Microsoft

Foreword by Tobias Ternström
Lead Program Manager, Microsoft SQL Server Engine team

Microsoft® SQL Server® 2012 High-Performance T-SQL Using Window Functions

Itzik Ben-Gan
SolidQ

Microsoft® SQL Server® 2012 High-Performance T-SQL Using Window Functions

Apply powerful window functions in T-SQL—and increase the performance and speed of your queries

Optimize your queries—and obtain simple and elegant solutions to a variety of problems—using window functions in Transact-SQL. Led by T-SQL expert Itzik Ben-Gan, you'll learn how to apply calculations against sets of rows in a flexible, clear, and efficient manner. Ideal whether you're a database administrator or developer, this practical guide demonstrates how to use a range of T-SQL querying solutions to address common business tasks.

Discover how to:

- Go beyond traditional query approaches to express set calculations more efficiently
- Delve into ordered set functions such as rank, distribution, and offset
- Implement hypothetical set and inverse distribution functions in standard SQL
- Use strategies to improve sequencing, paging, filtering, and pivoting
- Increase query speed using partitioning, ordering, and coverage indexing
- Apply new optimization iterators such as Window Spool
- Handle common issues such as running totals, intervals, medians, and gaps

Get code samples on the web
Ready to download at
http://go.microsoft.com/FWLink/?Linkid=246708

*For **system requirements**, see the Introduction.*

ISBN: 978-0-7356-5836-3

microsoft.com/mspress

U.S.A. $36.99
Canada $38.99
[*Recommended*]

Databases/Microsoft SQL Server

About the Author

Itzik Ben-Gan, a Microsoft MVP for SQL Server since 1999, is cofounder of SolidQ, a company that provides consulting and training services for the entire Microsoft data platform. He writes numerous articles for *SQL Server Pro* magazine and speaks at industry events such as the Professional Association for SQL Server (PASS) and Microsoft TechEd.

DEVELOPER ROADMAP

Start Here!
- Beginner-level instruction
- Easy to follow explanations and examples
- Exercises to build your first projects

Step by Step
- For experienced developers learning a new topic
- Focus on fundamental techniques and tools
- Hands-on tutorial with practice files plus eBook

Developer Reference
- Professional developers; intermediate to advanced
- Expertly covers essential topics and techniques
- Features extensive, adaptable code examples

Focused Topics
- For programmers who develop complex or advanced solutions
- Specialized topics; narrow focus; deep coverage
- Features extensive, adaptable code examples

Microsoft®

Microsoft®

Microsoft® SQL Server® 2012 High-Performance T-SQL Using Window Functions

Itzik Ben-Gan

Copyright © 2012 by Itzik Ben-Gan
All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

ISBN: 978-0-7356-5836-3

1 2 3 4 5 6 7 8 9 LSI 7 6 5 4 3 2

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at *http://www.microsoft.com/learning/booksurvey*.

Microsoft and the trademarks listed at *http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx* are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

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

Acquisitions and Developmental Editor: Ken Jones

Production Editor: Kristen Borg

Production Services: Curtis Philips

Technical Reviewer: Adam Machanic

Copyeditor: Roger LeBlanc

Indexer: Lucie Haskins

Cover Design: Twist Creative • Seattle

Cover Composition: Karen Montgomery

Illustrators: Robert Romano and Rebecca Demarest

To the Quartet.

—Q1

Contents at a Glance

	Foreword	*xi*
	Introduction	*xiii*
CHAPTER 1	**SQL Windowing**	**1**
CHAPTER 2	**A Detailed Look at Window Functions**	**33**
CHAPTER 3	**Ordered Set Functions**	**81**
CHAPTER 4	**Optimization of Window Functions**	**101**
CHAPTER 5	**T-SQL Solutions Using Window Functions**	**133**
	Index	*211*

Contents

Foreword . . . . . *xi*
Introduction . . . . . *xiii*

Chapter 1 SQL Windowing 1

Background of Window Functions 2
Window Functions Described 2
Set-Based vs. Iterative/Cursor Programming 6
Drawbacks of Alternatives to Window Functions 11
A Glimpse of Solutions Using Window Functions 15
Elements of Window Functions 19
Partitioning 20
Ordering 21
Framing 22
Query Elements Supporting Window Functions 23
Logical Query Processing 23
Clauses Supporting Window Functions 25
Circumventing the Limitations 28
Potential for Additional Filters 30
Reuse of Window Definitions 31
Summary 32

Chapter 2 A Detailed Look at Window Functions 33

Window Aggregate Functions 33
Window Aggregate Functions Described 33
Supported Windowing Elements 34

What do you think of this book? We want to hear from you!
Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Further Filtering Ideas 49
Distinct Aggregates 51
Nested Aggregates 53
Ranking Functions 57
Supported Windowing Elements 58
ROW_NUMBER 58
NTILE 63
RANK and DENSE_RANK 66
Distribution Functions 68
Supported Windowing Elements 68
Rank Distribution Functions 68
Inverse Distribution Functions 71
Offset Functions 74
Supported Windowing Elements 74
LAG and LEAD 74
FIRST_VALUE, LAST_VALUE, and NTH_VALUE 76
Summary 79

Chapter 3 Ordered Set Functions 81

Hypothetical Set Functions 82
RANK 82
DENSE_RANK 84
PERCENT_RANK 85
CUME_DIST 86
General Solution 87
Inverse Distribution Functions 90
Offset Functions 94
String Concatenation 98
Summary 100

Chapter 4 Optimization of Window Functions 101

Sample Data 101

Indexing Guidelines 103

POC Index 104

Backward Scans 105

Columnstore Indexes 108

Ranking Functions 108

ROW_NUMBER 109

NTILE 110

RANK and DENSE_RANK 111

Improved Parallelism with APPLY 112

Aggregate and Offset Functions 116

Without Ordering and Framing 116

With Ordering and Framing 119

Distribution Functions 128

Rank Distribution Functions 128

Inverse Distribution Functions 129

Summary 132

Chapter 5 T-SQL Solutions Using Window Functions 133

Virtual Auxiliary Table of Numbers 133

Sequences of Date and Time Values 137

Sequences of Keys 138

Update a Column with Unique Values 138

Applying a Range of Sequence Values 139

Paging 143

Removing Duplicates 145

Pivoting 148

TOP N per Group 151

Mode 154

Running Totals 158
Set-Based Solution Using Window Functions 160
Set-Based Solutions Using Subqueries or Joins 161
Cursor-Based Solution 162
CLR-Based Solution 164
Nested Iterations 166
Multirow UPDATE with Variables 167
Performance Benchmark 169
Max Concurrent Intervals 171
Traditional Set-Based Solution 173
Cursor-Based Solution 175
Solutions Based on Window Functions 178
Performance Benchmark 180
Packing Intervals 181
Traditional Set-Based Solution 183
Solutions Based on Window Functions 184
Gaps and Islands 193
Gaps 194
Islands 195
Median 202
Conditional Aggregate 204
Sorting Hierarchies 206
Summary 210

Index *211*

What do you think of this book? We want to hear from you!
Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Foreword

SQL is a very interesting programming language. When meeting with customers, I am constantly reminded of the language's dual nature with regard to complexity. Many people getting started with SQL see it as a simple programming language that supports four basic verbs: SELECT, INSERT, UPDATE, and DELETE. Some people never get much further than this. Maybe a few more figure out how to filter rows in a query using the WHERE clause and perhaps do the occasional JOIN. However, those who spend more time with SQL and learn about its declarative, relational, and set-based model will find a rich programming language that keeps you coming back for more.

