values, using *val & bitval = bitval* as the join condition. The result of the join will include all bit values that are turned on in any of the bitmaps. You can then find *SUM(DISTINCT <bitval>)* for each group. You can easily generate the auxiliary table of bit values from the Nums table used earlier. Filter as many numbers as the bits that you might need and raise 2 to the power *n–1*. Here's the complete solution:

```
SELECT groupid, SUM(DISTINCT bitval) AS agg_or
FROM dbo.Groups
  JOIN (SELECT POWER(2, n-1) AS bitval
        FROM dbo.Nums
        WHERE n <= 31) AS Bits
    ON val & bitval = bitval
GROUP BY groupid;
```

Aggregate Bitwise AND In a similar manner, you can calculate an aggregate bitwise AND. In the permissions scenario, an aggregate bitwise AND represents the most restrictive permission set. Just keep in mind that a bit value should be added to the arithmetic sum only if it's turned on in all bitmaps. So first group the data by *groupid* and *bitval* and filter only the groups where *MIN(val & bitval) > 0*, meaning that the bit value was turned on in all bitmaps. In an outer query, group the data by *groupid* and perform the arithmetic sum of the bit values from the inner query:

```
SELECT groupid, SUM(bitval) AS agg_and
FROM (SELECT groupid, bitval
      FROM dbo.Groups,
        (SELECT POWER(2, n-1) AS bitval
         FROM dbo.Nums
         WHERE n <= 31) AS Bits
      GROUP BY groupid, bitval
      HAVING MIN(val & bitval) > 0) AS D
GROUP BY groupid;
```

This query generates the following output:

```
groupid    agg_and     binval
---------- ----------- --------
a          6           00000110
b          3           00000011
c          8           00001000
```

Aggregate Bitwise XOR To calculate an aggregate bitwise XOR operation, filter only the *groupid, bitval* groups that have an odd number of bits turned on, as shown in the following code, which illustrates an aggregate bitwise XOR using Nums:

```
SELECT groupid, SUM(bitval) AS agg_xor
FROM (SELECT groupid, bitval
      FROM dbo.Groups,
        (SELECT POWER(2, n-1) AS bitval
         FROM dbo.Nums
         WHERE n <= 31) AS Bits
      GROUP BY groupid, bitval
      HAVING SUM(SIGN(val & bitval)) % 2 = 1) AS D
GROUP BY groupid;
```

This query produces the following output:

```
groupid    agg_xor     binval
---------- ----------- --------
a          1           00000001
b          12          00001100
c          14          00001110
```

Median

As another example of a specialized custom aggregate solution, I'll use the statistical median calculation. Suppose that you need to calculate the median of the *val* column for each group. There are two different definitions of median. Here we will return the middle value in case we have an odd number of elements and the average of the two middle values in case we have an even number of elements.

The following code shows a technique for calculating the median:

```
WITH Tiles AS
(
  SELECT groupid, val,
    NTILE(2) OVER(PARTITION BY groupid ORDER BY val) AS tile
  FROM dbo.Groups
),
GroupedTiles AS
(
  SELECT groupid, tile, COUNT(*) AS cnt,
    CASE WHEN tile = 1 THEN MAX(val) ELSE MIN(val) END AS val
  FROM Tiles
  GROUP BY groupid, tile
)
```

```
SELECT groupid,
  CASE WHEN MIN(cnt) = MAX(cnt) THEN AVG(1.*val)
       ELSE MIN(val) END AS median
FROM GroupedTiles
GROUP BY groupid;
```

This code generates the following output:

```
groupid    median
---------- ----------
a          6.500000
b          5.000000
c          10.000000
```

The Tiles CTE calculates the *NTILE(2)* value within the group, based on *val* order. When you have an even number of elements, the first half of the values gets tile number 1, and the second half gets tile number 2. In an even case, the median is supposed to be the average of the highest value within the first tile and the lowest in the second. When you have an odd number of elements, remember that an additional row is added to the first group. This means that the highest value in the first tile is the median.

The second CTE (GroupedTiles) groups the data by group and tile number, returning the row count for each group and tile as well as the *val* column, which for the first tile is the maximum value within the tile and for the second tile is the minimum value within the tile.

The outer query groups the two rows in each group (one representing each tile). A CASE expression in the SELECT list determines what to return based on the parity of the group's row count. When the group has an even number of rows (that is, the group's two tiles have the same row count), you get the average of the maximum in the first tile and the minimum in the second. When the group has an odd number of elements (that is, the group's two tiles have different row counts), you get the minimum of the two values, which happens to be the maximum within the first tile, which, in turn, happens to be the median.

Using the ROW_NUMBER function, you can come up with additional solutions to finding the median that are more elegant and somewhat simpler. Here's the first example:

```
WITH RN AS
(
  SELECT groupid, val,
    ROW_NUMBER()
      OVER(PARTITION BY groupid ORDER BY val, memberid) AS rna,
    ROW_NUMBER()
      OVER(PARTITION BY groupid ORDER BY val DESC, memberid DESC) AS rnd
  FROM dbo.Groups
)
SELECT groupid, AVG(1.*val) AS median
FROM RN
WHERE ABS(rna - rnd) <= 1
GROUP BY groupid;
```

The idea is to calculate two row numbers for each row: one based on *val*, *memberid* (the tiebreaker) in ascending order (*rna*) and the other based on the same attributes in descending order (*rnd*). Two sequences sorted in opposite directions have an interesting mathematical relationship that you can use to your advantage. The absolute difference between the two is smaller than or equal to 1 only for the elements that need to participate in the median calculation. Take, for example, a group with an odd number of elements; *ABS(rna – rnd)* is equal to 0 only for the middle row. For all other rows, it is greater than 1. Given an even number of elements, the difference is 1 for the two middle rows and greater than 1 for all others.

The reason for using *memberid* as a tiebreaker is to guarantee determinism of the row number calculations. Because you're calculating two different row numbers, you want to make sure that a value that appears at the *n*th position from the beginning in ascending order appears at the *n*th position from the end in descending order.

Once the values that need to participate in the median calculation are isolated, you just need to group them by *groupid* and calculate the average per group.

You can avoid the need to calculate two separate row numbers by deriving the second from the first. The descending row numbers can be calculated by subtracting the ascending row numbers from the count of rows in the group and adding one. For example, in a group of four elements, the row that got an ascending row number 1 would get the descending row number *4–1+1 = 4*. Ascending row number 2 would get the descending row number *4–2+1 = 3* and so on. Deriving the descending row number from the ascending one eliminates the need for a tiebreaker. You're not dealing with two separate calculations; therefore, nondeterminism is not an issue anymore.

So the calculation *rna – rnd* becomes the following: *rn – (cnt-rn+1) = 2*rn – cnt – 1*. Here's a query that implements this logic:

```
WITH RN AS
(
  SELECT groupid, val,
    ROW_NUMBER() OVER(PARTITION BY groupid ORDER BY val) AS rn,
    COUNT(*) OVER(PARTITION BY groupid) AS cnt
  FROM dbo.Groups
)
SELECT groupid, AVG(1.*val) AS median
FROM RN
WHERE ABS(2*rn - cnt - 1) <= 1
GROUP BY groupid;
```

Here's another way to figure out which rows participate in the median calculation based on the row number and the count of rows in the group: *rn IN((cnt+1)/2, (cnt+2)/2)*. For an odd number of elements, both expressions yield the middle row number. For example, if you have 7 rows, both *(7+1)/2* and *(7+2)/2* equal 4. For an even number of elements, the first expression yields the row number just before the middle point, and the second yields the

row number just after it. If you have 8 rows, *(8+1)/2* yields 4, and *(8+2)/2* yields 5. Here's the query that implements this logic:

```
WITH RN AS
(
  SELECT groupid, val,
    ROW_NUMBER() OVER(PARTITION BY groupid ORDER BY val) AS rn,
    COUNT(*) OVER(PARTITION BY groupid) AS cnt
  FROM dbo.Groups
)
SELECT groupid, AVG(1.*val) AS median
FROM RN
WHERE rn IN((cnt+1)/2, (cnt+2)/2)
GROUP BY groupid;
```

Mode

The last specialized solution of a custom aggregate that I'll cover is for the mode of a distribution. The mode is the most frequently occurring value. As an example of mode calculation, consider a request to return for each customer the ID of the employee who handled the most orders for that customer, according to the Sales.Orders table in the InsideTSQL2008 database. In case of ties, you need to determine what you want to do. One option is to return all tied employees; another option is to use a tiebreaker to determine which to return—for example, the one with the higher employee ID.

The first solution that I'll present is based on ranking calculations. I'll first describe a solution that applies a tiebreaker, and then I'll explain the required revisions for the solution to return all ties.

You group the rows by customer ID and employee ID. You calculate a count of orders per group, plus a row number partitioned by customer ID, based on the order of count descending and employee ID descending. The rows with the employee ID that is the mode—with the higher employee ID used as a tiebreaker—have row number 1. What's left is to define a table expression based on the query and in the outer query filter only the rows where the row number is equal to 1, like so:

```
USE InsideTSQL2008;

WITH C AS
(
  SELECT custid, empid, COUNT(*) AS cnt,
    ROW_NUMBER() OVER(PARTITION BY custid
                      ORDER BY COUNT(*) DESC, empid DESC) AS rn
  FROM Sales.Orders
  GROUP BY custid, empid
)
SELECT custid, empid, cnt
FROM C
WHERE rn = 1;
```

This query generates the following output, shown here in abbreviated form:

```
custid      empid       cnt
----------- ----------- -----------
1           4           2
2           3           2
3           3           3
4           4           4
5           3           6
6           9           3
7           4           3
8           4           2
9           4           4
10          3           4
11          6           2
12          8           2
...
```

If you want to return all ties, simply use the RANK function instead of ROW_NUMBER and calculate it based on count ordering alone (without the employee ID tiebreaker), like so:

```
WITH C AS
(
  SELECT custid, empid, COUNT(*) AS cnt,
    RANK() OVER(PARTITION BY custid
                ORDER BY COUNT(*) DESC) AS rn
  FROM Sales.Orders
  GROUP BY custid, empid
)
SELECT custid, empid, cnt
FROM C
WHERE rn = 1;
```

This time, as you can see in the following output, ties are returned:

```
custid      empid       cnt
----------- ----------- -----------
1           1           2
1           4           2
2           3           2
3           3           3
4           4           4
5           3           6
6           9           3
7           4           3
8           4           2
9           4           4
10          3           4
11          6           2
11          4           2
11          3           2
12          8           2
...
```

In case you do want to apply a tiebreaker, you can use another solution that is very efficient. It is based on the concatenation technique that I presented earlier in the chapter. Write a query that groups the data by customer ID and employee ID, and for each group, concatenate the count of rows and the employee ID to a single value (call it *binval*). Define a table expression based on this query. Have the outer query group the data by customer ID and calculate for each customer the maximum *binval*. This maximum value contains the max count and within it the maximum employee ID. What's left is to extract the count and employee ID from the binary value by using the SUBSTRING function and convert the values to the original types. Here's the complete solution query:

```
SELECT custid,
  CAST(SUBSTRING(MAX(binval), 5, 4) AS INT) AS empid,
  CAST(SUBSTRING(MAX(binval), 1, 4) AS INT) AS cnt
FROM (SELECT custid,
        CAST(COUNT(*) AS BINARY(4)) + CAST(empid AS BINARY(4)) AS binval
      FROM Sales.Orders
      GROUP BY custid, empid) AS D
GROUP BY custid;
```

As an exercise, you can test the solutions against a table with a large number of rows. You will see that this solution is very fast.

Histograms

Histograms are powerful analytical tools that express the distribution of items. For example, suppose you need to figure out from the order information in the Sales.OrderValues view how many small, medium, and large orders you have, based on the order values. In other words, you need a histogram with three steps. The extreme values (the minimum and maximum values) are what defines values as small, medium, or large. Suppose for the sake of simplicity that the minimum order value is 10 and the maximum is 40. Take the difference between the two extremes (*40 – 10 = 30*) and divide it by the number of steps (3) to get the step size. In this case, it's 30 divided by 3, which is 10. So the boundaries of step 1 (small) would be 10 and 20; for step 2 (medium), they would be 20 and 30; and for step 3 (large), they would be 30 and 40.

To generalize this, let *mn = MIN(val)* and *mx = MAX(val)* and let *stepsize = (mx – mn) / @numsteps*. Given a step number *n*, the lower bound of the step (*lb*) is *mn + (n – 1) * stepsize* and the higher bound (*hb*) is *mn + n * stepsize*. Something is tricky here. What predicate do you use to bracket the elements that belong in a specific step? You can't use *val BETWEEN lb and hb* because a value that is equal to *hb* appears in this step and also in the next step, where it equals the lower bound. Remember that the same calculation yielded the higher bound of one step and the lower bound of the next step. One approach to deal with this problem is to increase each of the lower bounds besides the first by one so that they exceed the previous step's higher bounds. With integers, this is a fine solution, but with another data type (such as NUMERIC in our case) it doesn't work because there are potential values between adjacent steps but not within either one—between the cracks, so to speak.

What I like to do to solve the problem is keep the same value in both bounds, and instead of using BETWEEN, I use *val* >= *lb* and *val* < *hb*. This technique has its own issues, but I find it easier to deal with than the previous technique. The issue here is that the item with the highest quantity (40, in our simplified example) is left out of the histogram. To solve this, I add a very small number to the maximum value before calculating the step size: *stepsize = ((1E0*mx + 0.0000000001) – mn) / @numsteps*. This technique allows the item with the highest value to be included, and the effect on the histogram is otherwise negligible. I multiplied *mx* by the float value *1E0* to protect against the loss of the upper data point when *val* is typed as MONEY or SMALLMONEY.

So you need the following ingredients to generate the lower and higher bounds of the histogram's steps: *@numsteps* (given as input), step number (the *n* column from the Nums auxiliary table), *mn*, and *stepsize*, which I described earlier.

Here's the T-SQL code required to produce the step number, lower bound, and higher bound for each step of the histogram:

```
USE InsideTSQL2008;

DECLARE @numsteps AS INT;
SET @numsteps = 3;

SELECT n AS step,
  mn + (n - 1) * stepsize AS lb,
  mn + n * stepsize AS hb
FROM dbo.Nums
  CROSS JOIN
    (SELECT MIN(val) AS mn,
       ((1E0*MAX(val) + 0.0000000001) - MIN(val))
       / @numsteps AS stepsize
     FROM Sales.OrderValues) AS D
WHERE n < = @numsteps;
```

This code generates the following output:

```
step        lb                    hb
----------- --------------------- ----------------------
1           12.5                  5470.83333333337
2           5470.83333333337      10929.1666666667
3           10929.1666666667      16387.5000000001
```

You might want to encapsulate this code in a user-defined function to simplify the queries that return the actual histograms, like so:

```
IF OBJECT_ID('dbo.HistSteps') IS NOT NULL
  DROP FUNCTION dbo.HistSteps;
GO
CREATE FUNCTION dbo.HistSteps(@numsteps AS INT) RETURNS TABLE
AS
RETURN
  SELECT n AS step,
    mn + (n - 1) * stepsize AS lb,
    mn + n * stepsize AS hb
```

```
  FROM dbo.Nums
    CROSS JOIN
      (SELECT MIN(val) AS mn,
         ((1E0*MAX(val) + 0.0000000001) - MIN(val))
         / @numsteps AS stepsize
       FROM Sales.OrderValues) AS D
  WHERE n < = @numsteps;
GO
```

To test the function, run the following query, which will give you a three-row histogram steps table:

```
SELECT * FROM dbo.HistSteps(3) AS S;
```

To return the actual histogram, simply join the steps table and the OrderValues view on the predicate I described earlier (*val >= lb AND val < hb*), group the data by step number, and return the step number and row count:

```
SELECT step, COUNT(*) AS numorders
FROM dbo.HistSteps(3) AS S
  JOIN Sales.OrderValues AS O
    ON val >= lb AND val < hb
GROUP BY step;
```

This query generates the following histogram:

```
step        numorders
----------- -----------
1           803
2           21
3           6
```

You can see that there are 803 small orders, 21 medium orders, and 6 large order. To return a histogram with 10 steps, simply provide 10 as the input to the *HistSteps* function:

```
SELECT step, COUNT(*) AS numorders
FROM dbo.HistSteps(10) AS S
  JOIN Sales.OrderValues AS O
    ON val >= lb AND val < hb
GROUP BY step;
```

This query generates the following output:

```
step        numorders
----------- -----------
1           578
2           172
3           46
4           14
5           3
6           6
7           8
8           1
10          2
```

Note that because you're using an inner join, empty steps are not returned like in the case of step 9. To return empty steps also, you can use the following outer join query:

```
SELECT step, COUNT(val) AS numorders
FROM dbo.HistSteps(10) AS S
  LEFT OUTER JOIN Sales.OrderValues AS O
    ON val >= lb AND val < hb
GROUP BY step;
```

As you can see in the output of this query, empty steps are included this time:

```
step        numorders
----------- -----------
1           578
2           172
3           46
4           14
5           3
6           6
7           8
8           1
9           0
10          2
```

Note Notice that *COUNT(val)* is used here and not *COUNT(*)*. *COUNT(*)* would incorrectly return 1 for empty steps because the group has an outer row. You have to provide the COUNT function an attribute from the nonpreserved side (*Orders*) to get the correct count.