One of the most fundamental additions to the SQL language, back in Microsoft SQL Server 2005, was the introduction of window functions with syntactic constructs such as the OVER clause and a new set of functions known as ranking functions (ROW_NUMBER, RANK, and so on). This addition enabled solving common problems in an easier, more intuitive, and often better-performing way than what was previously possible. A few years later, the single most-requested language feature was for Microsoft to extend its support for window functions—with a set of new functions and, more importantly, with the concept of frames. As a result of these requests from a wide range of customers, Microsoft decided to continue investing in window functions extensions in SQL Server 2012.

Today, when I talk to customers about new language functionality in SQL Server 2012, I always recommend they spend extra time with the new window functions and really understand the new dimension that this brings to the SQL language. I am happy that you are reading this book and thus taking what I am sure is precious time to learn how to use this rich functionality. I am confident that the combination of using SQL Server 2012 and reading this book will help you become an even more efficient SQL Server user, and help you solve both simple as well as complex problems significantly faster than before.

Enjoy!

Tobias Ternström

Lead Program Manager,
Microsoft SQL Server Engine team

Introduction

Window functions, to me, are the most profound feature supported by both standard SQL and Microsoft SQL Server's dialect—T-SQL. They allow you to perform calculations against sets of rows in a flexible, clear, and efficient manner. The design of window functions is ingenious, overcoming a number of shortcomings of the traditional alternatives. The range of problems that window functions help solve is so wide that it is well worth investing your time in learning those. SQL Server 2005 was the version in which window functions were introduced initially. SQL Server 2012 then added more complete support by enhancing some of the existing functions, as well as adding new ones. This book covers both the SQL Server–specific support for window functions, as well as standard SQL's support, including elements that were not yet implemented in SQL Server.

Who Should Read This Book

This book is intended for SQL Server developers and database administrators (DBAs); those who need to write queries and develop code using T-SQL. The book assumes that you already have at least half a year to a year of experience writing and tuning T-SQL queries.

Organization of This Book

The book covers both the logical aspects of window functions as well as their optimization and practical usage aspects. The logical aspects are covered in the first three chapters. The first chapter explains SQL windowing concepts, the second provides a breakdown of window functions, and the third covers ordered set functions. The fourth chapter covers optimization of window functions in SQL Server 2012. Finally, the fifth and last chapter covers practical uses of window functions.

Chapter 1, "SQL Windowing," covers standard SQL windowing concepts. It describes the design of window functions, the types of window functions, and the elements involved in a window specification, such as partitioning, ordering, and framing.

Chapter 2, "A Detailed Look at Window Functions," gets into the details and specifics of the different window functions. It describes window aggregate functions, window ranking functions, window offset functions, and window distribution functions.

Chapter 3, "Ordered Set Functions," describes the support standard SQL has for ordered set functions, including hypothetical set functions, inverse distribution functions, and others. The chapter also explains how to achieve similar calculations in SQL Server.

Chapter 4, "Optimization of Window Functions," covers in detail the optimization of window functions in SQL Server 2012. It provides indexing guidelines for optimal performance, explains how parallelism is handled and how to improve it, discusses the new Window Spool iterator, and more.

Chapter 5, "T-SQL Solutions Using Window Functions," covers practical uses of window functions to address common business tasks.

System Requirements

Window functions are part of the core database engine of Microsoft SQL Server 2012; hence, all editions of the product support this feature. To run the code samples in this book, you need access to an instance of the SQL Server 2012 database engine (any edition), and you need to have the sample database installed. If you don't have access to an existing instance, Microsoft provides trial versions. You can find details at: *http://www.microsoft.com/sql*. For hardware and software requirements, please consult SQL Server Books Online at: *http://msdn.microsoft.com/en-us/library/ms143506(v=sql.110).aspx*.

Code Samples

This book features a companion website that makes available to you all the code used in the book, sample data, the errata, additional resources, and more, at the following page:

http://www.insidetsql.com

In this website, go to the Books section and select the main page for the book in question. The book's page has a link to download a compressed file with the book's source code, including a file called TSQL2012.sql that creates and populates the book's sample database, TSQL2012.

Acknowledgments

A number of people contributed to making this book a reality, whether directly or indirectly, and deserve thanks and recognition.

To Lilach, for giving reason to everything I do, for tolerating me, and for helping review the text.

To my parents, Mila and Gabi, and to my siblings, Mickey and Ina, for the constant support and for accepting the fact that I'm away.

To members of the Microsoft SQL Server development team: Tobias Ternström, Lubor Kollar, Umachandar Jayachandran, Marc Friedman, Milan Stojic, and I'm sure many others. I know it wasn't a trivial effort to add support for window functions in SQL Server. Thanks for the great effort, and thanks for all the time you spent meeting with me and responding to my emails, addressing my questions, and answering my requests for clarification.

To the editorial team at O'Reilly and MSPress. Ken Jones, you spent the most Itzik hours of all, and it's a real pleasure working with you. Also thanks to Ben Ryan, Kristen Borg, Curtis Philips, and Roger LeBlanc.

To Adam Machanic. Thanks for agreeing to be the technical editor of the book. There aren't many people who understand SQL Server development as well as you do. You were the natural choice for me to fill this role for this book.

To "Q2," "Q3," and "Q4." It's great to be able to share ideas with people who understand SQL as well as you do, and are such good friends and take life lightly. I feel that I can share everything with you without worrying about any boundaries or consequences. Thanks for your early review of the text.

To SolidQ, my company for the last decade. It's gratifying to be part of such a great company that evolved to what it is today. The members of this company are much more than colleagues to me; they are partners, friends, and family. Thanks to Fernando G. Guerrero, Douglas McDowell, Herbert Albert, Dejan Sarka, Gianluca Hotz, Jeanne Reeves, Glenn McCoin, Fritz Lechnitz, Eric Van Soldt, Joelle Budd, Jan Taylor, Marilyn Templeton, Berry Walker, Alberto Martin, Lorena Jimenez, Ron Talmage, Andy Kelly, Rushabh Mehta, Eladio Rincón, Erik Veerman, Johan Richard Waymire, Carl Rabeler, Chris Randall, Åhlén, Raoul Illyés, Peter Larsson, Peter Myers, Paul Turley, and so many others.