There's another alternative to taking care of the issue with the step boundaries and the predicate used to identify a match. You can simply check whether the step number is 1, in which case you subtract 1 from the lower bound. Then, in the query generating the actual histogram, you use the predicate *val > lb AND val <= hb*.

Another approach is to check whether the step is the last, and if it is, add 1 to the higher bound. Then use the predicate *val >= lb AND val < hb*.

Here's the revised function implementing the latter approach:

```
ALTER FUNCTION dbo.HistSteps(@numsteps AS INT) RETURNS TABLE
AS
RETURN
  SELECT n AS step,
    mn + (n - 1) * stepsize AS lb,
    mn + n * stepsize + CASE WHEN n = @numsteps THEN 1 ELSE 0 END AS hb
  FROM dbo.Nums
    CROSS JOIN
      (SELECT MIN(val) AS mn,
         (1E0*MAX(val) - MIN(val)) / @numsteps AS stepsize
    FROM Sales.OrderValues) AS D
  WHERE n < = @numsteps;
GO
```

And the following query generates the actual histogram:

```
SELECT step, COUNT(val) AS numorders
FROM dbo.HistSteps(3) AS S
  LEFT OUTER JOIN Sales.OrderValues AS O
    ON val >= lb AND val < hb
GROUP BY step;
```

Grouping Factor

In earlier chapters, Chapter 6 in particular, I described a concept called a *grouping factor*. I used it in a problem to isolate islands, or ranges of consecutive elements in a sequence. Recall that the grouping factor is the factor you end up using in your GROUP BY clause to identify the group. In the earlier problem, I demonstrated two techniques to calculate the grouping factor. One method was calculating the maximum value within the group (specifically, the smallest value that is both greater than or equal to the current value and followed by a gap). The other method used row numbers.

Because this chapter covers aggregates, it is appropriate to revisit this very practical problem. In my examples here, I'll use the Stocks table, which you create and populate by running the following code:

```
USE tempdb;

IF OBJECT_ID('Stocks') IS NOT NULL DROP TABLE Stocks;

CREATE TABLE dbo.Stocks
(
  dt    DATE NOT NULL PRIMARY KEY,
  price INT  NOT NULL
);
GO

INSERT INTO dbo.Stocks(dt, price) VALUES
  ('20090801', 13),
  ('20090802', 14),
  ('20090803', 17),
  ('20090804', 40),
  ('20090805', 40),
  ('20090806', 52),
  ('20090807', 56),
  ('20090808', 60),
  ('20090809', 70),
  ('20090810', 30),
  ('20090811', 29),
  ('20090812', 29),
  ('20090813', 40),
  ('20090814', 45),
  ('20090815', 60),
  ('20090816', 60),
```

```
  ('20090817', 55),
  ('20090818', 60),
  ('20090819', 60),
  ('20090820', 15),
  ('20090821', 20),
  ('20090822', 30),
  ('20090823', 40),
  ('20090824', 20),
  ('20090825', 60),
  ('20090826', 60),
  ('20090827', 70),
  ('20090828', 70),
  ('20090829', 40),
  ('20090830', 30),
  ('20090831', 10);

CREATE UNIQUE INDEX idx_price_dt ON Stocks(price, dt);
```

The Stocks table contains daily stock prices.

Note Stock prices are rarely restricted to integers, and there is usually more than one stock, but I'll use integers and a single stock for simplification purposes. Also, stock markets usually don't have activity on Saturdays; because I want to demonstrate a technique over a sequence with no gaps, I introduced rows for Saturdays as well, with the same value that was stored in the preceding Friday.

The request is to isolate consecutive periods where the stock price was greater than or equal to 50. Figure 8-2 has a graphical depiction of the stock prices over time, and the arrows represent the periods you're supposed to return.

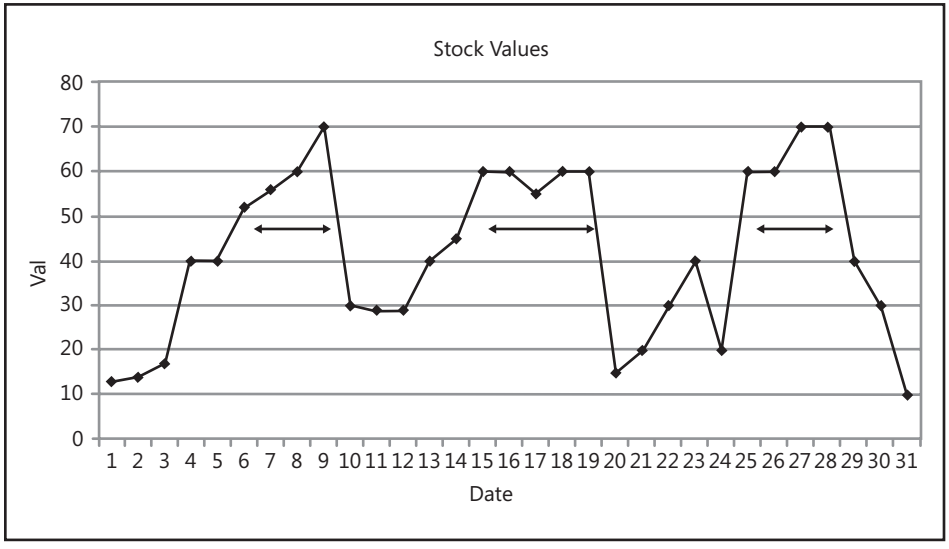


FIGURE 8-2 Periods in which stock values were greater than or equal to 50

For each such period, you need to return the starting date, ending date, duration in days, and the peak (maximum) price.

Let's start with a solution that does not use row numbers. The first step here is to filter only the rows where the price is greater than or equal to 50. Unlike the traditional problem where you really have gaps in the data, here the gaps appear only after filtering. The whole sequence still appears in the Stocks table. You can use this fact to your advantage. Of course, you could take the long route of calculating the maximum date within the group (the first date that is both later than or equal to the current date and followed by a gap). However, a much simpler and faster technique to calculate the grouping factor would be to return the first date that is greater than the current, on which the stock's price is less than 50. Here, you still get the same grouping factor for all elements of the same target group, yet you need only one nesting level of subqueries instead of two.

Here's the query:

```
SELECT MIN(dt) AS startrange, MAX(dt) AS endrange,
  DATEDIFF(day, MIN(dt), MAX(dt)) + 1 AS numdays,
  MAX(price) AS maxprice
FROM (SELECT dt, price,
        (SELECT MIN(dt)
         FROM dbo.Stocks AS S2
         WHERE S2.dt > S1.dt
          AND price < 50) AS grp
      FROM dbo.Stocks AS S1
      WHERE price >= 50) AS D
GROUP BY grp;
```

This query generates the following output, which is the desired result:

```
startrange endrange   numdays     maxprice
---------- ---------- ----------- -----------
2009-08-06 2009-08-09 4           70
2009-08-15 2009-08-19 5           60
2009-08-25 2009-08-28 4           70
```

Of course, post filtering, you could consider the problem as a classic islands problem in a temporal sequence scenario and address it with the very efficient technique that uses the ROW_NUMBER function, as I described in Chapter 6:

```
SELECT MIN(dt) AS startrange, MAX(dt) AS endrange,
  DATEDIFF(day, MIN(dt), MAX(dt)) + 1 AS numdays,
  MAX(price) AS maxprice
FROM (SELECT dt, price,
        DATEADD(day, -1 * ROW_NUMBER() OVER(ORDER BY dt), dt) AS grp
      FROM dbo.Stocks AS S1
      WHERE price >= 50) AS D
GROUP BY grp;
```

Grouping Sets

A grouping set is simply a set of attributes that you group by, such as in a query that has the following GROUP BY clause:

```
GROUP BY custid, empid, YEAR(orderdate)
```

You define a single grouping set—*(custid, empid, YEAR(orderdate))*. Traditionally, aggregate queries define a single grouping set, as demonstrated in the previous example. SQL Server supports features that allow you to define multiple grouping sets in the same query and return a single result set with aggregates calculated for the different grouping sets. The ability to define multiple grouping sets in the same query was available prior to SQL Server 2008 in the form of options called WITH CUBE and WITH ROLLUP and a helper function called GROUPING. However, those options were neither standard nor flexible enough. SQL Server 2008 introduces several new features that allow you to define multiple grouping sets in the same query. The new features include the GROUPING SETS, CUBE, and ROLLUP subclauses of the GROUP BY clause (not to be confused with the older WITH CUBE and WITH ROLLUP options) and the helper function GROUPING_ID. These new features are ISO compliant and substantially more flexible than the older, nonstandard ones.

Before I provide the technicalities of the grouping sets–related features, I'd like to explain the motivation for using those and the kind of problems that they solve. If you're interested only in the technicalities, feel free to skip this section.

Consider a data warehouse with a large volume of sales data. Users of this data warehouse frequently need to analyze aggregated views of the data by various dimensions, such as customer, employee, product, time, and so on. When a user such as a sales manager starts the analysis process, the user asks for some initial aggregated view of the data—for example, the total quantities for each customer and year. This request translates in more technical terms to a request to aggregate data for the grouping set *(custid, YEAR(orderdate))*. The user then analyzes the data, and based on the findings the user makes the next request—say, to return total quantities for each year and month. This is a request to aggregate data for a new grouping set—*(YEAR(orderdate), MONTH(orderdate))*. In this manner the user keeps asking for different aggregated views of the data—in other words, to aggregate data for different grouping sets.

To address such analysis needs of your system's users, you could develop an application that generates a different GROUP BY query for each user request. Each query would need to scan all applicable base data and process the aggregates. With large volumes of data, this approach is very inefficient, and the response time will probably be unreasonable.

To provide fast response time, you need to preprocess aggregates for all grouping sets that users might ask for and store those in the data warehouse. For example, you could do this every night. When the user requests aggregates for a certain grouping set, the aggregates will be readily available. The problem is that given n dimensions, 2^n possible grouping sets can be constructed from those dimensions. For example, with 10 dimensions you get 1,024 grouping sets. If you actually run a separate GROUP BY query for each, it will take a very long time to process all aggregates, and you might not have a sufficient processing window for this.

This is where the new grouping features come into the picture. They allow you to calculate aggregates for multiple grouping sets without rescanning the base data separately for each. Instead, SQL Server scans the data the minimum number of times that the optimizer figures is optimal, calculates the base aggregates, and on top of the base aggregates calculates the super aggregates (aggregates of aggregates).

Note that the product Microsoft SQL Server Analysis Services (SSAS, or just AS) specializes in preprocessing aggregates for multiple grouping sets and storing them in a specialized multidimensional database. It provides very fast response time to user requests, which are made with a language called Multidimensional Expressions (MDX). The recommended approach to handling needs for dynamic analysis of aggregated data is to implement an Analysis Services solution. However, some organizations don't need the scale and sophistication levels provided by Analysis Services and would rather get the most they can from their relational data warehouse with T-SQL. For those organizations, the new grouping features provided by SQL Server can come in very handy.

The following sections describe the technicalities of the grouping sets–related features supported by SQL Server 2008.

Sample Data

In my examples I will use the Orders table that you create and populate in tempdb by running the code provided earlier in Listing 8-1. This code is provided here again for your convenience:

```
SET NOCOUNT ON;
USE tempdb;

IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL DROP TABLE dbo.Orders;
GO

CREATE TABLE dbo.Orders
(
  orderid   INT        NOT NULL,
  orderdate DATETIME   NOT NULL,
  empid     INT        NOT NULL,
  custid    VARCHAR(5) NOT NULL,
  qty       INT        NOT NULL,
  CONSTRAINT PK_Orders PRIMARY KEY(orderid)
);
GO

INSERT INTO dbo.Orders
  (orderid, orderdate, empid, custid, qty)
VALUES
  (30001, '20060802', 3, 'A', 10),
  (10001, '20061224', 1, 'A', 12),
  (10005, '20061224', 1, 'B', 20),
  (40001, '20070109', 4, 'A', 40),
  (10006, '20070118', 1, 'C', 14),
```

```
  (20001, '20070212', 2, 'B', 12),
  (40005, '20080212', 4, 'A', 10),
  (20002, '20080216', 2, 'C', 20),
  (30003, '20080418', 3, 'B', 15),
  (30004, '20060418', 3, 'C', 22),
  (30007, '20060907', 3, 'D', 30);
```

The GROUPING SETS Subclause

SQL Server 2008 allows you to define multiple grouping sets in the same query by using the new GROUPING SETS subclause of the GROUP BY clause. Within the outermost pair of parentheses, you specify a list of grouping sets separated by commas. Each grouping set is expressed by a pair of parentheses containing the set's elements separated by commas. For example, the following query defines four grouping sets:

```
SELECT custid, empid, YEAR(orderdate) AS orderyear, SUM(qty) AS qty
FROM dbo.Orders
GROUP BY GROUPING SETS
(
  ( custid, empid, YEAR(orderdate) ),
  ( custid, YEAR(orderdate)        ),
  ( empid, YEAR(orderdate)         ),
  ()
);
```

The first grouping set is *(custid, empid, YEAR(orderdate))*, the second is *(custid, YEAR(orderdate))*, the third is *(empid, YEAR(orderdate))*, and the fourth is the empty grouping set *()*, which is used to calculate grand totals. This query generates the following output:

```
custid empid       orderyear   qty
------ ----------- ----------- -----------
A      1           2006        12
B      1           2006        20
NULL   1           2006        32
C      1           2007        14
NULL   1           2007        14
B      2           2007        12
NULL   2           2007        12
C      2           2008        20
NULL   2           2008        20
A      3           2006        10
C      3           2006        22
D      3           2006        30
NULL   3           2006        62
B      3           2008        15
NULL   3           2008        15
A      4           2007        40
NULL   4           2007        40
A      4           2008        10
NULL   4           2008        10
NULL   NULL        NULL        205
A      NULL        2006        22
B      NULL        2006        20
```

```
C       NULL        2006        22
D       NULL        2006        30
A       NULL        2007        40
B       NULL        2007        12
C       NULL        2007        14
A       NULL        2008        10
B       NULL        2008        15
C       NULL        2008        20
```

Note To specify a single-element grouping set, the parentheses are optional. (A one-element grouping set means the same as a simple group by item.) If you simply list elements directly within the outer pair of parentheses of the GROUPING SETS clause itself, as opposed to listing them within an inner pair of parentheses, you get a separate grouping set made of each element. For example, *GROUPING SETS(a, b, c)* defines three grouping sets: one with the element *a*, one with *b* and one with *c*. *GROUPING SETS((a, b, c))* defines a single grouping set made of the elements *a, b, c*.

As you can see in the output of the query, NULLs are used as placeholders in inapplicable attributes. You could also think of these NULLs as indicating that the row represents an aggregate over all values of that column. This way, SQL Server can combine rows associated with different grouping sets to one result set. So, for example, in rows associated with the grouping set *(custid, YEAR(orderdate))*, the *empid* column is NULL. In rows associated with the empty grouping set, the columns *empid*, *custid,* and *orderyear* are NULLs and so on.

Compared to a query that unifies the result sets of four GROUP BY queries, our query that uses the GROUPING SETS subclause requires much less code. It has a performance advantage as well. Examine the execution plan of this query shown in Figure 8-3.

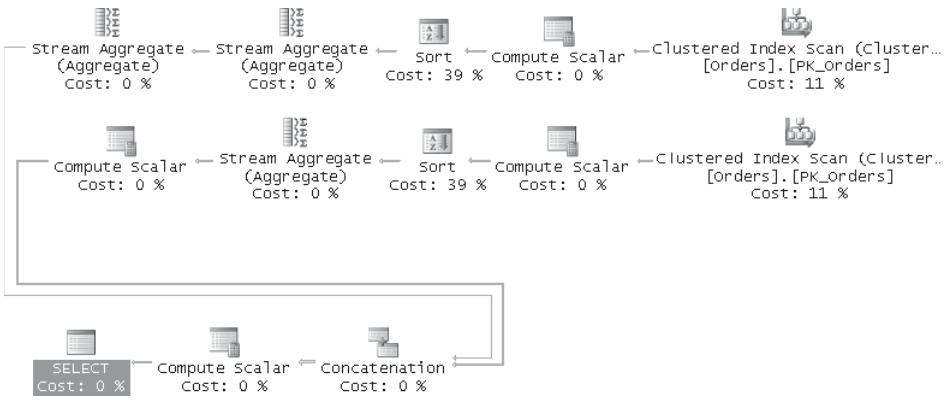


FIGURE 8-3 Execution plan of query with GROUPING SETS subclause

Observe that even though the query defines four grouping sets, the execution plan shows only two scans of the data. In particular, observe that the first branch of the plan shows two Stream Aggregate operators. The Sort operator sorts the data by *empid, YEAR(orderdate), custid*. Based on this sorting, the first Stream Aggregate operator calculates the aggregates for the grouping set *(custid, empid, YEAR(orderdate))*; the second Stream Aggregate operates

on the results of the first and calculates the aggregates for the grouping set *(empid, YEAR(orderdate))* and the empty grouping set. The second branch of the plan sorts the data by *YEAR(orderdate), custid* to allow the Stream Aggregate operator that follows to calculate aggregates for the grouping set *(custid, YEAR(orderdate))*.

Following is a query that is logically equivalent to the previous one, except that this one actually invokes four GROUP BY queries—one for each grouping set—and unifies their result sets:

```
SELECT custid, empid, YEAR(orderdate) AS orderyear, SUM(qty) AS qty
FROM dbo.Orders
GROUP BY custid, empid, YEAR(orderdate)

UNION ALL

SELECT custid, NULL AS empid, YEAR(orderdate) AS orderyear, SUM(qty) AS qty
FROM dbo.Orders
GROUP BY custid, YEAR(orderdate)

UNION ALL

SELECT NULL AS custid, empid, YEAR(orderdate) AS orderyear, SUM(qty) AS qty
FROM dbo.Orders
GROUP BY empid, YEAR(orderdate)

UNION ALL

SELECT NULL AS custid, NULL AS empid, NULL AS orderyear, SUM(qty) AS qty
FROM dbo.Orders;
```

The execution plan for this query is shown in Figure 8-4. You can see that the data is scanned four times.

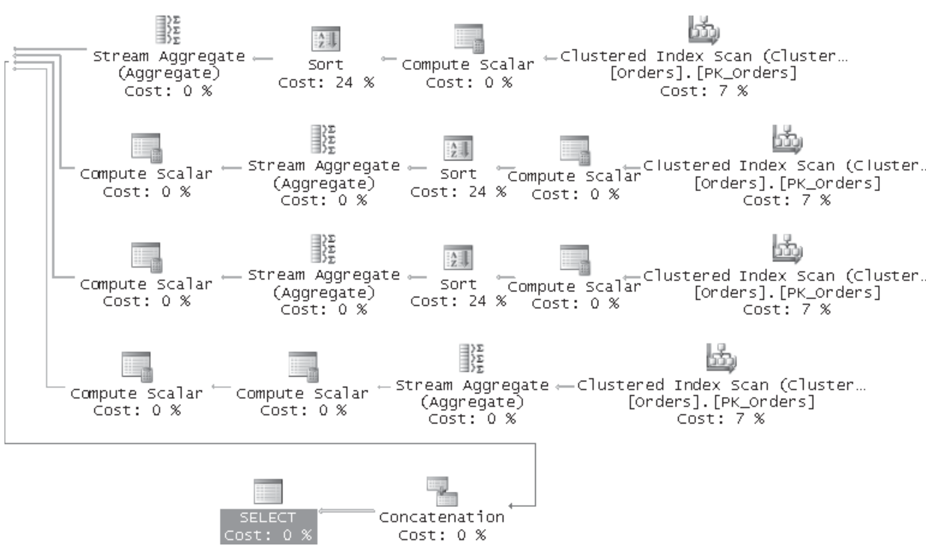


FIGURE 8-4 Execution plan of code unifying four GROUP BY queries

SQL Server 2008 allows you to define up to 4,096 grouping sets in a single query.

The CUBE Subclause

SQL Server 2008 also introduces the CUBE subclause of the GROUP BY clause (not to be confused with the older WITH CUBE option). The CUBE subclause is merely an abbreviated way to express a large number of grouping sets without actually listing them in a GROUPING SETS subclause. CUBE accepts a list of elements as input and defines all possible grouping sets out of those, including the empty grouping set. In set theory, this is called the *power set* of a set. The power set of a set V is the set of all subsets of V. Given *n* elements, CUBE produces 2^n grouping sets. For example, *CUBE(a, b, c)* is equivalent to *GROUPING SETS((a, b, c), (a, b), (a, c), (b, c), (a), (b), (c), ())*.

The following query uses the CUBE option to define all four grouping sets that can be made of the elements *custid* and *empid*:

```
SELECT custid, empid, SUM(qty) AS qty
FROM dbo.Orders
GROUP BY CUBE(custid, empid);
```

This query generates the following output:

```
custid empid       qty
------ ----------- -----------
A      1           12
B      1           20
C      1           14
NULL   1           46
B      2           12
C      2           20
NULL   2           32
A      3           10
B      3           15
C      3           22
D      3           30
NULL   3           77
A      4           50
NULL   4           50
NULL   NULL        205
A      NULL        72
B      NULL        47
C      NULL        56
D      NULL        30
```

The following query using the GROUPING SETS subclause is equivalent to the previous query:

```
SELECT custid, empid, SUM(qty) AS qty
FROM dbo.Orders
GROUP BY GROUPING SETS
  (
    ( custid, empid ),
    ( custid        ),
    ( empid         ),
    ()
  );
```

Note that each of the elements in the list you provide to CUBE as input can be made of either a single attribute or multiple attributes. The previous CUBE expression used two single-attribute elements. To define a multi-attribute element, simply list the element's attributes in parentheses. As an example, the expression CUBE(*x, y, z*) has three single-attribute elements and defines eight grouping sets: *(x, y, z), (x, y), (x, z), (y, z), (x), (y), (z), ()*. The expression CUBE((*x, y*), *z*) has one two-attribute element and one single-attribute element and defines four grouping sets: *(x, y, z), (x, y), (z), ()*.

Prior to SQL Server 2008, you could achieve something similar to what the CUBE subclause gives you by using a WITH CUBE option that you specified after the GROUP BY clause, like so:

```
SELECT custid, empid, SUM(qty) AS qty
FROM dbo.Orders
GROUP BY custid, empid
WITH CUBE;
```

This is an equivalent to our previous CUBE query, but it has two drawbacks. First, it's not standard, while the new CUBE subclause is. Second, when you specify the WITH CUBE option, you cannot define additional grouping sets beyond the ones defined by CUBE, while you can with the new CUBE subclause.

The ROLLUP Subclause

The new ROLLUP subclause of the GROUP BY clause is similar to the CUBE subclause. It also allows defining multiple grouping sets in an abbreviated way. However, while CUBE defines all possible grouping sets that can be made of the input elements (the power set), ROLLUP defines only a subset of those. ROLLUP assumes a hierarchy between the input elements. For example, *ROLLUP(a, b, c)* assumes a hierarchy between the elements a, *b,* and *c*. When there is a hierarchy, not all possible grouping sets that can be made of the input elements make sense in terms of having business value. Consider, for example, the hierarchy country, region, city. You can see the business value in the grouping sets *(country, region, city)*, *(country, region)*, *(country),* and *()*. But as grouping sets, *(city)*, *(region), (region, city)* and *(country, city)* have no business value. For example, the grouping set *(city)* has no business value because different cities can have the same name, and a business typically needs totals by city, not by city name. When the input elements represent a hierarchy, ROLLUP produces only the grouping sets that make business sense for the hierarchy. Given *n* elements, ROLLUP will produce *n + 1* grouping sets.

The following query shows an example of using the ROLLUP subclause:

```
SELECT
  YEAR(orderdate) AS orderyear,
  MONTH(orderdate) AS ordermonth,
  DAY(orderdate) AS orderday,
  SUM(qty) AS qty
FROM dbo.Orders
GROUP BY
  ROLLUP(YEAR(orderdate), MONTH(orderdate), DAY(orderdate));
```

Out of the three input elements, ROLLUP defines four (3 + 1) grouping sets—*(YEAR(orderdate), MONTH(orderdate), DAY(orderdate))*, *(YEAR(orderdate), MONTH(orderdate))*, *(YEAR(orderdate))*, and *()*. This query generates the following output:

```
orderyear   ordermonth  orderday    qty
----------- ----------- ----------- -----------
2006        4           18          22
2006        4           NULL        22
2006        8           2           10
2006        8           NULL        10
2006        9           7           30
2006        9           NULL        30
2006        12          24          32
2006        12          NULL        32
2006        NULL        NULL        94
2007        1           9           40
2007        1           18          14
2007        1           NULL        54
2007        2           12          12
2007        2           NULL        12
2007        NULL        NULL        66
2008        2           12          10
2008        2           16          20
2008        2           NULL        30
2008        4           18          15
2008        4           NULL        15
2008        NULL        NULL        45
NULL        NULL        NULL        205
```

This query is equivalent to the following query that uses the GROUPING SETS subclause to define the aforementioned grouping sets explicitly:

```
SELECT
  YEAR(orderdate) AS orderyear,
  MONTH(orderdate) AS ordermonth,
  DAY(orderdate) AS orderday,
  SUM(qty) AS qty
FROM dbo.Orders
GROUP BY
  GROUPING SETS
  (
    ( YEAR(orderdate), MONTH(orderdate), DAY(orderdate) ),
    ( YEAR(orderdate), MONTH(orderdate)                 ),
    ( YEAR(orderdate)                                   ),
    ()
  );
```

Like with CUBE, each of the elements in the list you provide to ROLLUP as input can be made of either a single attribute or multiple attributes. As an example, the expression *ROLLUP(x, y, z)* defines four grouping sets: *(x, y, z), (x, y), (x), ()*. The expression *ROLLUP((x, y), z)* defines three grouping sets: *(x, y, z), (x, y), ()*.

Similar to the WITH CUBE option that I described earlier, previous versions of SQL Server prior to SQL Server 2008 supported a WITH ROLLUP option. Following is a query that is equivalent to the previous ROLLUP query, except that it uses the older WITH ROLLUP option:

```
SELECT
  YEAR(orderdate) AS orderyear,
  MONTH(orderdate) AS ordermonth,
  DAY(orderdate) AS orderday,
  SUM(qty) AS qty
FROM dbo.Orders
GROUP BY YEAR(orderdate), MONTH(orderdate), DAY(orderdate)
WITH ROLLUP;
```

Like the WITH CUBE option, the WITH ROLLUP option is nonstandard and doesn't allow you to define further grouping sets in the same query.

Grouping Sets Algebra

One beautiful thing about the design of the grouping sets–related features implemented in SQL Server 2008 is that they support a whole algebra of operations that can help you define a large number of grouping sets using minimal coding. You have support for operations that you can think of as multiplication, division, and addition.

Multiplication

Multiplication means producing a Cartesian product of grouping sets. You perform multiplication by separating GROUPING SETS subclauses (or the abbreviated CUBE and ROLLUP subclauses) by commas. For example, if A represents a set of attributes *a1, a2, . . ., an*, and B represents a set of attributes *b1, b2, . . ., bn*, and so on, the product *GROUPING SETS((A), (B), (C)), GROUPING SETS((D), (E))* is equal to *GROUPING SETS ((A, D), (A, E), (B, D), (B, E), (C, D), (C, E))*.

Consider the following query and try to figure out which grouping sets it defines:

```
SELECT custid, empid,
  YEAR(orderdate) AS orderyear,
  MONTH(orderdate) AS ordermonth,
  DAY(orderdate) AS orderday,
  SUM(qty) AS qty
FROM dbo.Orders
GROUP BY
  CUBE(custid, empid),
  ROLLUP(YEAR(orderdate), MONTH(orderdate), DAY(orderdate));
```

First, expand the CUBE and ROLLUP subclauses to the corresponding GROUPING SETS subclauses, and you get the following query:

```
SELECT custid, empid,
  YEAR(orderdate) AS orderyear,
  MONTH(orderdate) AS ordermonth,
  SUM(qty) AS qty
```

```
FROM dbo.Orders
GROUP BY
  GROUPING SETS
  (
    ( custid, empid ),
    ( custid        ),
    ( empid         ),
    ()
  ),
  GROUPING SETS
  (
    ( YEAR(orderdate), MONTH(orderdate), DAY(orderdate) ),
    ( YEAR(orderdate), MONTH(orderdate)                 ),
    ( YEAR(orderdate)                                   ),
    ()
  );
```

Now apply the multiplication between the GROUPING SETS subclauses, and you get the following query:

```
SELECT custid, empid,
  YEAR(orderdate) AS orderyear,
  MONTH(orderdate) AS ordermonth,
  SUM(qty) AS qty
FROM dbo.Orders
GROUP BY
  GROUPING SETS
  (
    ( custid, empid, YEAR(orderdate), MONTH(orderdate), DAY(orderdate) ),
    ( custid, empid, YEAR(orderdate), MONTH(orderdate)                 ),
    ( custid, empid, YEAR(orderdate)                                   ),
    ( custid, empid                                                    ),
    ( custid, YEAR(orderdate), MONTH(orderdate), DAY(orderdate)        ),
    ( custid, YEAR(orderdate), MONTH(orderdate)                        ),
    ( custid, YEAR(orderdate)                                          ),
    ( custid                                                           ),
    ( empid, YEAR(orderdate), MONTH(orderdate), DAY(orderdate)         ),
    ( empid, YEAR(orderdate), MONTH(orderdate)                         ),
    ( empid, YEAR(orderdate)                                           ),
    ( empid                                                            ),
    ( YEAR(orderdate), MONTH(orderdate), DAY(orderdate)                ),
    ( YEAR(orderdate), MONTH(orderdate)                                ),
    ( YEAR(orderdate)                                                  ),
    ()
  );
```

Division

When multiple grouping sets in an existing GROUPING SETS subclause share common elements, you can separate the common elements to another GROUPING SETS subclause and multiply the two. The concept is similar to arithmetic division, where you divide operands of an expression by a common element and pull it outside the parentheses. For example, (5×3 + 5×7) can be expressed as (5)×(3 + 7). Based on this logic, you can sometimes reduce

the amount of code needed to define multiple grouping sets. For example, see if you can reduce the code in the following query while preserving the same grouping sets:

```
SELECT
  custid,
  empid,
  YEAR(orderdate) AS orderyear,
  MONTH(orderdate) AS ordermonth,
  SUM(qty) AS qty
FROM dbo.Orders
GROUP BY
  GROUPING SETS
  (
    ( custid, empid, YEAR(orderdate), MONTH(orderdate) ),
    ( custid, empid, YEAR(orderdate)                   ),
    ( custid,        YEAR(orderdate), MONTH(orderdate) ),
    ( custid,        YEAR(orderdate)                   ),
    ( empid,         YEAR(orderdate), MONTH(orderdate) ),
    ( empid,         YEAR(orderdate)                   )
  );
```

Because *YEAR(orderdate)* is a common element to all grouping sets, you can move it to another GROUPING SETS subclause and multiply the two, like so:

```
SELECT
  custid,
  empid,
  YEAR(orderdate) AS orderyear,
  MONTH(orderdate) AS ordermonth,
  SUM(qty) AS qty
FROM dbo.Orders
GROUP BY
  GROUPING SETS
  (
    ( YEAR(orderdate)                 )
  ),
  GROUPING SETS
  (
    ( custid, empid, MONTH(orderdate) ),
    ( custid, empid                   ),
    ( custid,        MONTH(orderdate) ),
    ( custid                          ),
    ( empid,         MONTH(orderdate) ),
    ( empid                           )
  );
```

Note that when a GROUPING SETS subclause contains only one grouping set, it is equivalent to listing the grouping set's elements directly in the GROUP BY clause. Hence, the previous query is logically equivalent to the following:

```
SELECT
  custid,
  empid,
  YEAR(orderdate) AS orderyear,
```

```
  MONTH(orderdate) AS ordermonth,
  SUM(qty) AS qty
FROM dbo.Orders
GROUP BY
  YEAR(orderdate),
  GROUPING SETS
  (
    ( custid, empid, MONTH(orderdate) ),
    ( custid, empid                   ),
    ( custid,        MONTH(orderdate) ),
    ( custid                          ),
    ( empid,         MONTH(orderdate) ),
    ( empid                           )
  );
```

You can reduce this form even further. Notice in the remaining GROUPING SETS subclause that three subsets of elements appear once with *MONTH(orderdate)* and once without. Hence, you can reduce this form to a multiplication between a GROUPING SETS subclause containing those three and another containing two grouping sets, *(MONTH(orderdate))* and the empty grouping set, like so:

```
SELECT
  custid,
  empid,
  YEAR(orderdate) AS orderyear,
  MONTH(orderdate) AS ordermonth,
  SUM(qty) AS qty
FROM dbo.Orders
GROUP BY
  YEAR(orderdate),
  GROUPING SETS
  (
    ( custid, empid ),
    ( custid        ),
    ( empid         )
  ),
  GROUPING SETS
  (
    ( MONTH(orderdate) ),
    ()
  );
```

Addition

Recall that when you separate GROUPING SETS, CUBE, and ROLLUP subclauses by commas, you get a Cartesian product between the sets of grouping sets that each represents. But what if you have an existing GROUPING SETS subclause and you just want to add—not multiply—the grouping sets that are defined by a CUBE or ROLLUP subclause? This can be achieved by specifying the CUBE or ROLLUP subclause (or multiple ones) within the parentheses of the GROUPING SETS subclause.