To members of the *SQL Server Pro* editorial team: Megan Keller, Lavon Peters, Michele Crockett, Mike Otey, and I'm sure many others. I've been writing for the

magazine for over a decade and am grateful for the opportunity to share my knowledge with the magazine's readers.

To SQL Server MVPs—Alejandro Mesa, Erland Sommarskog, Aaron Bertrand, Paul White, and many others—and to the MVP lead, Simon Tien. This is a great program that I'm grateful and proud to be part of. The level of expertise of this group is amazing, and I'm always excited when we all get to meet, both to share ideas and just to catch up at a personal level over beer. I believe that, in great part, Microsoft's decision to provide more complete support for window functions in SQL Server 2012 is thanks to the efforts of SQL Server MVPs and, more generally, the SQL Server community. It is great to see this synergy yielding such meaningful and important results.

Finally, to my students: teaching SQL is what drives me. It's my passion. Thanks for allowing me to fulfill my calling, and for all the great questions that make me seek more knowledge.

Errata & Book Support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site:

http://www.microsoftpressstore.com/title/ 9780735658363

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

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

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

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

itzik@SolidQ.com

Stay in Touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*

CHAPTER 1

SQL Windowing

Window functions are functions applied to sets of rows defined by a clause called OVER. They are used mainly for analytical purposes allowing you to calculate running totals, calculate moving averages, identify gaps and islands in your data, and perform many other computations. These functions are based on an amazingly profound concept in standard SQL (which is both an ISO and ANSI standard)—the concept of *windowing*. The idea behind this concept is to allow you to apply various calculations to a set, or *window*, of rows and return a single value. Window functions can help to solve a wide variety of querying tasks by helping you express set calculations more easily, intuitively, and efficiently than ever before.

There are two major milestones in Microsoft SQL Server support for the standard window functions: SQL Server 2005 introduced partial support for the standard functionality, and SQL Server 2012 added more. There's still some standard functionality missing, but with the enhancements added in SQL Server 2012, the support is quite extensive. In this book, I cover both the functionality SQL Server implements as well as standard functionality that is still missing. Whenever I describe a feature for the first time in the book, I also mention whether it is supported in SQL Server, and if it is, in which version of the product it was added.

From the time SQL Server 2005 first introduced support for window functions, I found myself using those functions more and more to improve my solutions. I keep replacing older solutions that rely on more classic, traditional language constructs with the newer window functions. And the results I'm getting are usually simpler and more efficient. This happens to such an extent that the majority of my querying solutions nowadays make use of window functions. Also, standard SQL and relational database management systems (RDBMSs) in general are moving toward analytical solutions, and window functions are an important part of this trend. Therefore, I feel that window functions are the future in terms of SQL querying solutions, and that the time you take to learn them is time well spent.

This book provides extensive coverage of window functions, their optimization, and querying solutions implementing them. This chapter starts by explaining the concept. It provides the background of window functions, a glimpse of solutions using them, coverage of the elements involved in window specifications, an account of the query elements supporting window functions, and a description of the standard's solution for reusing window definitions.

Background of Window Functions

Before you learn the specifics of window functions, it can be helpful to understand the context and background of those functions. This section provides such background. It explains the difference between set-based and cursor/iterative approaches to addressing querying tasks and how window functions bridge the gap between the two. Finally, this section explains the drawbacks of alternatives to window functions and why window functions are often a better choice than the alternatives. Note that although window functions can solve many problems very efficiently, there are cases where there are better alternatives. Chapter 4, "Optimization of Window Functions," goes into details about optimizing window functions, explaining when you get optimal treatment of the computations and when treatment is nonoptimal.

Window Functions Described

A window function is a function applied to a set of rows. A *window* is the term standard SQL uses to describe the context for the function to operate in. SQL uses a clause called OVER in which you provide the window specification. Consider the following query as an example:

See Also *See the book's Introduction for information about the sample database TSQL2012 and companion content.*

```
USE TSQL2012;

SELECT orderid, orderdate, val,
  RANK() OVER(ORDER BY val DESC) AS rnk
FROM Sales.OrderValues
ORDER BY rnk;
```

Here's abbreviated output for this query:

```
orderid  orderdate               val       rnk
-------- ----------------------- --------- ---
10865    2008-02-02 00:00:00.000 16387.50  1
10981    2008-03-27 00:00:00.000 15810.00  2
11030    2008-04-17 00:00:00.000 12615.05  3
10889    2008-02-16 00:00:00.000 11380.00  4
10417    2007-01-16 00:00:00.000 11188.40  5
10817    2008-01-06 00:00:00.000 10952.85  6
10897    2008-02-19 00:00:00.000 10835.24  7
10479    2007-03-19 00:00:00.000 10495.60  8
10540    2007-05-19 00:00:00.000 10191.70  9
10691    2007-10-03 00:00:00.000 10164.80  10
...
```

The OVER clause is where you provide the window specification that defines the exact set of rows that the current row relates to, the ordering specification, if relevant, and other elements. Absent any elements that restrict the set of rows in the window—as is the case in this example—the set of rows in the window is the final result set of the query.

Note More precisely, the window is the set of rows, or relation, given as input to the logical query processing phase where the window function appears. But this explanation probably doesn't make much sense yet. So to keep things simple, for now I'll just refer to the final result set of the query, and I'll provide the more precise explanation later.

For ranking purposes, ordering is naturally required. In this example, it is based on the column *val* ranked in descending order.

The function used in this example is RANK. This function calculates the rank of the current row with respect to a specific set of rows and a sort order. When using descending order in the ordering specification—as in this case—the rank of a given row is computed as one more than the number of rows in the relevant set that have a greater ordering value than the current row. So pick a row in the output of the sample query—say, the one that got rank 5. This rank was computed as 5 because based on the indicated ordering (by *val* descending), there are 4 rows in the final result set of the query that have a greater value in the *val* attribute than the current value (11188.40), and the rank is that number plus 1.

What's most important to note is that conceptually the OVER clause defines a window for the function with respect to the current row. And this is true for all rows in the result set of the query. In other words, with respect to each row, the OVER clause defines a window independent of the other rows. This idea is really profound and takes some getting used to. Once you get this, you get closer to a true understanding of the windowing concept, its magnitude, and its depth. If this doesn't mean much to you yet, don't worry about it for now—I wanted to throw it out there to plant the seed.

The first time standard SQL introduced support for window functions was in an extension document to SQL:1999 that covered, what they called "OLAP functions" back then. Since then, the revisions to the standard continued to enhance support for window functions. So far the revisions have been SQL:2003, SQL:2008, and SQL:2011. The latest SQL standard has very rich and extensive coverage of window functions, showing the standard committee's belief in the concept, and the trend seems to be to keep enhancing the standard's support with more window functions and more functionality.