For example, the following query demonstrates adding the grouping sets defined by a ROLLUP subclause to the grouping sets defined by the hosting GROUPING SETS subclause:

```
SELECT
  custid,
  empid,
  YEAR(orderdate) AS orderyear,
  MONTH(orderdate) AS ordermonth,
  SUM(qty) AS qty
FROM dbo.Orders
GROUP BY
  GROUPING SETS
  (
    ( custid, empid ),
    ( custid        ),
    ( empid         ),
    ROLLUP(YEAR(orderdate), MONTH(orderdate), DAY(orderdate))
  );
```

This query is a logical equivalent of the following query:

```
SELECT
  custid,
  empid,
  YEAR(orderdate) AS orderyear,
  MONTH(orderdate) AS ordermonth,
  SUM(qty) AS qty
FROM dbo.Orders
GROUP BY
  GROUPING SETS
  (
    ( custid, empid ),
    ( custid        ),
    ( empid         ),
    ( YEAR(orderdate), MONTH(orderdate), DAY(orderdate) ),
    ( YEAR(orderdate), MONTH(orderdate)                 ),
    ( YEAR(orderdate)                                   ),
    ()
  );
```

Unfortunately, there is no built-in option to do subtraction. For example, you can't somehow express the idea of CUBE*(a, b, c, d)* minus GROUPING SETS *((a, c), (b, d), ())*. Of course, you can achieve this with the EXCEPT set operation and other techniques but not as a direct algebraic operation on grouping sets–related subclauses.

The GROUPING_ID Function

In your applications you may need to be able to identify the grouping set with which each result row of your query is associated. Relying on the NULL placeholders may lead to convoluted code, not to mention the fact that if a column is defined in the table as allowing NULLs, a NULL in the result will be ambiguous. SQL Server 2008 introduces a very convenient tool for this

purpose in the form of a function called GROUPING_ID. This function accepts a list of attributes as input and constructs an integer bitmap where each bit represents the corresponding attribute (the rightmost bit represents the rightmost input attribute). The bit is 0 when the corresponding attribute is a member of the grouping set and 1 otherwise.

You provide the function with all attributes that participate in any grouping set as input, and you will get a unique integer representing each grouping set. So, for example, the expression GROUPING_ID*(a, b, c, d)* would return 0 (0×8 + 0×4 + 0×2 + 0×1) for rows associated with the grouping set *(a, b, c, d)*, 1 (0×8 + 0×4 + 0×2 + 1×1) for the grouping set *(a, b, c)*, 2 (0×8 + 0×4 + 1×2 + 0×1) for the grouping set *(a, b, d)*, 3 (0×8 + 0×4 + 1×2 + 1×1) for the grouping set *(a, b)*, and so on.

The following query demonstrate the use of the GROUPING_ID function:

```
SELECT
  GROUPING_ID(
    custid, empid,
    YEAR(orderdate), MONTH(orderdate), DAY(orderdate) ) AS grp_id,
  custid, empid,
  YEAR(orderdate) AS orderyear,
  MONTH(orderdate) AS ordermonth,
  DAY(orderdate) AS orderday,
  SUM(qty) AS qty
FROM dbo.Orders
GROUP BY
  CUBE(custid, empid),
  ROLLUP(YEAR(orderdate), MONTH(orderdate), DAY(orderdate));
```

This query generates the following output:

```
grp_id  custid empid  orderyear  ordermonth  orderday  qty
------- ------ ------ ---------- ----------- --------- ----
0       C      3      2006       4           18        22
16      NULL   3      2006       4           18        22
0       A      3      2006       8           2         10
24      NULL   NULL   2006       4           18        22
25      NULL   NULL   2006       4           NULL      22
16      NULL   3      2006       8           2         10
24      NULL   NULL   2006       8           2         10
25      NULL   NULL   2006       8           NULL      10
0       D      3      2006       9           7         30
16      NULL   3      2006       9           7         30
...
```

For example, the *grp_id* value 25 represents the grouping set *(YEAR(orderdate), MONTH(orderdate))*. These attributes are represented by the second (value 2) and third (value 4) bits. However, remember that the bits representing members that participate in the grouping set are turned off. The bits representing the members that do not participate in the grouping set are turned on. In our case, those are the first (1), fourth (8), and fifth (16) bits representing the attributes *DAY(orderdate)*, *empid* and *custid*, respectively. The sum of the values of the bits that are turned on is 1 + 8 + 16 = 25.

The following query helps you see which bits are turned on or off in each integer bitmap generated by the GROUPING_ID function with five input elements:

```
SELECT
  GROUPING_ID(e, d, c, b, a) as n,
  COALESCE(e, 1) as [16],
  COALESCE(d, 1) as [8],
  COALESCE(c, 1) as [4],
  COALESCE(b, 1) as [2],
  COALESCE(a, 1) as [1]
FROM (VALUES(0, 0, 0, 0, 0)) AS D(a, b, c, d, e)
GROUP BY CUBE (a, b, c, d, e)
ORDER BY n;
```

This query generates the following output:

```
n           16          8           4           2           1
----------- ----------- ----------- ----------- ----------- -----------
0           0           0           0           0           0
1           0           0           0           0           1
2           0           0           0           1           0
3           0           0           0           1           1
4           0           0           1           0           0
5           0           0           1           0           1
6           0           0           1           1           0
7           0           0           1           1           1
8           0           1           0           0           0
9           0           1           0           0           1
10          0           1           0           1           0
11          0           1           0           1           1
12          0           1           1           0           0
13          0           1           1           0           1
14          0           1           1           1           0
15          0           1           1           1           1
16          1           0           0           0           0
17          1           0           0           0           1
18          1           0           0           1           0
19          1           0           0           1           1
20          1           0           1           0           0
21          1           0           1           0           1
22          1           0           1           1           0
23          1           0           1           1           1
24          1           1           0           0           0
25          1           1           0           0           1
26          1           1           0           1           0
27          1           1           0           1           1
28          1           1           1           0           0
29          1           1           1           0           1
30          1           1           1           1           0
31          1           1           1           1           1
```

Remember—when the bit is off, the corresponding member *is* part of the grouping set.

As mentioned, the GROUPING_ID function was introduced in SQL Server 2008. You could produce a similar integer bitmap prior to SQL Server 2008, but it involved more work. You could use a function called GROUPING that accepts a single attribute as input and returns 0 if

the attribute is a member of the grouping set and 1 otherwise. You could construct the integer bitmap by multiplying the GROUPING value of each attribute by a different power of 2 and summing all values. Here's an example of implementing this logic in a query that uses the older WITH CUBE option:

```
SELECT
  GROUPING(custid)          * 4 +
  GROUPING(empid)           * 2 +
  GROUPING(YEAR(orderdate)) * 1 AS grp_id,
  custid, empid, YEAR(orderdate) AS orderyear,
  SUM(qty) AS totalqty
FROM dbo.Orders
GROUP BY custid, empid, YEAR(orderdate)
WITH CUBE;
```

This query generates the following output:

```
grp_id      custid empid       orderyear   totalqty
----------- ------ ----------- ----------- -----------
0           A      1           2006        12
0           B      1           2006        20
4           NULL   1           2006        32
0           A      3           2006        10
0           C      3           2006        22
0           D      3           2006        30
4           NULL   3           2006        62
6           NULL   NULL        2006        94
0           C      1           2007        14
4           NULL   1           2007        14
...
```

Materialize Grouping Sets

Recall that before I started describing the technicalities of the grouping sets–related features, I explained that one of their uses is to preprocess aggregates for multiple grouping sets and store those in the data warehouse for fast retrieval. The following code demonstrates materializing aggregates for multiple grouping sets, including an integer identifier of the grouping set calculated with the GROUPING_ID function in a table called MyGroupingSets:

```
USE tempdb;
IF OBJECT_ID('dbo.MyGroupingSets', 'U') IS NOT NULL  DROP TABLE dbo.MyGroupingSets;
GO

SELECT
  GROUPING_ID(
    custid, empid,
    YEAR(orderdate), MONTH(orderdate), DAY(orderdate) ) AS grp_id,
  custid, empid,
  YEAR(orderdate) AS orderyear,
  MONTH(orderdate) AS ordermonth,
  DAY(orderdate) AS orderday,
  SUM(qty) AS qty
INTO dbo.MyGroupingSets
```

```
FROM dbo.Orders
GROUP BY
  CUBE(custid, empid),
  ROLLUP(YEAR(orderdate), MONTH(orderdate), DAY(orderdate));

CREATE UNIQUE CLUSTERED INDEX idx_cl_grp_id_grp_attributes
  ON dbo.MyGroupingSets(grp_id, custid, empid, orderyear, ordermonth, orderday);
```

The index created on the table MyGroupingSets is defined on the *grp_id* column as the first key to allow efficient retrieval of all rows associated with a single grouping set. For example, consider the following query, which asks for all rows associated with the grouping set *(custid, YEAR(orderdate), MONTH(orderdate))*:

```
SELECT *
FROM dbo.MyGroupingSets
WHERE grp_id = 9;
```

This query generates the following output:

```
grp_id      custid empid       orderyear   ordermonth  orderday    qty
----------- ------ ----------- ----------- ----------- ----------- -----------
9           A      NULL        2006        8           NULL        10
9           A      NULL        2006        12          NULL        12
9           A      NULL        2007        1           NULL        40
9           A      NULL        2008        2           NULL        10
9           B      NULL        2006        12          NULL        20
9           B      NULL        2007        2           NULL        12
9           B      NULL        2008        4           NULL        15
9           C      NULL        2006        4           NULL        22
9           C      NULL        2007        1           NULL        14
9           C      NULL        2008        2           NULL        20
9           D      NULL        2006        9           NULL        30
```

Figure 8-5 shows the plan for this query.

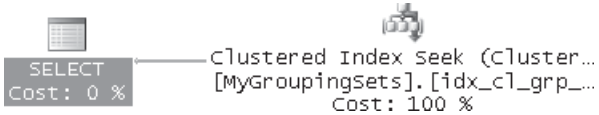


FIGURE 8-5 Execution plan of query that filters a single grouping set

This plan is very efficient. It scans only the rows that are associated with the requested grouping set because they reside in a consecutive section in the leaf of the clustered index.

Provided that you are using aggregates that are additive measures, like SUM, COUNT, and AVG, you can apply incremental updates to the stored aggregates with only the delta of additions since you last processed those aggregates. You can achieve this by using the new MERGE statement that was introduced in SQL Server 2008. Here I'm just going to show the code to demonstrate how this is done. For details about the MERGE statement, please refer to Chapter 10, "Data Modification."

Run the following code to simulate another day's worth of order activity (April 19, 2008):

```
INSERT INTO dbo.Orders
  (orderid, orderdate, empid, custid, qty)
VALUES
  (50001, '20080419', 1, 'A', 10),
  (50002, '20080419', 1, 'B', 30),
  (50003, '20080419', 2, 'A', 20),
  (50004, '20080419', 2, 'B',  5),
  (50005, '20080419', 3, 'A', 15)
```

Then run the following code to incrementally update the stored aggregates with the new day's worth of data:

```
WITH LastDay AS
(
  SELECT
    GROUPING_ID(
      custid, empid,
      YEAR(orderdate), MONTH(orderdate), DAY(orderdate) ) AS grp_id,
    custid, empid,
    YEAR(orderdate) AS orderyear,
    MONTH(orderdate) AS ordermonth,
    DAY(orderdate) AS orderday,
    SUM(qty) AS qty
  FROM dbo.Orders
  WHERE orderdate = '20080419'
  GROUP BY
    CUBE(custid, empid),
    ROLLUP(YEAR(orderdate), MONTH(orderdate), DAY(orderdate))
)
MERGE INTO dbo.MyGroupingSets AS TGT
USING LastDay AS SRC
  ON     (TGT.grp_id    = SRC.grp_id)
     AND (TGT.orderyear  = SRC.orderyear
          OR (TGT.orderyear IS NULL AND SRC.orderyear IS NULL))
     AND (TGT.ordermonth = SRC.ordermonth
          OR (TGT.ordermonth IS NULL AND SRC.ordermonth IS NULL))
     AND (TGT.orderday   = SRC.orderday
          OR (TGT.orderday IS NULL AND SRC.orderday IS NULL))
     AND (TGT.custid   = SRC.custid
          OR (TGT.custid IS NULL AND SRC.custid IS NULL))
     AND (TGT.empid    = SRC.empid
          OR (TGT.empid IS NULL AND SRC.empid IS NULL))
WHEN MATCHED THEN
  UPDATE SET
    TGT.qty += SRC.qty
WHEN NOT MATCHED THEN
  INSERT (grp_id, custid, empid, orderyear, ordermonth, orderday)
  VALUES (SRC.grp_id, SRC.custid, SRC.empid, SRC.orderyear, SRC.ordermonth, SRC.orderday);
```

The code in the CTE LastDay calculates aggregates for the same grouping sets as in the original query but filters only the last day's worth of data. The MERGE statement then increments the quantities of groups that already exist in the target by adding the new quantities and inserts the groups that don't exist in the target.

Sorting

Consider a request to calculate the total quantity aggregate for all grouping sets in the hierarchy order year > order month > order day. You can achieve this, of course, by simply using the ROLLUP subclause. However, a tricky part of the request is that you need to sort the rows in the output in a hierarchical manner, that is, days of a month, followed by the month total, months of a year followed by the yearly total, and finally the grand total. This can be achieved with the help of the GROUPING function as follows:

```
SELECT
  YEAR(orderdate)  AS orderyear,
  MONTH(orderdate) AS ordermonth,
  DAY(orderdate)   AS orderday,
  SUM(qty)         AS totalqty
FROM dbo.Orders
GROUP BY
  ROLLUP(YEAR(orderdate), MONTH(orderdate), DAY(orderdate))
ORDER BY
  GROUPING(YEAR(orderdate)) , YEAR(orderdate),
  GROUPING(MONTH(orderdate)), MONTH(orderdate),
  GROUPING(DAY(orderdate))  , DAY(orderdate);
```

Remember that the GROUPING function returns 0 when the element is a member of a grouping set (representing detail) and 1 when the element isn't (representing an aggregate). Because we want to present detail before aggregates, the GROUPING function is very convenient. We want to first see the detail of years and at the end the grand total. Within the detail of years, we want to sort by year. Within each year, we want to first see the detail of months and then the year total. Within the detail of months, we want to sort by month. Within the month we want to sort by the detail of days and then month total. Within the detail of days, we want to sort by day.

This query generates the following output:

```
orderyear   ordermonth  orderday    totalqty
----------- ----------- ----------- -----------
2006        4           18          22
2006        4           NULL        22
2006        8           2           10
2006        8           NULL        10
2006        9           7           30
2006        9           NULL        30
2006        12          24          32
2006        12          NULL        32
2006        NULL        NULL        94
2007        1           9           40
2007        1           18          14
2007        1           NULL        54
2007        2           12          12
2007        2           NULL        12
2007        NULL        NULL        66
```

```
2008        2           12          10
2008        2           16          20
2008        2           NULL        30
2008        4           18          15
2008        4           19          80
2008        4           NULL        95
2008        NULL        NULL        125
NULL        NULL        NULL        285
```

Conclusion

This chapter covered various solutions to data-aggregation problems that reused fundamental querying techniques I introduced earlier in the book. It also introduced new techniques, such as dealing with tiebreakers by using concatenation, calculating a minimum using the MAX function, pivoting, unpivoting, calculating custom aggregates by using specialized techniques, and more. This chapter also covered the new grouping sets features in SQL Server 2008 and showed how you can use those to efficiently address the need for dynamic analysis of aggregates.

As you probably noticed, data-aggregation techniques involve a lot of logical manipulation. If you're looking for ways to improve your logic, you can practice pure logical puzzles, which have a lot in common with querying problems in terms of the thought processes involved. You can find pure logic puzzles in Appendix A.

Index

Symbols and Numbers

#CachedPages sample table, 592–94
.NET
  CLR database code, 476–77
  reference types, 484
∈ (set membership operator), 44–45
@expression argument, 610–11
@length argument, 610–11
@myOD variable, 605
@offset argument, 610–11

A

abstraction
Accumulate method, 482–83
Actual Execution Plan, 653
acyclic graphs, 660
ad hoc paging, 350–51
ad hoc queries, 136
Add Outer Rows phase, 5, 11
adjacency list model, 99–100
AFTER triggers, 110
aggregate functions
  OVER clause, 29
  subqueries, 14
aggregate product, pivoting, 475
aggregate window functions, 454
aggregation, 445. *See also* pivoting
  bitwise operations specialized solution, 490–94
  cumulative, 453–57
  custom, 473–99
  duration by query, 155–57
  OVER clause, 445–48
  PIVOT operator, 24
  product specialized solution, 488–90
  query signature, 157–59
  running, 451–52
  sliding, 457–59
  specialized solutions, 487–99
  string concatenation specialized solution, 487–88
  tiebreakers, 448–51
  top wait isolation, 137–38
  user-defined aggregates (UDA), 476–82
  year-to-date, 459–60
Aldous, David, 292
algebra, relational, 90–104
algorithms, 43–44, 277–79. *See also* complexity
  binary search, 282
  joins, 421–29
  linear complexity, 133–34
  LISLP problem solution, 292
  O(n log n), 288–89
  quadratic sorting, 288
  running time comparsions, 286
  scale, 279–82
  sorting, 287–89
  swapping, 289
  ultra sort, 289
aliases
  column, 319–20, 322
  reuse, 16
  SELECT list, 14–15
  table, 606
ALL predicate, 316–18
all-at-once operations, 14–15
allocation order scans, 192, 208–19
allocation units, 189
alphabetical order, 43, 57–58
ALTER DATABASE option, 647
ALTER INDEX statement, 258
ALTER TABLE SWITCH, 645–46
An Introduction to Database Systems (Date), 83, 125
Analysis Services, wait analysis, 140
analytical ranking functions, 330–32
  NTILE, 354–59
  RANK and DENSE RANK, 352–54
  ROW_NUMBER, 332–52
  tile number, 354–59
ancestors, iteration/recursion, 681–84
Anchor Member, 328–30
anchor rows, 549–50, 592
And operator, 68–70
  ALL predicate, 317
  logical transformations, 556–59
ANSI SQL, 1
  aggregate window functions, 454
  constraints, 105
  cross joins, 396–97
  cursors, 17
  INSERT VALUES clause, 562
  join logical processing order, 409
  join syntax, 389–90
  nonsupported joins, 401
  NULL values, 111
  ORDER BY clause, 16
  outer joins, 399
  OVER clause subclauses, 459
  relations, 103
  semicolon termination, 322
  set operations, 436
  two-valued logic, 623
anti-semi joins, 415–16
antisymmetric relation properties, 75–76

ANY predicate, 316–18
APPLY operator, 21–22, 527, 535–36
  TOP n for each group, 537–43
arguments
  common table expressions, 323
  derived tables, 320–21
arrays, 287
  separating elements, 429–35
assembly creation and deployment, 482–87
assignment SELECT, 612–14
assignment UPDATE, 614–16
asterisk, 306
asymptotic complexity, 283
atomic types, 86
attributes
  pivoting, 460–64
  relations, 85–87
  scalar, 86–87
  tuples, 84
  types, 86
AUTO_CREATE_STATISTICS property, 228
auxiliary table of numbers, 359–62
average fragmentation in percent, 256–57
AVG aggregate, 453–57

B

bag theory, 64–65
balanced trees, 191
base columns, 14
bcp.exe, 565
BEFORE triggers, 109
benchmarks, row numbering, 344–48
Ben-Gan, Gabriel, 757
Ben-Gan, Itzik, 44
Bernoulli sampling algorithm, 268
BETWEEN predicate, 651–52
Big Oh notation, 283–84
BigNumSeq table, 364
bill of materials (BOM) example, 663–66
bin packing problem, 281
binary search algorithm, 282
bitmap filters, 426–28
Bitmap operator, 426–28
bitmap pages, 190–91
bitwise AND operation, 490–94
bitwise operations specialized solution, 490–94
bitwise OR operation, 490–94
bitwise XOR operation, 490–94
BLOBs, 290
block sequence values, 597–98
blocking sequences, 596
BOM sample table, 663–66
Boolean algebra, 74
Boolean expressions, 65–66
  restriction expression, 91–92
  T-SQL, 67
Boolean operators, 90–91
Boyce-Codd normal form, 117–19
braces, 45
B-trees, 189. *See also* subtrees; trees
  INSERT SELECT statement scenarios, 578–89
BULK INSERT statement, 567–68
BULK rowset provider, 565–67
bushy plans, 411–14

C

C# code
  UDA creation, 477–82
  user-defined functions, 160–61
cache
  clearing, 171–72
  query execution plans, 171
calculus, relational, 90–104
candidate keys, 105–06
cardinal numbers, 59–60
cardinality
  notation, 56
  sets, 56–57
Cartesian Product phase, 3, 7–8
Cartesian products, 53–54
  cross joins, 390–91
Cascade implementation, 107–08
CASE expressions
  aggregate product specialized solution, 489
  EXISTS predicate, 310–12
  NTILE function, 355
  outer joins, 400
  PIVOT operator, 24
  pivoting, 462–63
  unsupported logical phrases, 442
characteristic function definition, 55
CHARINDEX function, 432
CHECK constraints, 108–09, 670
  MERGE statement, 632
CHECKSUM, 554
Chen, Peter, 87
chiastic relationships, 410–11
Cities sample table, 666–70
CLR (Common Language Runtime). *See* Common Language Runtime (CLR)
Clustered Index Scan operator. *See* index scans; clustered indexes
Clustered Index Seek operator. *See* index seek; clustered indexes
clustered indexes, 191–95
  index seek + ordered partial scan, 250
  index tuning, 169–70
  ordered scan, 202–04
  seek + ordered partial scan, 233–38
  unordered index scan, 245
  unordered scan, 198–201
clustering key, 196
CMEMTHREAD wait, 136

COALESCE expression, 312
COALESCE function, 475
concurrency
  wait analysis, 137
Codd, Edgar F., 1
code revision, query tuning, 269–76
collation, 57–58
columns
  aliases, 14–16, 319–20, 322
  base, 14
  Boyce-Codd normal form, 117–19
  copy generation, 26–27
  extraction of elements, 27
  fifth normal form, 120–21
  first normal form, 113–15
  foreign key, 106–08
  fourth normal form, 119–20
  identity, 595–96
  IDENTITY property, 110
  included nonkey, 237
  key, 105–06
  nonunique sort column method, with tiebreaker, 337–38
  nonunique sort column method, without tiebreaker, 338–40
  pivoting. *See* pivoting
  second normal form, 115–16
  SELECT list ordering, 17
  set operations, 32
  spreading, 24
  third normal form, 116–17
  unique sort column method, 335–37
Common Language Runtime (CLR)
  database code, 476–77
  user-defined data type, 188
  user-defined functions, 160–61
common table expressions (CTEs), 321–22
  arguments, 323
  auxiliary table of numbers, 362
  column aliases, 322
  data modification, 324–25
  DELETE statement, 606
  EmpsPaths, 722–26
  EmpsRn, 722–26
  inline function definitions, 325–26
  level limiting, 680
  multiple, 323
  multiple references, 324
  recursive, 327–30
  Tiles, 495
  unsupported logical phrases, 442
  UPDATE statement, 608
  views, 325–26
  WITH keyword, 322
compatibility mode, 398
compatibility views, 171
Completed event classes, 150
complexity, 277–79. *See also* algorithms
  asymptotic, 283
  best- and worst-case, 283
  Big Oh notation, 283–84
  comparisons, 285–86
  constant, 283
  exponential and superexponential, 134–35
  linear, 133–34
  polynomial and nonpolynomial, 284–85
  sublinear, 282
  technical definitions, 283
composable DML, 636–38
composite joins, 397
Concatenation operator, 330
connected graphs, 660
consistency vs. correctness, 105
constant complexity, 283
Constant Scan operator, 643–45, 651–52
  parallelism, 652–57
constraints, 104–05
  check. *See* CHECK constraints
  declarative, 105–09
  join dependency, 121
  order of enforcement, 110
context, mathematics and, 41–43
contrapositives, 71
control-of-flow statements, 65–66
correctness vs. consistency, 105
correlated subqueries, 297–98, 302
  EXISTS predicate, 305–14
  tiebreaker, 302–06
COUNT aggregate, 466
COUNT(*), 14, 30, 655
COUNT(O.orderid), 14
COUNT(val), 502
covering indexes, 201
CREATE AGGREGATE command, 485
CREATE ASSEMBLY command, 485
CREATE CLUSTERED INDEX statement, 645
CREATE INDEX command, 548, 647–48
CREATE STATISTICS command, 645, 647
CREATE SYNONYM command, 360
CROSS APPLY operator, 21, 536
Cross Join phase. *See* Cartesian Product phase
cross joins, 7, 390–95. *See also* Cartesian Product phase
CTEs (common table expressions). *See* common table expressions (CTEs)
CUBE subclass, 506
CUBE subclause, 511–12
cumulative aggregation, 453–57
cursors, 17
  custom aggregations, 473
  gaps solution, 374
  islands solution, 383–84
  query tuning, 268–76
  row number calculation, 341–42

custom aggregation, 473–99
  pivoting, 474–99
custom sequences, 596–600
CustomerData sample table, 567
Customers sample table, 306, 308
  cross joins, 390–95
  hash joins, 425–26
  merge joins, 424–25
  MERGE statement, 616–17
  multiple joins, 408–11
  triggers, 627
  UPDATE statement, 607
CustomersDim sample table, 629
CustomersStage sample table, 616–17
CXPACKET wait, 136, 145
cycles, iteration/recursion, 691–94

D

Dafni, Adi, 757
DAG (directed acyclic graph).
    See directed acyclic graph (DAG)
data
  aggregation. *See* aggregation
  bad, domains and, 47–48
  collection, 187
  deletion, 601–06
  duplicate, removal, 601–03
  insertion, 561–601
  integrity, 104–11. *See also* constraints
  large value type updates, 610–11
  maintenance, materialized path, 695–701
  merging, 616–28
  model, 83
  modification, CTEs, 324–25
  modification, TOP option, 531–33
  OUTPUT clause, 628–38
  preparation, sample, 259–65
  processing, 83
  schema, 83
  structure, 277, 279
  temporal, 122
  trend identification, 291
  type. *See* types
  updating, 606–16
data collector, 187
data definition language (DDL), 460
  partitioned views and tables, 640
  triggers, 109
data integrity
  domain, 108–09
  enforcing, 109–11
  entity, 105–06
  referential, 106–08
Data Manipulation Language (DML), 460
  composable, 636–38
  constraints, 105
  relations, 103–04
  triggers, 109
Data Modeling Essentials (Simsion and Witt), 111–12
database
  data integrity. *See* data integrity
  FULL recovery model, 571–74
  generalization, 124–25
  I/O analysis, 145–48
  ID, 256
  non-FULL recovery mode, 574–75
  NULL values, 110–11
  relational model. *See* relational database model
  schema, 104
  specialization, 124–25
Database Design for Smarties (Muller), 112
Database Engine Tuning Advisor, 187
DATE type, 48
  binary string conversion, 450
date values, 42
Date, C. J., 83, 122, 125
DATEADD function, 368, 373, 392
DATEDIFF function, 373
DATETIME type, 417
  accuracy level, 458
  binary string conversion, 450
DBCC DROPCLEANBUFFERS, 118
DBCC FLUSHPROCINDB, 171
DBCC FREEPROCCACHE, 171
DBCC FREESYSTEMCACHE, 171–72
DBCC IND, 213–14
dbo.Customers table, 5–7
dbo.EmpYearValues table, 24–28
dbo.Orders table, 5–7
DDL (data definition language).
    See data definition language (DDL)
DecToBase function, 491
definitions, 38–39
  cardinality, 56
  Cartesian products, 54
  characteristic function of a set, 55
  complexity, 283
  logical operators, 69
  ordered pairs and tuples, 53
  propositions and predicates, 66
  set complement, 62
  set difference, 63
  set partition, 63
  subsets, 61
  undefined terms, 39
  union and intersect, 62–63
defragmentation utilities, 258
Degree of Parallelism event, 653
DELETE statement, 103–04, 601
  OUTPUT clause, 630–32
  TOP option, 531–33
DELETE trigger, 627–28
DeMorgan, Augustus, 70

DeMorgans laws, 70
denormalization, 122–24
DENSE_RANK function, 352–54, 383
derived tables, 318–19
  arguments, 320–21
  column aliases, 319–20
  multiple references, 321
  nesting, 320–21
Designing Database Solutions
    (Sarka, Leonard, Loria, and Wiernik), 122
determinism, 333–34
  RANK and DENSE_RANK functions, 353
  TOP option, 529–30
Diaconis, Persi, 292
Difference operator, 93–94
direct subordinates, 717–18
directed acyclic graph (DAG), 666
  transitive closure, 740–45
directed graphs, 659–60
Discard Results option, 344
Disk Usage collection set, 148
Disk Usage Summary report, 148
DISTINCT clause, 15–16, 369,
    371, 742–43
DISTINCT COUNT, 299
DISTINCT phase, 5
DISTINCT predicate, 414
Distribute Streams Parallelism operator, 652–57
Divide operator, 95–97
dividend relation, 95–97
divisor relation, 95–97
dm_db_index_operational_stats, 256
dm_db_index_usage_stats, 256
dm_db_index_physical_stats, 257
DMFs (Dynamic Management Functions).
    See specific DMFs
DML (Data Manipulation Language).
    See Data Manipulation Language (DML)
DMOs (Dynamic Management Objects), 172.
    See also specific DMOs
DMVs (Dynamic Management Views).
    See specific DMVs
domain integrity, 108–09
domain-key normal form, 122
domains, 84
  bad data, 47–48
  calculus, 102–03
  check constraint, 108–09
  modeling, 49
DROP statistics command, 645
DROP TABLE statement, 601
dta.exe command-line utility, 187
Dynamic Management Functions (DMFs).
    See specific DMFs
Dynamic Management Objects (DMOs), 172.
    See also specific DMOs
Dynamic Management Views (DMVs).
    See specific DMVs
dynamic pivoting, 487–88