Note You can purchase the standards documents from ISO or ANSI. For example, from the following URL, you can purchase from ANSI the foundation document of the SQL:2011 standard, which covers the language constructs: *http://webstore.ansi.org/RecordDetail.aspx?sku=ISO%2fIEC+9075-2%3a2011*.

Standard SQL supports several types of window functions: aggregate, ranking, distribution, and offset. But remember that windowing is a concept; therefore, we might see new types emerging in future revisions of the standard.

Aggregate window functions are the all-familiar aggregate functions you already know—like SUM, COUNT, MIN, MAX, and others—though traditionally, you're probably used to using them in the context of grouped queries. An aggregate function needs to operate on a set, be it a set defined by

a grouped query or a window specification. SQL Server 2005 introduced partial support for window aggregate functions, and SQL Server 2012 added more functionality.

Ranking functions are RANK, DENSE_RANK, ROW_NUMBER, and NTILE. The standard actually puts the first two and the last two in different categories, and I'll explain why later. I prefer to put all four functions in the same category for simplicity, just like the official SQL Server documentation does. SQL Server 2005 introduced these four ranking functions, with already complete functionality.

Distribution functions are PERCENT_RANK, CUME_DIST, PERCENTILE_CONT, and PERCENTILE_DISC. SQL Server 2012 introduces support for these four functions.

Offset functions are LAG, LEAD, FIRST_VALUE, LAST_VALUE, and NTH_VALUE. SQL Server 2012 introduces support for the first four. There's no support for the NTH_VALUE function yet in SQL Server as of SQL Server 2012.

Chapter 2, "A Detailed Look at Window Functions," provides the meaning, the purpose, and details about the different functions.

With every new idea, device, and tool—even if the tool is better and simpler to use and implement than what you're used to—typically, there's a barrier. New stuff often seems hard. So if window functions are new to you and you're looking for motivation to justify making the investment in learning about them and making the leap to using them, here are a few things I can mention from my experience:

- Window functions help address a wide variety of querying tasks. I can't emphasize this enough. As mentioned, nowadays I use window functions in most of my query solutions. After you've had a chance to learn about the concept and the optimization of the functions, the last chapter in the book (Chapter 5) shows some practical applications of window functions. But just to give you a sense of how they are used, querying tasks that can be solved with window functions include:
  - Paging
  - De-duplicating data
  - Returning top *n* rows per group
  - Computing running totals
  - Performing operations on intervals such as packing intervals, and calculating the maximum number of concurrent sessions
  - Identifying gaps and islands
  - Computing percentiles
  - Computing the mode of the distribution
  - Sorting hierarchies
  - Pivoting
  - Computing recency

- I've been writing SQL queries for close to two decades and have been using window functions extensively for several years now. I can say that even though it took a bit of getting used to the concept of windowing, today I find window functions both simpler and more intuitive in many cases than alternative methods.
- Window functions lend themselves to good optimization. You'll see exactly why this is so in later chapters.

Declarative Language and Optimization

You might wonder why in a declarative language such as SQL, where you logically just declare your request as opposed to describing how to achieve it, two different forms of the same request—say, one with window functions and the other without—can get different performance? Why is it that an implementation of SQL such as SQL Server, with its T-SQL dialect, doesn't always figure out that the two forms really represent the same thing, and hence produce the same query execution plan for both?

There are several reasons for this. For one, SQL Server's optimizer is not perfect. I don't want to sound unappreciative—SQL Server's optimizer is truly a marvel when you think of what this software component can achieve. But it's a fact that it doesn't have all possible optimization rules encoded within it. Two, the optimizer has to limit the amount of time spent on optimization; otherwise, it could spend a much longer time optimizing a query than the amount of time the optimization shaves off from the run time of the query. The situation could be as absurd as producing a plan in a matter of several dozen milliseconds without going over all possible plans and getting a run time of only seconds, but producing all possible plans in hopes of shaving off a couple of seconds might take a year or even several. You can see that, for practical reasons, the optimizer needs to limit the time spent on optimization. Based on factors like the sizes of the tables involved in the query, SQL Server calculates two values: one is a cost considered *good enough* for the query, and the other is the maximum amount of time to spend on optimization before stopping. If either threshold is reached, optimization stops, and SQL Server uses the best plan found at that point.

The design of window functions, which we will get to later, often lends itself to better optimization than alternative methods of achieving the same thing.

What's important to understand from all this is that you need to make a conscious effort to make the switch to using SQL windowing because it's a new idea, and as such it takes some getting used to. But once the switch is made, SQL windowing is simple and intuitive to use; think of any gadget you can't live without today and how it seemed like a difficult thing to learn at first.

Set-Based vs. Iterative/Cursor Programming

People often characterize T-SQL solutions to querying tasks as either set-based or iterative/cursor-based solutions. The general consensus among T-SQL developers is to try and stick to the former approach, but still, there's wide use of the latter. There are several interesting questions here. Why is the set-based approach the recommended one? And if it is the recommended one, why do so many developers use the iterative approach? What are the obstacles that prevent people from adopting the recommended approach?

To get to the bottom of this, one first needs to understand the foundations of T-SQL, and what the set-based approach truly is. When you do, you realize that the set-based approach is nonintuitive for most people, whereas the iterative approach is. It's just the way our brains are programmed, and I will try to clarify this shortly. The gap between iterative and set-based thinking is quite big. The gap can be closed, though it certainly isn't easy to do so. And this is where window functions can play an important role; I find them to be a great tool that can help bridge the gap between the two approaches and allow a more gradual transition to set-based thinking.

So first, I'll explain what the set-based approach to addressing T-SQL querying tasks is. T-SQL is a dialect of standard SQL (both ISO and ANSI standards). SQL is based (or attempts to be based) on the relational model, which is a mathematical model for data management formulated and proposed initially by E. F. Codd in the late 1960s. The relational model is based on two mathematical foundations: set-theory and predicate logic. Many aspects of computing were developed based on intuition, and they keep changing very rapidly—to a degree that sometimes makes you feel that you're chasing your tail. The relational model is an island in this world of computing because it is based on much stronger foundations—mathematics. Some think of mathematics as the ultimate truth. Being based on such strong mathematical foundations, the relational model is very sound and stable. It keeps evolving, but not as fast as many other aspects of computing. For several decades now, the relational model has held strong, and it's still the basis for the leading database platforms—what we call *relational database management systems (RDBMSs)*.

SQL is an attempt to create a language based on the relational model. SQL is not perfect and actually deviates from the relational model in a number of ways, but at the same time it provides enough tools that, if you understand the relational model, you can use SQL relationally. It is doubtless the leading, de facto language used by today's RDBMSs.