E

edges, 99–100
Element Of operator, 90–91
elements, separating, 429–35
ellipsis, 45
employee organization chart example, 661–63
Employees sample table, 661–63
  cross joins, 390–95
  self joins, 402–04
  TOP n, 539–42
EmpOrders sample table, 451–52
empty sets, 54–55, 315
encapsulated types, 86
English-to-mathematics translation, 35–44
entity
  defined, 87
  primitive, 124
Entity Attribute Value (EAV), 460–61
entity integrity, 105–06
enumeration, sets, 45
equality, 39
Equals operator, 90–91
equi-joins, 94, 402–03
errors
  composite joins, 397
  duplicate key, 312
  ORDER BY table expressions, 18–19
  partitioned views updates, 640
  subqueries, 314–16
Estimated Execution Plan, 644
Estimated Subtree Cost, 178
Evaluate Expressions phase, 5
EXCEPT DISTINCT operation, 437–38
EXCEPT operation, 31–32, 435–39
excluded middle, law of, 68
exclusive locks, 257–58
Exclusive or, 70
execution plan, 2
  analysis, 174–85
  cached, 169–71
  graphical, 174–85
EXISTS predicate
  asterisk use, 306
  correlated subqueries, 305–14
  minimum missing values, 309–12
  semi joins, 414–16
  vs. IN predicate, 307
expand-collapse technique, 404
exponential complexity, 134–35
expressions
  logical transformations, 556–59
  TOP option, 530–31
Extend operator, 98
  T-SQL support, 103–04
extents, 188–89
external column aliasing, 319–20
external fragmentation, 256–57
external sorting, 287

F

factorial function, 281–82
faithfulness, 49–51
FALSE values, 9
FAST_FORWARD cursor, 268–69
Fermats Last Theorem, 110
fifth normal form, 120
fillfactor, 194, 257
filtering
  bitmap filters, 426–28
  indexes, statistics and, 239–42
filters, 8. *See also* specific filters
first normal form, 113–15
  bitwise operations, 490
first page request, 548–49
FLOAT data type, 41
fn_dblog function, 569
fn_trace_gettable function, 149, 155
FOR keyword, 464
FOR XML PATH option, 214
FOR XML query option, 487–88
FORCE ORDER hint, 406
  bushy plans, 413
foreign keys, 106–08
  nested loops, 423
forests, 661
format file, 565
Format.Native property, 484
Format.UserDefined property, 484
forwarding pointers, 191
fourth normal form, 119–20
fragmentation, 256–58
  logical index, 233–34
  logical scan, 192–93
Freedman, Craig, 429
FROM clause
  derived tables, 318
  MERGE statement, 618
  TABLESAMPLE, 265
FROM phase, 3, 7
FULL keyword, 397–401
FULL recovery model, 571–74
FULLSCAN, 647–48
functional dependencies
  multivalued dependency, 120
  normal forms, 112
functions, 43. *See also* specific functions
  aggregate. *See* aggregate functions
  aggregate window, 454
  analytical ranking, 330–32
  inline definitions, CTEs, 325–26
fuzzy logic, 75

G

Galindo-Legaria, Cesar, 273
gaps, 363–86
Gather Stream operator, 653–57
generalization
  database, 124–25
  relational database model, 124–26
GetAncestor method, 717–18
GetDescendant method, 711–12
GetFirstRows, 591–94
GetLevel method, 708
GetNextPage, 549–51
GetNextRows, 591–94
GetPrevPage, 551–52
GetReparentedValue, 712–14
GetSequence procedure, 598
GetTopProducts sample table, 535
Global Aggregation operator, 655
globally unique identifiers (GUIDs), 600–01
  random, 212
  temporary tables, 216
graph theory, 99–100
graphical execution plans, 174–85
graphs, 659–60. *See also* specific graphs
GROUP BY ALL, inner joins, 395–97
GROUP BY clause
  derived tables, 319
  grouping sets, 506–07
  relational division, 299
  self joins, 404
  subclasses, 506
GROUP BY phase, 5, 12–13
grouping factor, 503–05
GROUPING function, 524
grouping sets, 12–13, 506–07
  algebra, 514–18
  CUBE subclause, 511–12
  GROUPING SETS subclause, 508–10
  GROUPING_ID function, 518–21
  materialize, 521–23
  PIVOT operator, 23
  ROLLUP subclause, 512–14
  sample data, 507
  sorting, 524
GROUPING SETS subclass, 506
GROUPING SETS subclause, 508–10
  addition, 517–18
  division, 515–17
  multiplication, 514–15
GROUPING_ID function, 506, 518–21
Groups sample table, 473–74
  median, 554
GUIDs (globally unique identifiers).
    See globally unique identifiers (GUIDs)

H

Halpin, Terry, 88, 111–12
hash algorithm, 428
Hash Match operator, 426
hash tables, 425–28
HAVING clause, 80
  cumulative aggregations, 455–57

HAVING phase, 5, 13–14
heaps, 189–91
  INSERT SELECT statement scenarios, 575–78
Heisenberg Uncertainty Principle, 149
Heisenberg, Werner, 149
hierarchies, 99–100, 661
HIERARCHYID data type, 719
  list sorting, 726–30
  materialized path, 706–14
  normalizing, 719–23
  parent-child conversion, 724–26
hints, 185–86
  joins, 407–14
histograms, 499–503
HOBT, 189
Hungarian notation, 89

I

I/O subsystem
  AND logic costs, 558–59
  current and previous occurrence matching, 545
  OR logic costs, 558–59
  performance analysis, 145–48
  query costs, 224, 229
  reads, index seek cost, 193
  STATISTICS IO option, 172–73
  TOP n costs, 539, 542
  wait analysis, 136–37, 143, 145
IBinarySerialize interface, 484
identity, 39
IDENTITY function, 342–44
IDENTITY property
  inserting values, 110
  SELECT INTO statement, 564
  sequence mechanisms, 595–96
IF EXISTS, 626
IF keyword, 65–66
if.then statements, 70–72
ijk dialect, 40
IN predicate
  vs. EXISTS predicate, 307
IN_ROW_DATA allocation units, 189
Include Actual Execution Plan, 654
INCLUDE clause, 548
  filtered indexes, 240–41
included nonkey columns, 237
increasing subsequences, 291
Index Allocation Map (IAM) pages, 190–91
  allocation order scans, 192
index ID, 256
index keys
  updates, 219–23
index order scans, 204, 208
Index Scan operator, 205
  allocation order scans, 208–12
  index order scans, 219
index scans, 544
  allocation order scans, 192, 208–19
  APPLY operator, 546
  index order scans, 204, 208
  ordered clustered index, 202–04
  ordered covering nonclustered index scan, 204–07
  Storage Engine, 207–23, 256
  strategy analysis, 244–56
  unordered clustered index, 198–201, 245
  unordered covering index scan, 245–46
  unordered covering nonclustered index, 201–02
index seek, 193, 544
  clustered index seek + ordered partial scan, 233–38, 250
  covering nonclustered index seek + ordered partial scan, 251
  nonclustered index seek + ordered partial scan + lookups, 223–28, 247–50
  partion elimination, 649–50
  subtree removal, 700–01
  TOP n, 539
  unordered nonclustered index scan + lookups, 228–33, 246–47
Index Seek operator, 223–26
indexed views, 242–44
indexes
  access methods, 197–239. *See also* index scans; index seek
  clustered. *See* clustered indexes
  costs, 238
  covering, 201
  covering index seek + ordered partial scan, 251
  filtered, statistics and, 239–42
  fragmentation, 192–93, 256–58
  index seek + ordered partial scan + lookups, 247–50
  intersection, 238–39
  joins and, 421–23
  level calculations, 193–95
  nonclustered index seek + ordered partial scan + lookups method, 223–28
  on a clustered table, 196–97
  on a heap, 195–96
  ordered covering scan, 204–07
  pages and extents, 188–89
  partitioning, 258–59
  performance monitoring, 256
  rebuilding, 257–58
  rebuilds, 648
  reorganizing, 251
  strategy analysis, 244–56
  tuning, 169–70, 188–97. *See* index tuning
  unordered covering scan, 201–02, 245–46
  unordered index scan + lookups, 246–47
  unordered nonclustered index scan + lookups, 228–33
INDEXPROPERTY function, 193
induced order, 59

Information Modeling and Relational Databases (Halpin and Morgan), 88, 111–12
Information Principle, 83
Init method, 482–83
inline column aliasing, 319–20
inline function definitions, CTEs, 325–26
inner joins, 395–97
  sliding total sample, 417–20
  strategy forcing, 428–29
input expressions
  TOP option, 530–31
INSERT EXEC statement, 590–94
INSERT loop, 360
INSERT SELECT FROM OPENROWSET statement, 566
  minimal logging, 567–68
INSERT SELECT statement
  CASE expression, 310–12
  minimal logging, 567–68
  minimal logging summary, 590
  TABLOCK hint, heap, B-tree, TF-610, key range scenarios, 575–89
INSERT statement, 103–04
  auxiliary table of numbers, 360–62
  MERGE statement, 617–21
  OUTPUT clause, 629–30
  TOP option, 531–33
INSERT TOP, 532
INSERT trigger, 627–28
INSERT VALUES statement, 562
insertion sort, 288
Inside Microsoft SQL Server 2008, 105, 109, 122, 127, 318
INSTEAD OF triggers, 109
instructions, 43–44
integrity
  domain, 108–09
  entity, 105–06
  referential, 106–08
interchangeability, principle of, 88
internal fragmentation, 257
INTERSECT operation, 435–36, 439–40
  precedence, 440
Intersect operator, 93, 31–32
  T-SQL support, 103–04
intersect, set, 62–63
IntervalWaits function, 139–40
INTO clause, 441
intractable problems, 285
IP address, 704–30
irreflexive relation properties, 75–76
IsDescendantOf method, 715–16
IsInvariantToDuplicates property, 484–85
IsInvariantToNulls property, 484–85
IsInvariantToOrder property, 484–85
islands, 363–86
  variation, 384–86
IsNullIfEmpty property, 484–85
isolation levels, 211–12
iteration/recursion, 670
  ancestors, 681–84
  cycles, 691–94
  sorting, 688–91
  subgraph/subtree with path enumeration, 685–88
  subordinates, 671–81
iterative/procedural query tuning vs. set-based approaches, 268–76

J

Jensen, Clifford, 757
join hints, 185–86
JOIN keyword, 185, 428–29
Join operator, 94
  T-SQL support, 103–04
joins, 389
  algorithims, 421–29
  anti-semi, 415–16
  composite, 397
  cross, 390–95
  DELETE statement, 603–06
  dependency constraints, 121
  equi-, 94, 402–03
  hash, 428
  hints, 407–14
  inner, 395–97
  logical evaluation order, 408–11
  logical processing phase, 390
  many-to-many, 423
  merge, 423–25
  multiple, 405–06
  nested loops, 422–23
  nonsupported, 401
  old vs. new style, 389–403
  outer, 397–401
  self, 402–04
  semi, 98
  semi joins, 414–16
  theta, 94
  UPDATE statement, 606–10

K

Kass, Steve, 35, 267–68, 277
Kelly, Andrew J., 127
key lookups, 196–97
key-range, INSERT SELECT statement scenarios, 579–89
keys. *See also* foreign keys; primary keys
  Boyce-Codd normal form, 117–19
  duplicate, 312
  entity integrity, 105–06
  first normal form, 113–15
  natural vs. surrogate, 106

NULL values, 106
second normal form, 115–16
third normal form, 116–17
uniqueness and applicability, 106
Kogan, Eugene, 487
k-tuples, 53

L

L_SUPPKEY, 648
large object (LOB) data, 565–67
large value type updates, 610–11
LargeOrders sample table, 533, 630
LastDay CTE, 523
latch waits, 137
law of excluded middle, 68
LCK waits, 137
leaf level, 191–95
split pages, 192–93
leaf nodes, 718, 738–39
leaf_row_size, 193
left input, 20–21
APPLY operator, 21–22
LEFT keyword, 397–401
LEFT OUTER join, 543
left semi joins, 414–16
Leonard, Andy, 122
LIKE condition, 702
LIKE predicate, 232, 727
linear complexity, 133–34
LINEITEM sample table, 641–45
LINEITEMPART sample table, 641–45
lists, 287
LOB_DATA allocation units, 189
locks
exclusive, 257–58
index rebuilds, 257–58
shared, 219, 257–58
wait analysis, 137
LOG function, 489–90
logging
analysis, 569–71
minimally logged operations, 567–90
testing insert scenarios, 571–89
logic. *See also* fuzzy logic; predicate logic
puzzles, 757–77
three-valued, 9, 74
two-valued, 623
logical equivalence, 70
logical index fragmentation, 233–34
allocation order scans, 208–19
logical operators, 68–70. *See also* specific operators
logical query processing, 1–2
OVER clause, 29–31
phases, 2–5, 7–20. *See also* specific phases
phases, joins and, 390
sample query, 5–7
set operators, 31–32
table operators, 20–28
logical reads, 251–52
logical scan fragmentation, 192–93, 256–57
logical transformations, 556–59
longest increasing subsequence length problem (LISLP), 291–95
lookups
cost, 196
key, 196–97
RID, 196
Loria, Javier, 122

M

Machanic, Adam, 757
magnetic tape storage, 287
Management data warehouse, 187
manual partitioning, 88
materialize grouping sets, 521–23
materialized path, 694–95
data maintenance, 695–701
querying, 701–06
materialized path, HIERARCHYID data type, 706–08
data maintenance, 708–14
querying, 715–19
mathematics
context, 41–43
conventions, 39–40
definitions, 38–39
equality, identity, and sameness, 39
functions, parameters, and values, 43
graph theory, 99–100
grouping sets algebra, 514–18
instructions and algorithms, 43–44
median, 494–97, 554–56
mode, 497–99
numbers, 41
relational algebra and calculus, 90–104
set S, 35–37
well-definedness, 37–38
Matrix sample table, 468–69
MAX(order date), 302
MAX(ordered), 302–05
tiebreaker, 448–51
MAX(requireddate), 302–05
MaxByteSize property, 484
MAXDOP hint, 257
MAXRECURSION hint, 329–30, 680–81
MDX (Multidimensional Expressions), 507
median, 494–97
TOP option, 554–56
memory, wait analysis, 143
merge algorithm, 423–25
Merge Interval operator, 351

MERGE INTO clause, 618
Merge method, 482–83
MERGE predicate, 617–18
MERGE statement, 103–04, 294, 617–21
  multiple WHEN clauses, 623–24
  OUTPUT clause, 634–36
  predicate addition, 621–23
  triggers, 627–28
  values, 626–27
Messages sample table, 632
metadata table queries, 648
Microsoft SQL Server Customer Advisory Team, 158
minimally logged operations, 567–90
minimum missing values
  EXISTS predicate, 309–12
  outer joins, 400–01
Minus operator, 93–94
  T-SQL support, 103–04
missing values
  EXISTS predicate, 309–12
  outer joins, 400–01
mode, 497–99
modeling, 111–12
  domains, 49
  Object-Role Modeling (ORM), 111–12
  relational databases, 88
modifications
  TOP option, 531–33
modus ponens, 70
MonthlyOrders sample table, 417–20
Moran, Brian, 149
Morgan, Tony, 88, 111–12
Muller, Robert J., 112
Multidimensional Expressions (MDX), 507
multipage access, 351–52
multiple joins, 405–06
multiple references
  common table expressions, 324
  table expressions, 321
multiset theory, 64–119
multivalued dependencies, 120
multivalued subqueries, 297–98
mutator operators, 86
MyGroupingSets sample table, 521–22
MyOrders sample table, 557