However, as mentioned, thinking in a relational way is not intuitive for many. Part of what makes it hard for people to think in relational terms is the key differences between the iterative and set-based approaches. It is especially difficult for people who have a procedural programming background, where interaction with data in files is handled in an iterative way, as the following pseudocode demonstrates:

```
open file
fetch first record
while not end of file
begin
  process record
  fetch next record
end
```

Data in files (or, more precisely, in indexed sequential access method, or ISAM, files) is stored in a specific order. And you are guaranteed to fetch the records from the file in that order. Also, you fetch the records one at a time. So your mind is programmed to think of data in such terms: ordered, and manipulated one record at a time. This is similar to cursor manipulation in T-SQL; hence, for developers with a procedural programming background, using cursors or any other form of iterative processing feels like an extension to what they already know.

A relational, set-based approach to data manipulation is quite different. To try and get a sense of this, let's start with the definition of a *set* by the creator of set theory—Georg Cantor:

> *By a "set" we mean any collection M into a whole of definite, distinct objects m (which are called the "elements" of M) of our perception or of our thought.*
>
> —*Joseph W. Dauben,* Georg Cantor *(Princeton University Press, 1990)*

There's so much in this definition of a set that I could spend pages and pages just trying to interpret the meaning of this sentence. But for the purposes of our discussion, I'll focus on two key aspects—one that appears explicitly in this definition and one that is implied:

- **Whole** Observe the use of the term *whole*. A set should be perceived and manipulated as a whole. Your attention should focus on the set as a whole, and not on the individual elements of the set. With iterative processing, this idea is violated because records of a file or a cursor are manipulated one at a time. A table in SQL represents (albeit not completely successfully) a relation from the relational model, and a relation is a set of elements that are alike (that is, have the same attributes). When you interact with tables using set-based queries, you interact with tables as whole, as opposed to interacting with the individual rows (the tuples of the relations)—both in terms of how you phrase your declarative SQL requests and in terms of your mindset and attention. This type of thinking is what's very hard for many to truly adopt.
- **Order** Observe that nowhere in the definition of a set is there any mention of the order of the elements. That's for a good reason—there is no order to the elements of a set. That's another thing that many have a hard time getting used to. Files and cursors do have a specific order to their records, and when you fetch the records one at a time, you can rely on this order. A table has no order to its rows because a table is a set. People who don't realize this often confuse the logical layer of the data model and the language with the physical layer of the implementation. They assume that if there's a certain index on the table, you get an implied guarantee that, when querying the table, the data will always be accessed in index order. And sometimes even the correctness of the solution will rely on this assumption. Of course, SQL Server doesn't provide any such guarantees. For example, the only way to guarantee that the rows in a result will be presented in a certain order is to add a presentation ORDER BY clause to the query. And if you do add one, you need to realize that what you get back is not relational because the result has a guaranteed order.

If you need to write SQL queries and you want to understand the language you're dealing with, you need to think in set-based terms. And this is where window functions can help bridge the gap between iterative thinking (one row at a time, in a certain order) and set-based thinking (seeing the

set as a whole, with no order). What can help you transition from one type of thinking to the other is the ingenious design of window functions.

For one, window functions support an ORDER BY clause when relevant, where you specify the order. But note that just because the function has an order specified doesn't mean it violates any relational concepts. The input to the query is relational with no ordering expectations, and the output of the query is relational with no ordering guarantees. It's just that there's ordering as part of the specification of the calculation, producing a result attribute in the resulting relation. There's no assurance that the result rows will be returned in the same order used by the window function; in fact, different window functions in the same query can specify different ordering. This kind of ordering has nothing to do—at least conceptually—with the query's presentation ordering. Figure 1-1 tries to illustrate the idea that both the input to a query with a window function and the output are relational, even though the window function has ordering as part of its specification. By using ovals in the illustration, and having the positions of the rows look different in the input and the output, I'm trying to express the fact that the order of the rows does not matter.

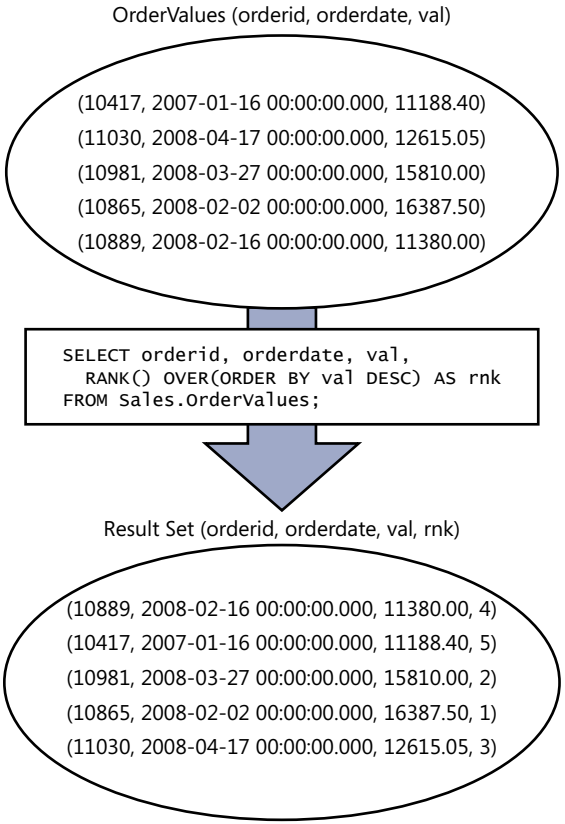

FIGURE 1-1 Input and output of a query with a window function.

There's another aspect of window functions that helps you gradually transition from thinking in iterative, ordered terms to thinking in set-based terms. When teaching a new topic, teachers

sometimes have to "lie" when explaining it. Suppose that you, as a teacher, know the student's mind is not ready to comprehend a certain idea if you explain it in full depth. You can sometimes get better results if you initially explain the idea in simpler, albeit not completely correct, terms to allow the student's mind to start processing the idea. Later, when the student's mind is ready for the "truth," you can provide the deeper, more correct meaning.

Such is the case with understanding how window functions are conceptually calculated. There's a basic way to explain the idea, although it's not really conceptually correct, but it's one that leads to the correct result! The basic way uses a row-at-a-time, ordered approach. And then there's the deep, conceptually correct way to explain the idea, but one's mind needs to be in a state of maturity to comprehend it. The deep way uses a set-based approach.

To demonstrate what I mean, consider the following query:

```
SELECT orderid, orderdate, val,
  RANK() OVER(ORDER BY val DESC) AS rnk
FROM Sales.OrderValues;
```

Here's an abbreviated output of this query (note there's no guarantee of presentation ordering here):

```
orderid  orderdate               val       rnk
-------- ----------------------- --------- ---
10865    2008-02-02 00:00:00.000 16387.50  1
10981    2008-03-27 00:00:00.000 15810.00  2
11030    2008-04-17 00:00:00.000 12615.05  3
10889    2008-02-16 00:00:00.000 11380.00  4
10417    2007-01-16 00:00:00.000 11188.40  5
...
```

The basic way to think of how the rank values are calculated conceptually is the following example (expressed as pseudocode):

```
arrange the rows sorted by val
iterate through the rows
for each row
  if the current row is the first row in the partition emit 1
  else if val is equal to previous val emit previous rank
  else emit count of rows so far
```

Figure 1-2 is a graphical depiction of this type of thinking.

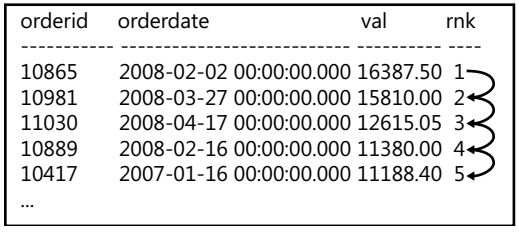

FIGURE 1-2 Basic understanding of the calculation of rank values.

Again, although this type of thinking leads to the correct result, it's not entirely correct. In fact, making my point is even more difficult because the process just described is actually very similar to how SQL Server physically handles the rank calculation. But my focus at this point is not the physical implementation, but rather the conceptual layer—the language and the logical model. What I meant by "incorrect type of thinking" is that conceptually, from a language perspective, the calculation is thought of differently, in a set-based manner—not iterative. Remember that the language is not concerned with the physical implementation in the database engine. The physical layer's responsibility is to figure out how to handle the logical request and both produce a correct result and produce it as fast as possible.

So let me attempt to explain what I mean by the deeper, more correct understanding of how the language thinks of window functions. The function logically defines—for each row in the result set of the query—a separate, independent window. Absent any restrictions in the window specification, each window consists of the set of all rows from the result set of the query as the starting point. But you can add elements to the window specification (for example, partitioning, framing, and so on, which I'll say more about later) that will further restrict the set of rows in each window. Figure 1-3 is a graphical depiction of this idea as it applies to our query with the RANK function.

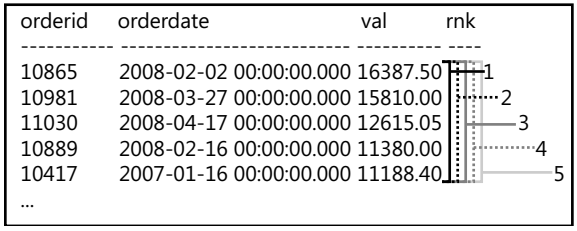

FIGURE 1-3 Deep understanding of the calculation of rank values.

With respect to each window function and row in the result set of the query, the OVER clause conceptually creates a separate window. In our query, we have not restricted the window specification in any way; we just defined the ordering specification for the calculation. So in our case, all windows are made of all rows in the result set. And they all coexist at the same time. And in each, the rank is calculated as one more than the number of rows that have a greater value in the *val* attribute than the current value.

As you might realize, it's more intuitive for many to think in the basic terms of the data being in an order and a process iterating through the rows one at a time. And that's okay when you're starting out with window functions because you get to write your queries—or at least the simple ones—correctly. As time goes by, you can gradually transition to the deeper understanding of the window functions' conceptual design and start thinking in a set-based manner.

Drawbacks of Alternatives to Window Functions

Window functions have several advantages compared to alternative, more traditional, ways to achieve the same calculations—for example, grouped queries, subqueries, and others. Here I'll provide a couple of straightforward examples. There are several other important differences beyond the advantages I'll show here, but it's premature to discuss those now.

I'll start with traditional grouped queries. Those do give you insight into new information in the form of aggregates, but you also lose something—the detail.

Once you group data, you're forced to apply all calculations in the context of the group. But what if you need to apply calculations that involve both detail and aggregates? For example, suppose that you need to query the Sales.OrderValues view and calculate for each order the percentage of the current order value of the customer total, as well as the difference from the customer average. The current order value is a detail element, and the customer total and average are aggregates. If you group the data by customer, you don't have access to the individual order values. One way to handle this need with traditional grouped queries is to have a query that groups the data by customer, define a table expression based on this query, and then join the table expression with the base table to match the detail with the aggregates. Here's a query that implements this approach:

```
WITH Aggregates AS
(
  SELECT custid, SUM(val) AS sumval, AVG(val) AS avgval
  FROM Sales.OrderValues
  GROUP BY custid
)
SELECT O.orderid, O.custid, O.val,
  CAST(100. * O.val / A.sumval AS NUMERIC(5, 2)) AS pctcust,
  O.val - A.avgval AS diffcust
FROM Sales.OrderValues AS O
  JOIN Aggregates AS A
    ON O.custid = A.custid;
```

Here's the abbreviated output generated by this query:

```
orderid  custid  val     pctcust  diffcust
-------- ------- ------- -------- ------------
10835    1       845.80  19.79    133.633334
10643    1       814.50  19.06    102.333334
10952    1       471.20  11.03    -240.966666
10692    1       878.00  20.55    165.833334
11011    1       933.50  21.85    221.333334
10702    1       330.00  7.72     -382.166666
10625    2       479.75  34.20    129.012500
10759    2       320.00  22.81    -30.737500
10926    2       514.40  36.67    163.662500
10308    2       88.80   6.33     -261.937500
...
```

Now imagine needing to also involve the percentage from the grand total and the difference from the grand average. To do this, you need to add another table expression, like so:

```
WITH CustAggregates AS
(
  SELECT custid, SUM(val) AS sumval, AVG(val) AS avgval
  FROM Sales.OrderValues
  GROUP BY custid
),
GrandAggregates AS
(
  SELECT SUM(val) AS sumval, AVG(val) AS avgval
  FROM Sales.OrderValues
)
SELECT O.orderid, O.custid, O.val,
  CAST(100. * O.val / CA.sumval AS NUMERIC(5, 2)) AS pctcust,
  O.val - CA.avgval AS diffcust,
  CAST(100. * O.val / GA.sumval AS NUMERIC(5, 2)) AS pctall,
  O.val - GA.avgval AS diffall
FROM Sales.OrderValues AS O
  JOIN CustAggregates AS CA
    ON O.custid = CA.custid
  CROSS JOIN GrandAggregates AS GA;
```

Here's the output of this query:

```
orderid  custid  val     pctcust  diffcust     pctall  diffall
-------- ------- ------- -------- ------------ ------- -------------
10835    1       845.80  19.79    133.633334   0.07    -679.252072
10643    1       814.50  19.06    102.333334   0.06    -710.552072
10952    1       471.20  11.03    -240.966666  0.04    -1053.852072
10692    1       878.00  20.55    165.833334   0.07    -647.052072
11011    1       933.50  21.85    221.333334   0.07    -591.552072
10702    1       330.00  7.72     -382.166666  0.03    -1195.052072
10625    2       479.75  34.20    129.012500   0.04    -1045.302072
10759    2       320.00  22.81    -30.737500   0.03    -1205.052072
10926    2       514.40  36.67    163.662500   0.04    -1010.652072
10308    2       88.80   6.33     -261.937500  0.01    -1436.252072
...
```

You can see how the query gets more and more complicated, involving more table expressions and more joins.