N

naming conventions, 49–51. *See also* notation
  Hungarian notation, 89
  relational database model, 89
National Institute of Standards and Technology (NIST), 659
Natural Join operator, 94
natural keys, 106
natural numbers, 86
nave set theory, 52
nested loop algorithm, 422–23
Nested Loops operator, 544
  parallel query plans, 654–57
  partition elimination, 649
nested sets
  left and right value assignment, 731–36
  querying, 737–39
nesting, derived tables, 320–21
network waits, 145
NEWID function, 553, 601
NEWSEQUENTIALID function, 601
next page request, 549–51
next pointers, 204
NIST (National Institute of Standards and Technology), 659
No Action implementation, 107–08
NOCOUNT option, 618–19
nodes, 99–100
NOEXPAND hint, 244
NOLOCK hint
  allocation order, 215–19
  index order scan, 223
non_leaf_row_size, 194
nonblocking sequences, 598–600
non-equi-join joins
  sliding total sample, 417–20
non-FULL recovery mode, 574–75
nonpolynomial complexity, 284–85
nonscalar types, 86–87
nonunique sort column method
  with tiebreaker, 337–38
  without tiebreaker, 338–40
NORECOMPUTE option, 647
normal forms
  additional, 122
  Boyce-Codd, 117–19
  domain-key, 122
  fifth, 120
  first, 113–15
  fourth, 119–20
  functional dependencies, 112
  higher, 119–22
  second, 115–16
  sixth, 122
  third, 116–17
normalization, 111–22. *See also* normal forms
normalizing
  HIERARCHYID data type, 719–23
Not Equals operator, 90–91
NOT EXISTS predicate, 742
  semi joins, 415–16
  vs. NOT IN predicate, 307–09
NOT IN predicate
  semi joins, 415–16
  vs. NOT EXISTS predicate, 307–09
Not operator, 68–70

notation
Big Oh, 283–84
cardinality, 56
Hungarian, 89
ordered pairs and tuples, 53
set theory, 45–46
set-builder, 45–46
sets, 45–46
shorthand, 56
NP switch, 428
NTILE function, 354–59
NULL values, 9, 48
@expression, @length, and @value arguments, 611
aggregate product specialized solution, 489
COALESCE function, 475
EXCEPT DISTINCT operation, 437
filtered indexes, 239
GROUP BY phase, 13
GROUPING SETS subclause, 509
GROUPING_ID function, 518–21
HIERARCHYID data type, 710
in databases, 110–11
IN predicate, 307
INTERSECT operation, 439
key constraints, 106
multiple joins, 408
NOT EXISTS and NOT in predicate, 307–09
ORDER BY clause, 19–20
outer joins, 399
pivoting, 462–63
ranking functions, 336
row removal, UNPIVOT operator, 28
set operations, 32
specialization, 124–25
UNIQUE constraint, 241–42
UNPIVOT operator, 471
NULLIF, 489
num_leaf_pages, 194
num_rows, 193
numbers
cardinal, 59–60
mathematics and, 41
natural, 86
ordinal, 59–60
whichth, 60–61
numerical order, 57
Nums sample table, 131, 359–62
cross joins, 390–95
missing values, returning, 375–83
NumSeq table, 363–64
NVARCHAR data type, 188–89
NVARCHAR(MAX) data type, 189
NVARCHAR(MAX) type
updating, 610–11

O

O(n log n)
LISLP problem, 292
sorting algorithms, 288–89
object ID, 256
Object-Role Modeling (ORM), 88, 111–12
order
trichotomy, 58–59
offline index rebuilding, 257–58
OLEDB wait, 137
OLTP (online transaction processing). *See* online transaction processing (OLTP)
ON clause
bushy plans, 413
inner joins, 395–97
MERGE statement, 618
multiple joins, 409–11
ON filter, 3
OUTER JOIN clause, 12
ON filter phase, 8–10
online index rebuilding, 257–58
online transaction processing (OLTP)
MERGE statement, 616
wait analysis, 136
open schema, 460–62
OPENROWSET function, 565
OpenSchema sample table, 461–62
operators. *See also* specific operators
Boolean, 90–91
Codds, 91–97
cost percentages, 178
mutator, 86
relational algebra, 98–102
relations. *See* relations
relations and tuples, 90–91
selector, 86
set, 31–32
table, 20–28
ToolTip information, 179–85
type, 86
optimization. *See also* query optimizer
indexing strategies analysis, 244–56
nested loops, 421–22
partitioned views and partitioned tables, 640
optimized bitmap filters, 426–28
optimizer. *See* query optimizer
OPTION clause, 185
Or operator, 68–70
IN predicate, 316–17
logical transformations, 556–59
order, 57
alphabetical, 57–58
induced, 59
numerical, 57
sets, 57–61
total, 59

ORDER BY clause, 205
  cross joins, 393
  derived tables, 318–19
  ranking function, 331, 334
  TOP option, 16, 527, 534–35
ORDER BY list, 353
ORDER BY operation, 91, 436
ORDER BY phase, 5, 16–20
  OVER clause, 30–31
Order property, 208–12
  index order scans, 219
OrderDetails sample table, 465
  TOP n, 537–38
OrderDups sample table, 602–03
ordered pairs, 53–54
Ordered property, 204–05
  allocation order vs. index order scans, 207–08
Orders sample table, 131, 269–76, 306–07, 507
  data aggregation, 466–68
  hash joins, 425–26
  merge joins, 424–25
  multiple joins, 408–11
  TOP n, 537–38
OrdersArchive sample table, 631
ordinal numbers, 59–60
ORM (Object-Role Modeling), 88, 111–12
orthogonal design, 125–26
OUTER APPLY operator, 21, 536
OUTER JOIN clause, 12
outer joins, 11, 397–401
  filters, 12
  sliding total sample, 417–20
OUTER keyword, 398
OVER clause, 29–31
  aggregation, 445–70
  ranking functions, 331
  subcaluses, 459
Ozer, Stuart, 158

P

Pack operator, 100–01
Page Free Space (PFS) pages, 191
page splits, 191
  allocation order scans, 208–12
page_density, 194
PAGEIOLATCH_SH wait, 142
pages, 188–89
paging
  multipage access, 351–52
  row numbers, 349–52
  TOP option, 547–52
parallel queries, 228–31
parallel query plans, 136
  wait analysis, 145
Parallelism operators, 228, 652–57
parallelism, partitioning and, 652–57
parameters, 43
parent-child representation conversion, 724–26
parentheses, 322, 528
  chiastic relationships, 410–11
Partial Aggregation operator, 655
PARTITION BY clause, 30
  OVER clause, 447–48
  ranking functions, 331–32
  Segment operator, 333
partition ID, 256
partitioned row numbers, 344
partitioned tables, 639–40
  partition elimination, 649–52
  query plans, 641–45
  statistics, 645–48
  vs. partitioned views, 640
partitioned views, 639–40
partitioning. *See also* partitioned tables
  manual, 88
  parallelism, 652–57
  partitioned views, 639–40
  ranking functions, 334
  subqueries, 340–41
partitions sets, 63–64
Parts sample table, 663–66
Pascal, Fabian, 119
path enumeration, 685–88
PATH mode, 487–88
path queries, 716–17
penguin dialect, 39–40
PERCENT keyword, 265
PERCENT option, 528, 555
performance
  row number calculation, 344–49
  selectivity and query cost, 253–55
  tracing effects on, 149–50
  tuning methodology, 131–34
  workload tracing, 150–55
performance counters, 143–44
Performance sample database, 127–31
  join algorithms, 421
performance testing
  data preparation, 259–65
  TABLESAMPLE, 265–68
PerfWorkloadTraceStart procedure, 151
physical query processing, 2
PIVOT operator, 22–24, 463–64, 466, 470
  phases, 23–24
pivoting, 460
  aggregate product, 475
  attributes, 460–64
  custom aggregation, 474–99
  data aggregation, 466–70
  dynamic, 487–88
  relational division, 465–66
  string concatenation, 475
  unpivoting, 470–73

PivotTables
  wait analysis, 140–42
plan guides, 124
plan handles, 168
plan hash, 168
point queries, 233–34
Poletti, Marcello, 757
polynomial complexity, 284–85
pool cache, 171–72
POWER function, 489–90
Practical Issues in Database Management (Pascal), 119
precedence
  set operations, 440
predicate logic, 35, 65
  alternatives, 73–75
  DeMorgans laws, 70
  generalizations, 73–75
  implications, 70–72
  law of excluded middle, 68
  logical equivalence, 70
  operators, 68–70
  predicates. *See* predicates
  programming languages, 65–66
  propositions, 66–68
  quantification, 72–73
  relations, 75–80
predicates, 66–68
  MERGE statement additions, 621–23
  proposition creation from, 67–68
  quantified, negating, 73
  relations and, 87–88
  truth value, 68
  uncommon, subqueries, 316–18
preserved tables, 11
previous page request, 551–52
previous pointers, 204
primary keys, 105–06
  nested loops, 423
primitive entities, 124
principle of interchangeability, 88
process-level analysis, 148–50
  performance workload tracing, 150–55
  query statistics, 167–69
  trace data analysis, 155–67
product aggregate specialized solution, 488–90
Product operator, 92–93
  T-SQL support, 103–04
Profiler, 186
programming languages
  dialects, 40
  fourth-generation, 277
  predicate logic, 65–66
Project operator, 92
proof by contradiction, 68
proof by contrapositive, 71
propositional functions, 35
propositions, 66–68
  creation from predicates, 67–68
  relations and, 87–88
proto-tuple, 103
PvtCustOrders sample table, 470

Q

quadratic scaling, 280
quadratic sorting algorithms, 288
quantification
  multiple, 73
  predicate logic, 72–73
quantified statements, 72
  multiple, 73
  negating, 72–73
queries. *See also* query optimizer; query plan; query tuning
  ad hoc, 136
  aggregation, 156–67
  compilation, 640
  cost and performance statistics, 253–55
  cost percentages, 178–79
  execution plan. *See* execution plan
  filters. *See* specific filters
  HIERARCHYID data type, 715–19
  materialized path, 701–06
  nested sets, 737–39
  ORDER BY clause, 31
  parallel, 228–31
  partitioned tables. *See* partitioned tables
  path, 716–17
  plan guides, 124
  point, 233–34
  processing. *See* logical query processing; physical query processing
  range, 233–34
  recursive, CTEs, 327–30
  run time measurement, 173–74
  S set sample application, 77–80
  sample, 5–7
  selectivity vs. query cost, 253–55
  set operations. *See* set operations
  set-based, 268–76
  signature, 157–67
  statistics, 167–69
  subqueries. *See* subqueries
  wait analysis. *See* wait analysis
query hash, 168
query hints, 185–86
query optimizer, 2
  bitmap filter, 427
  Database Engine Tuning Advisor, 187
  hash table, 425–28
  hints, 185–86
  join hints, 407–14
  join strategy forcing, 428–29
  joins, 412–13

query optimizer *(continued)*
logical transformations, 556–59
merge joins, 423–25
paging, 350–52
relational algebra operators, 101–02
scan order, 273–76
semi joins, 415
query plans
parallel, 136, 145
parallelism, 652–57
partitioned tables, 641–45
query processing. *See* logical query processing; physical query processing
Query Statistics History report, 167
query tuning, 127
course of action determination, 145
database/file level analysis, 145–48
index tuning, 187–259. *See also* index tuning
indexes and queries, 169–70
methodology, 131–34
process level analysis, 148–69
sample data, 127–31
set-based vs. iterative/procedural approaches, 268–76
tools, 171–87
wait analysis, 134–43
wait correlation with queues, 143–44
queues, wait correlation, 143–44
quick sort, 289
QUOTENAME function, 488
quotient relation, 95–97

R

RAND function, 552–54
random vs. sequential, 193
Range Expression, 650–52
ranges, 108–09
missing and existing, 363–86
queries, 233–34
RANK function, 352–54
mode, 498
ranking functions, 60–61
analytical. *See* analytical ranking functions
gaps solution, 372–73
NULL values, 336
RDBMS (relational database management systems), 1, 83. *See also* relational database model
read committed isolation level, 219
Read method, 484
read uncommitted isolation, 219
READPAST hint, 633
real numbers, 41, 51
recursion. *See* iteration/recursion
recursive common table expressions, 327–30
Recursive Member, 328–30
Redistribute Streams operator, 653–57
references, multiple
common table expressions, 324
table expressions, 321
referential integrity, 106–08
reflexive relation properties, 75–76
RegexReplace function, 160–61
relational algebra, 90–104
operators, 98–102
T-SQL support, 103–04
relational calculus, 90–104
T-SQL support, 103–04
relational database management systems (RDBMS), 1, 83
relational database management systems (RDMBS), 83. *See also* relational database model
relational database model, 83
algebra and calculus, 90–104
data integrity, 104–11
denormalization, 122–24
generalization and specialization, 124–26
naming conventions, 89
normalization, 111–22
relations, tuples and types, 84–89
summary, 89–90
views, 88–89
relational division, 312–14
pivoting, 465–66
relations
attributes, 85–87
divisor, dividend, and quotient, 95–97
operators, 90–91
properties of, 75–76
propositions and predicates, 87–88
relational database model, 84–89
universe, 76
virtual, 88–89
relvar, 126
Rename operator, 98
T-SQL support, 103–04
Repartition Streams operator, 228
REPEATTABLE clause, 266
REPLACE function, 433–34
representation, faithful, 49–51
Resource Governor, 171–72
Restrict operator, 91–92
T-SQL support, 103–04
restriction expression, 91–92
Results to Text output mode, 435
reverse logic, 72
relational division problems, 312–14
RID lookup operation, 196
right input, 20–21
APPLY operator, 21–22
RIGHT keyword
outer joins, 397–401
right semi joins, 414–16
Rincon, Eladio, 127
RNBenchmark table, 344–48
Road System example, 666–70
Roads sample table, 666–70
ROLLUP subclass, 506
ROLLUP subclause, 512–14

root pages, 193
roots node, 738–39
ROUND function, 570
row number calculation
  benchmarks, 348–49
  cursors, 341–42
  IDENTITY-based, 342–44
  nonpartitioned, 343
  partitioned, 344
  performance considerations, 344–49
  subqueries, 335–41
row numbers
  benchmarks, 348–49
  calculation. *See* row numbers calculation
  paging, 349–52
row overflow pages, 188
ROW_NUMBER function, 330–52, 433–34
  benchmarks, 348–49
  cross joins, 392–93
  current and previous occurrence matching, 546–47
  median, 495
  TOP n, 542
ROW_OVERFLOW_DATA allocation units, 189
ROWMODCTR, 647
rows
  anchor, 549–50, 592
  copy generation, 26–27
  current and previous occurrence matching, 543–47
  duplicate, 15
  duplicate data removal, 601–03
  foreign key, 106–08
  grouping, 23
  index levels, 193–95
  keys, 105–06
  NULL values removal, 28
  pivoting. *See* pivoting
  random, TOP option, 552–54
  removal, 28
  set operations, 31–32
  size limits, 188–89
  TOP option, 16
  value constructors, 607–08
ROWS keyword, 265
ROWS option, 266
rows_per_leaf_page, 194
rows_per_non_leaf_page, 194
rowsets, 88
RPCCompleted event class, 150
running aggregation, 451–52
Russell, Bertrand, 52
Russell's Paradox, 52, 96, 110–11
Rys, Michael, 487

S

S set, 46
  sample application, 77–80
Sales sample table, 330–31
Sales.MyShippers sample table, 314–16
Sales.Orders sample table, 497
SalesRN CTE, 350
sameness, 39
sample data. *See also* specific sample tables
  grouping sets, 507
  Performance database, 127–31
  preparation, 259–65
  TABLESAMPLE, 265–68
Sarka, Dejan, 44, 122, 757
Scalar operator, 650
scalar subqueries, 297–98
scalar types, 86–87
scale, algorithms, 279–82
SCOPE_IDENTITY function, 629
second normal form, 115–16
Segment operator, 207, 333
SELECT clause, 331
SELECT INTO statement, 216, 563–64
  FULL recovery model, 571–74
  minimal logging, 567–68
  non-FULL recovery mode, 574–75
SELECT list
  aliases, 14–15
  asterisk use, 306
  bushy plans, 414
  column order, 17
  DATEADD function, 392
  derived tables, 319
  DISTINCT clause, 16, 369, 371
  pivoting, 462
  self joins, 404
  unpivot operator, 471
SELECT phase, 5, 14–16
  ORDER BY clause, 29–30
SELECT query, 278
  partition elimination, 649–52
  TOP option, 527–35
SELECT statement, 103–04
  assignments, 611–14
  NOLOCK hint, 216
  showplan, 643–45
SELECT TOP, 528–29
SELECT_INTO statement, 343–44
selection sort, 288
selectivity, 224, 251
  logical reads and, 251–52
  performance statistics and query cost, 253–55
  point determination, 248–49
  vs. logical reads, 252
  vs. query cost, 253–55
selector operators, 86
self joins, 402–04
self-contained subqueries, 297–302
semi joins, 98, 414–16
semicolons, 322
Semijoin operator, 98
SEQUEL, 1
sequence mechanisms
  custom sequences, 596–600
  IDENTITY property, 595–96