Another way to perform similar calculations is to use a separate subquery for each calculation. Here are the alternatives, using subqueries to the last two grouped queries:

```
-- subqueries with detail and customer aggregates
SELECT orderid, custid, val,
  CAST(100. * val /
        (SELECT SUM(O2.val)
         FROM Sales.OrderValues AS O2
         WHERE O2.custid = O1.custid) AS NUMERIC(5, 2)) AS pctcust,
  val - (SELECT AVG(O2.val)
         FROM Sales.OrderValues AS O2
         WHERE O2.custid = O1.custid) AS diffcust
FROM Sales.OrderValues AS O1;
```

```
-- subqueries with detail, customer and grand aggregates
SELECT orderid, custid, val,
  CAST(100. * val /
        (SELECT SUM(O2.val)
         FROM Sales.OrderValues AS O2
         WHERE O2.custid = O1.custid) AS NUMERIC(5, 2)) AS pctcust,
  val - (SELECT AVG(O2.val)
         FROM Sales.OrderValues AS O2
         WHERE O2.custid = O1.custid) AS diffcust,
  CAST(100. * val /
        (SELECT SUM(O2.val)
         FROM Sales.OrderValues AS O2) AS NUMERIC(5, 2)) AS pctall,
  val - (SELECT AVG(O2.val)
         FROM Sales.OrderValues AS O2) AS diffall
FROM Sales.OrderValues AS O1;
```

There are two main problems with the subquery approach. One, you end up with lengthy complex code. Two, SQL Server's optimizer is not coded at the moment to identify cases where multiple subqueries need to access the exact same set of rows; hence, it will use separate visits to the data for each subquery. This means that the more subqueries you have, the more visits to the data you get. Unlike the previous problem, this one is not a problem with the language, but rather with the specific optimization you get for subqueries in SQL Server.

Remember that the idea behind a window function is to define a window, or a set, of rows for the function to operate on. Aggregate functions are supposed to be applied to a set of rows; therefore, the concept of windowing can work well with those as an alternative to using grouping or subqueries. And when calculating the aggregate window function, you don't lose the detail. You use the OVER clause to define the window for the function. For example, to calculate the sum of all values from the result set of the query, simply use the following:

```
SUM(val) OVER()
```

If you do not restrict the window (empty parentheses), your starting point is the result set of the query.

To calculate the sum of all values from the result set of the query where the customer ID is the same as in the current row, use the partitioning capabilities of window functions (which I'll say more about later), and partition the window by *custid*, as follows:

```
SUM(val) OVER(PARTITION BY custid)
```

Note that the term *partitioning* suggests filtering rather than grouping.

Using window functions, here's how you address the request involving the detail and customer aggregates, returning the percentage of the current order value of the customer total as well as the difference from the average (with window functions in bold):

```
SELECT orderid, custid, val,
  CAST(100. * val / SUM(val) OVER(PARTITION BY custid) AS NUMERIC(5, 2)) AS pctcust,
  val - AVG(val) OVER(PARTITION BY custid) AS diffcust
FROM Sales.OrderValues;
```

And here's another query where you also add the percentage of the grand total and the difference from the grand average:

```
SELECT orderid, custid, val,
  CAST(100. * val / SUM(val) OVER(PARTITION BY custid) AS NUMERIC(5, 2)) AS pctcust,
  val - AVG(val) OVER(PARTITION BY custid) AS diffcust,
  CAST(100. * val / SUM(val) OVER() AS NUMERIC(5, 2)) AS pctall,
  val - AVG(val) OVER() AS diffall
FROM Sales.OrderValues;
```

Observe how much simpler and more concise the versions with the window functions are. Also, in terms of optimization, note that SQL Server's optimizer was coded with the logic to look for multiple functions with the same window specification. If any are found, SQL Server will use the same visit (whichever kind of scan was chosen) to the data for those. For example, in the last query, SQL Server will use one visit to the data to calculate the first two functions (the sum and average that are partitioned by *custid*), and it will use one other visit to calculate the last two functions (the sum and average that are nonpartitioned). I will demonstrate this concept of optimization in Chapter 4, "Optimization of Window Functions."

Another advantage window functions have over subqueries is that the initial window prior to applying restrictions is the result set of the query. This means that it's the result set after applying table operators (for example, joins), filters, grouping, and so on. You get this result set because of the phase of logical query processing in which window functions get evaluated. (I'll say more about this later in this chapter.) Conversely, a subquery starts from scratch—not from the result set of the outer query. This means that if you want the subquery to operate on the same set as the result of the outer query, it will need to repeat all query constructs used by the outer query. As an example, suppose that you want our calculations of the percentage of the total and the difference from the average to apply only to orders placed in the year 2007. With the solution using window functions, all you need to do is add one filter to the query, like so:

```
SELECT orderid, custid, val,
  CAST(100. * val / SUM(val) OVER(PARTITION BY custid) AS NUMERIC(5, 2)) AS pctcust,
  val - AVG(val) OVER(PARTITION BY custid) AS diffcust,
  CAST(100. * val / SUM(val) OVER() AS NUMERIC(5, 2)) AS pctall,
  val - AVG(val) OVER() AS diffall
FROM Sales.OrderValues
WHERE orderdate >= '20070101'
  AND orderdate < '20080101';
```

The starting point for all window functions is the set after applying the filter. But with subqueries, you start from scratch; therefore, you need to repeat the filter in all of your subqueries, like so:

```
SELECT orderid, custid, val,
  CAST(100. * val /
        (SELECT SUM(O2.val)
         FROM Sales.OrderValues AS O2
         WHERE O2.custid = O1.custid
           AND orderdate >= '20070101'
           AND orderdate < '20080101') AS NUMERIC(5, 2)) AS pctcust,
```

```
  val - (SELECT AVG(O2.val)
         FROM Sales.OrderValues AS O2
         WHERE O2.custid = O1.custid
           AND orderdate >= '20070101'
           AND orderdate < '20080101') AS diffcust,
  CAST(100. * val /
        (SELECT SUM(O2.val)
         FROM Sales.OrderValues AS O2
         WHERE orderdate >= '20070101'
           AND orderdate < '20080101') AS NUMERIC(5, 2)) AS pctall,
  val - (SELECT AVG(O2.val)
         FROM Sales.OrderValues AS O2
         WHERE orderdate >= '20070101'
           AND orderdate < '20080101') AS diffall
FROM Sales.OrderValues AS O1
WHERE orderdate >= '20070101'
  AND orderdate < '20080101';
```

Of course, you could use workarounds, such as first defining a common table expression (CTE) based on a query that performs the filter, and then have both the outer query and the subqueries refer to the CTE. However, my point is that with window functions, you don't need any workarounds because they operate on the result of the query. I will provide more details about this aspect in the design of window functions later in the chapter, in the "Query Elements Supporting Window Functions" section.