Sequence Project operator, 333
sequential access, 287
Serializable attribute, 484
Server Activity collections, 148
Server Activity History report, 139
Server Actual History report, 148
server instance
  partitioned view, 639
  wait analysis, 134–37
Server Management Objects (SMO), 187
Server Management Studio (SMSS)
  cross joins, 396–97
  Discard Results option, 329–44
Sessions sample table, 260–65
Set Default implementation, 107–08
SET FORCEPLAN ON statement, 406
Set Null implementation, 107–08
set operations, 31–32, 435–36
  EXCEPT, 437–39
  INTERSECT, 439–40
  INTO clause, 441
  NULL values, 32
  precedence, 440
  UNION, 436–37
  unsupported logical phrases, circumventing, 441–42
set operators, 31–32, 56–63
set S. *See* S set
SET STATISTICS IO option, 351
set theory, 35, 44. *See also* sets
  domains of discourse, 46–49
  faithfulness, 49–51
  generalizations, 64–65
  multiset theory, 64–65
  nave, 52
  notation, 45–46
  ordered pairs, tuples, and Cartesian products, 53–54
  Russell's Paradox, 52
  set membership operator definition, 44–45
  set U, 46
  empty sets, 54–55
set-based query tuning vs. iterative/procedural approaches, 268–76
sets. *See also* set operations; set operators; set theory
  cardinality, 56–57
  characteristic function, 77–80
  characteristic function definition, 55
  complement, 62
  difference, 63
  empty, 54–55, 315
  enumeration, 45
  membership operator definition, 44–45
  nested. *See* nested sets
  notation, 45–46
  operators. *See* set operators
  order, 57–61
  partitions, 63–64
  set-builder notation, 45–46
  subsets, 61–62
  union and intersection, 62–63
  universe. *See* U set
  well-definedness, 46
shared locks, 219, 257–58
Shippers sample table, 269–76, 566
SHOWPLAN_XML option, 186
SIMPLE recovery model, 575
Simsion, Graeme, 111–12
Singh, Simon, 110
single sequence values, 596–97
SINGLE_BLOB type, 566
SINGLE_CLOB type, 566
SINGLE_NCLOB type, 566
sixth normal form, 122
sliding aggregation, 457–59
sliding total, previous year, example, 417–20
sliding window scenario, 642
SMO (Server Management Objects), 187
SMSS (Server Management Studio). *See* Server Management Studio (SMSS)
Solid Quality Mentors, 127
SOME predicate, 316–18
Sort operator, 286, 509–10
SORT_IN_TEMP_DB option, 257
sorting
  algorithms, 285–86
  external, 287
  grouping sets, 524
  HIERARCHYID data type, 726–30
  insertion and selection, 288
  iteration/recursion, 688–91
  O(n log N) algorithms, 288–89
  quadratic algorithms, 288
  quick sort, 289
  running time comparsions, 285–86
  swapping, 289
  ultra sort, 289
source code, 43–44
sp_autostats, 647
sp_configure, 653
sp_create_plan_guide, 124
sp_get_query_template procedure, 157
sp_updatestats, 142–43
specialization
  database, 124–25
  relational database model, 124–26
specialized solutions, 487–99
  bitwise operations, 490–94
  product, 488–90
  string concatenation, 487–88
spread by element, 464
spreading, PIVOT operator, 24
SPStmtCompleted event class, 150
SQL
  pronunciation origin, 1
  relations, 103
SQL handle, 168

SQL Server 2005
partitioning, 641–45
query plans, parallelism, 654–57
showplan, 649–52
SQL Server 2008
CLR database code, 476–77
constraints, order of enforcement, 110
data collection and Management data warehouse, 187
hash joins, 425–26
hints, 185
partitioning, 639–57
query plans, parallelism, 654–57
showplan, 649–52
Timestamp type, 109–10
tracing, 149–50
triggers support, 109
XML type, 109
SQL Server Magazine, 757
SQL_VARIANT data type, 188–89, 461
UNPIVOT operator, 473
SQLBatchCompleted event class, 150
SQLStmtCompleted event class, 150
SqlUserDefinedAggregate attribute, 484–85
statistics
automatic maintenance, 142–43
cloning, 187
filtered indexes, 239–42
partitioned tables, 645–48
queries, 167–69
statistics cloning, 187
STATISTCS IO, 172–73
STATISTICS IO option, 172–73
STATISTICS TIME option, 173–74
STATISTICS XML option
Storage Engine, 207–23
stored procedures, 109
Stream Aggregate operator, 509–10, 655
string concatenation, 449
aggregate specialized solution, 487–88
pivoting, 475
StringBuilder class, 484
StringConcat function, 487
strings, searching, 289–90
StructLayoutAttribute, 485
STUFF function, 610–11
subgraph/subtree, with path enumeration, 685–88
sublinear complexity, 282
subordinates
direct, 717–18
iteration/recursion, 671–81
subqueries, 297–98
aggregate functions, 14
correlated. *See* correlated subqueries
gaps solution 1, 366–69
gaps solution 2, 369–71
misbehaving, 314–16
multivalued, 297–98
partitioning, 340–41
RANK and DENSE_RANK functions, 352–54
row number calculation, 335–41
scalar, 297–98
self joins, 404
self-contained, 297–302
table-valued, 297–98
uncommon predicates, 316–18
subsequences, increasing, 291
Subset Of operator, 90–91
subsets, 61–62
SUBSTRING function, 214, 431
mode, 499
subtrees
cost, 178
moving, 697–99, 712–14
querying, 715–17
removal, 700–01
subtypes, 124
SUM aggregate, 453–57
SUM function, 489–90
SUM(qty) function, 468
superexponential complexity, 134–35
Superset Of operator, 90–91
supertypes, 124
surrogate keys, 106
swapping algorithms, 289
SWITCH command, 645–46
SWITCH OUT command, 647
symmetric relation properties, 75–76
syntax, joins, 389–90
sys.assemblies, 486
sys.assembly_modules, 486
sys.dm_db_missing_index_columns, 232
sys.dm_db_missing_index_details, 232
sys.dm_db_missing_index_group_stats, 232
sys.dm_db_missing_index_groups, 232
sys.dm_exec_cached_plans, 171
sys.dm_exec_plan_attributes, 171
sys.dm_exec_query_plan, 168, 171
sys.dm_exec_query_stats, 167–69
sys.dm_exec_sql_text, 168, 171
sys.dm_io_virtual file_stats, 145–48
sys.dm_os_performance_counters, 143–44
sys.dm_os_wait_stats, 134–37
sys.syscacheobjects, 171
sys.system_internals_allocation_units, 189–90
SYSDATETIME function, 173–74
SYSTEM keyword, 265
SYSTEM method, 265–66
system types, 87
System.Object class, 484

T

table expressions, 318
  common (CTEs), 321–30. *See also* common table expressions (CTEs)
  derived tables, 318–21
  interchangeability, 89
  left and right input, 20–21
  ORDER BY clause, 18–20
  TOP option, 18–20
table hints, 185–86
table operators, 20–28. *See also* specific operators
  processing order, 11
table scan, 198–201, 245, 557
Table Scan operator, 643–45
Table Spool operator, 263
tables
  aliases, 606
  auxiliary table of numbers, 359–62
  clustered, nonclustered indexes, 196–97
  constraints, 109–10
  derived. *See* derived tables
  foreign key, 106–08
  heaps, 189–91
  joins. *See* table joins
  key, 105–06
  metadata query, 648
  normalization. *See* normal forms; normalization
  organization, 189
  parent and child relations, 106–08
  partitioned. *See* partitioned tables
  partitioning, 258–59
  pivoting. *See* pivoting
  preserved, 11
TABLESAMPLE, 265–68
table-valued subqueries, 297–98
TABLOCK hint, 211–12, 215, 566, 568
  INSERT statement heap, B-tree, TF-610, key range scenarios, 575–89
  minimal logging summary, 590
Talmage, Ron, 757
Tchernitsky, Nicolay, 757
TClose operator, 99–100
temp db database, 137, 148
temporal data, 122
TempSeq table, 364–65
Terminate method, 482–83
testing, insert scenarios, 571–89
TF-610, INSERT SELECT statement scenarios, 579–89
theta joins, 94
third normal form, 116–17
three-valued logic, 9, 74
tiebreaker, 302–06
  aggregation, 448–51
  determinism, 334
  median, 496
  mode, 498–99
  nonunique sort column method, 337–38
  TOP option, 529–30
tile number functions, 354–59
Tiles CTE, 495
TOP n for each group, 537–43
Top operator, 207, 308
TOP option, 16, 527
  determinism, 529–30
  input expressions, 530–31
  matching current and previous occurrences, 543–47
  median, 554–56
  modifications, 531–33
  on steroids, 534–35
  paging, 547–52
  random rows, 552–54
  table expressions, 18–20
  TOP n for each group, 537–43
TOP PERCENT option, 554
TOP phase, 5
ToString method, 484, 711
total order, 59
TPC-H benchmark, 641–45
tracing, 149–50, 186
  data analysis, 155–67
  performance workload, 150–55
transaction log, wait analysis, 136, 148
transactions, 105
Transact-SQL. *See* T-SQL
transitive closure, 99–100, 740
  directed acyclic graphs, 740–45
  undirected cyclic graphs, 745–54
transitive relation properties, 75–76
translation, English to mathematics, 35–44
tree diagrams, 99–100
trees, 660–61. *See also* subtrees
  left and right values assignment, 731–36
trend identification, 291
trend marker practical application, 290–92
trichotomy, 58–59
triggers, 109
  denormalization, 123–24
  MERGE statement, 627–28
TRUE values, 9
true/false expressions. *See* Boolean expressions
TRUNCATE TABLE statement, 600–01
truth value, 68
T-SQL, 1–2
  Boolean expressions, 67
  cycle detection, 691
  HIERARCHYID data type, 707
  joins logical processing order, 409
  joins, nonsupported, 401
  LISLP problem solution, 292–95
  MAX attribute, 449
  relational algebra and calculus support, 103–04
  semicolon termination, 322
  statement assignments, 611
  UPDATE syntax, 596
  vs. CLR, function implementation, 159

tuples, 53–54
  attributes, 84
  calculus, 102–03
  header, 103
  heading, 84
  operators, 90–91
  properties, 84
  relational database nodel, 84–89
  sets. *See* relations
two-valued logic, 623
types
  atomic, 86
  constraints, 109
  defined, 85
  encapsulated, 86
  operators, 86, 90–91
  relational database model, 84–89
  scalar vs. nonscalar, 86–87
  subtypes and supertypes, 124
  system, 87
  user-defined, 87
  vs. domains, 84

U

U set, 46
  empty sets, 54–55
UDAs (user-defined aggregates), 476–82
ultra sort, 289
undefined terms, 39
undirected cyclic graphs, 745–54
undirected cyclic weighted graphs, 670
undirected graphs, 659–60
UNION ALL operation, 437
UNION ALL operator, 31–32
UNION DISTINCT operation, 437
UNION operation, 31–32, 435–37
Union operator, 92–93
  T-SQL support, 103–04
union, set, 62–63
UNIQUE constraint
  NULL values, 241–42
unique sort column method, 335–37
UNIQUEIDENTIFIER value, 600
uniquifier, 191, 196
UNKNOWN values, 9, 74
  EXISTS predicate, 305–06
  IN predicate, 307
Unpack operator, 100–01
UNPIVOT operator, 24–28, 471–73
  phases, 25–28
unpivoting, 470–73
UPDATE statement, 103–04
  assignments, 614–16
  joins, 606–10
  MERGE statement, 617–21
  OUTPUT clause, 632–34
  TOP option, 531–33
UPDATE STATISTICS command, 645, 647
UPDATE trigger, 627–28
updating data, 606–16
updating, partitioned views, 639–40
UPDLOCK hint, 710
USE PLAN hint, 122–24
user-defined aggregates (UDAs), 476–82
user-defined functions
  auxiliary table of numbers, 362
  inline, CTEs, 325–26
user-defined types, 87
USING clause
  MERGE statement, 618

V

vacuous truths, 71–72
values, 43
VALUES clause, 472–73, 561–84
VARBINARY data type, 188–89
VARBINARY(MAX) data type, 189
  updating, 610–11
VARCHAR data type, 188–89
VARCHAR(MAX) data type, 189, 484, 688
  updating, 610–11
variables
  functional dependencies, 112
  types, 86
vertices, 99–100
views, 88–89
  common table expressions, 325–26
  compatibility, 171
  indexed, 242–44
  updatable, 109
Visual Studio 2008, assembly creation and deployment, 482–87

W

wait analysis
  instance level, 134–37
  top wait isolation, 137–38
  wait information collection, 139–43
weighted graphs, 666
well-definedness, 37–38
  sets, 46
WHEN clause, 623–24
WHEN MATCHED clause, 624
  MERGE statement, 621
  multiple, 623–24
WHEN MATCHED THEN clause, 618–20
WHEN NOT MATCHED BY SOURCE clause, 624
WHEN NOT MATCHED clause
  MERGE statement, 623, 624
WHEN NOT MATCHED THEN clause
  MERGE statement, 618–20

WHERE clause
  inner joins, 395–97
  outer joins, 399–401
WHERE filter, 399. *See also* WHERE phase
  OUTER JOIN clause, 12
WHERE phase, 5, 11–12
whichth number, 60–61
Wiernik, Adolfo, 122
window-based calculations, 29, 445
WITH clause, table hints, 185
WITH CUBE option, 506, 511–12
WITH keyword, CTEs, 322
WITH ROLLUP option, 506, 514
WITH statement, multiple CTEs, 323
WITH TIES option, 16
  TOP option, 530
Witt, Graham, 111–12
WRITE method, 484, 610–11
WRITELOG wait, 136

X

XML
  showplans, 185–86
  triggers and validations, 109

Y

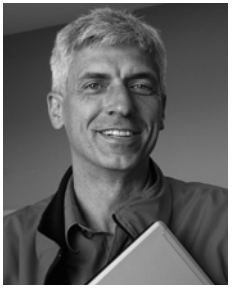
YEAR(orderdate), 468
year-to-date aggregation, 459–60

About the Authors

Itzik Ben-Gan

Itzik Ben-Gan is a mentor and cofounder of Solid Quality Mentors. An SQL Server Microsoft MVP (Most Valuable Professional) since 1999, Itzik has delivered numerous training events around the world focused on T-SQL querying, query tuning, and programming. Itzik is the author of several books about T-SQL. He has written many articles for *SQL Server Magazine* as well as articles and white papers for MSDN. Itzik's speaking engagements include Tech Ed, DevWeek, PASS, SQL Server Magazine Connections, various user groups around the world, and Solid Quality Mentors events.

Lubor Kollar



Lubor Kollar is Group Program Manager in Microsoft Corp. He has been working in SQL Server development organization since 1996. Prior to joining Microsoft, he was developing various DB2 engines at IBM. Currently Lubor is leading SQL Server Customer Advisory Team (SQL CAT) working on the most challenging SQL Server deployments around the world. SQL CAT is responsible for maintaining tight connections between the users and creators of new SQL Server releases. Another goal of SQL CAT is to spread the wisdom learned from the most advanced SQL Server deployments. One of the major channels easily accessible to the widest audience is the *www.sqlcat.com* Web site.

Dejan Sarka

Dejan Sarka focuses on development of database and business intelligence applications. Besides projects, he spends about half of his time on training and mentoring. He is a frequent speaker at some of the most important international conferences, including PASS, TechEd, and SqlDevCon. He is also indispensable at regional Microsoft events—for example, the NT Conference (the biggest Microsoft conference in Central and Eastern Europe). He is the founder of the Slovenian SQL Server and .NET Users Group. Dejan is the main author, coauthor, or guest author of seven books about databases and SQL Server. Dejan also developed two courses for Solid Quality Learning: Data Modeling Essentials and Data Mining with SQL Server 2008.

Steve Kass

Steve Kass holds a Ph.D. in mathematics from the University of Wisconsin, and he is a professor of Mathematics and Computer Science at Drew University, where he has taught since 1988. An SQL Server Microsoft MVP since 2002, he has written for *SQL Server Magazine* and spoken at SQL Server Magazine Connections events and to user groups in the New York City area. Steve's mathematical work has appeared in *Complex Systems* and the *Journal of Algebra*.