As mentioned earlier, window functions also lend themselves to good optimization, and often, alternatives to window functions don't get optimized as well, to say the least. Of course, there are cases where the inverse is also true. I explain the optimization of window functions in Chapter 4 and provide plenty of examples for using them efficiently in Chapter 5.

A Glimpse of Solutions Using Window Functions

The first four chapters of the book describe window functions and their optimization. The material is very technical, and even though I find it fascinating, I can see how some might find it a bit boring. What's usually much more interesting for people to read about is the use of the functions to solve practical problems, which is what this book gets to in the final chapter. When you see how window functions are used in problem solving, you truly realize their value. So how can I convince you it's worth your while to go through the more technical parts and not give up reading before you get to the more interesting part later? What if I give you a glimpse of a solution using window functions right now?

The querying task I will address here involves querying a table holding a sequence of values in some column and identifying the consecutive ranges of existing values. This problem is also known as the *islands problem*. The sequence can be a numeric one, a temporal one (which is more common), or any data type that supports *total ordering*. The sequence can have unique values or allow duplicates. The interval can be any fixed interval that complies with the column's type (for example, the integer 1, the integer 7, the temporal interval 1 day, the temporal interval 2 weeks, and so on). In Chapter 5, I will get to the different variations of the problem. Here, I'll just use a simple case to give you a sense

of how it works—using a numeric sequence with the integer 1 as the interval. Use the following code to generate the sample data for this task:

```
SET NOCOUNT ON;
USE TSQL2012;

IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;
GO

CREATE TABLE dbo.T1
(
  col1 INT NOT NULL
    CONSTRAINT PK_T1 PRIMARY KEY
);

INSERT INTO dbo.T1(col1)
  VALUES(2),(3),(11),(12),(13),(27),(33),(34),(35),(42);
GO
```

As you can see, there are some gaps in the col1 sequence in T1. Your task is to identify the consecutive ranges of existing values (also known as *islands*) and return the start and end of each island. Here's what the desired result should look like:

```
start_range end_range
----------- -----------
2           3
11          13
27          27
33          35
42          42
```

If you're curious as to the practicality of this problem, there are numerous production examples. Examples include producing availability reports, identifying periods of activity (for example, sales), identifying consecutive periods in which a certain criterion is met (for example, periods where a stock value was above or below a certain threshold), identifying ranges of license plates in use, and so on. The current example is very simplistic on purpose so that we can focus on the techniques used to solve it. The technique you will use to solve a more complicated case requires minor adjustments to the one you use to address the simple case. So consider it a challenge to come up with an efficient, set-based solution to this task. Try to first come up with a solution that works. Then repopulate the table with a decent number of rows—say, 10,000,000—and try your technique again. See how it performs. Only then take a look at my solutions.

Before showing the solution using window functions, I'll show one of the many possible solutions that use more traditional language constructs. In particular, I'll show one that uses subqueries. To explain the strategy of the first solution, examine the values in the T1.col1 sequence, where I added a conceptual attribute that doesn't exist at the moment and that I think of as a group identifier:

```
col1  grp
----- ---
2     a
3     a
11    b
```

```
12    b
13    b
27    c
33    d
34    d
35    d
42    e
```

The *grp* attribute doesn't exist yet. Conceptually, it is a value that uniquely identifies an island. This means that it has to be the same for all members of the same island and different then the values generated for other islands. If you manage to calculate such a group identifier, you can then group the result by this *grp* attribute and return the minimum and maximum *col1* values in each group (island). One way to produce this group identifier using traditional language constructs is to calculate, for each current *col1* value, the minimum *col1* value that is greater than or equal to the current one, and that has no following value.

As an example, following this logic, try to identify with respect to the value 2 what the minimum *col1* value is that is greater than or equal to 2 and that appears before a missing value? It's 3. Now try to do the same with respect to 3. You also get 3. So 3 is the group identifier of the island that starts with 2 and ends with 3. For the island that starts with 11 and ends with 13, the group identifier for all members is 13. As you can see, the group identifier for all members of a given island is actually the last member of that island.

Here's the T-SQL code required to implement this concept:

```
SELECT col1,
  (SELECT MIN(B.col1)
    FROM dbo.T1 AS B
    WHERE B.col1 >= A.col1
      -- is this row the last in its group?
      AND NOT EXISTS
        (SELECT *
         FROM dbo.T1 AS C
         WHERE C.col1 = B.col1 + 1)) AS grp
FROM dbo.T1 AS A;
```

This query generates the following output:

```
col1        grp
----------- -----------
2           3
3           3
11          13
12          13
13          13
27          27
33          35
34          35
35          35
42          42
```

The next part is pretty straightforward—define a table expression based on the last query, and in the outer query, group by the group identifier and return the minimum and maximum *col1* values for each group, like so:

```
SELECT MIN(col1) AS start_range, MAX(col1) AS end_range
FROM (SELECT col1,
        (SELECT MIN(B.col1)
         FROM dbo.T1 AS B
         WHERE B.col1 >= A.col1
           AND NOT EXISTS
             (SELECT *
              FROM dbo.T1 AS C
              WHERE C.col1 = B.col1 + 1)) AS grp
      FROM dbo.T1 AS A) AS D
GROUP BY grp;
```

There are two main problems with this solution. One, it's a bit complicated to follow the logic here. Two, it's horribly slow. I don't want to start going over query execution plans yet—there will be plenty of this later in the book—but I can tell you that for each row in the table, SQL Server will perform almost two complete scans of the data. Now think of a sequence of 10,000,000 rows, and try to translate it to the amount of work involved. The total number of rows that will need to be processed is simply enormous.

The next solution is also one that calculates a group identifier, but using window functions. The first step in the solution is to use the ROW_NUMBER function to calculate row numbers based on *col1* ordering. I will provide the gory details about the ROW_NUMBER function later in the book; for now, it suffices to say that it computes unique integers within the partition starting with 1 and incrementing by 1 based on the given ordering.

With this in mind, the following query returns the *col1* values and row numbers based on *col1* ordering:

```
SELECT col1, ROW_NUMBER() OVER(ORDER BY col1) AS rownum
FROM dbo.T1;

col1        rownum
----------- --------------------
2           1
3           2
11          3
12          4
13          5
27          6
33          7
34          8
35          9
42          10
```

Now focus your attention on the two sequences. One (*col1*) is a sequence with gaps, and the other (*rownum*) is a sequence without gaps. With this in mind, try to figure out what's unique to the relationship between the two sequences in the context of an island. Within an island, both sequences keep incrementing by a fixed interval. Therefore, the difference between the two is constant. For

the next island, *col1* increases by more than 1, whereas *rownum* increases just by 1, so the difference keeps growing. In other words, the difference between the two is constant and unique for each island. Run the following query to calculate this difference:

```
SELECT col1, col1 - ROW_NUMBER() OVER(ORDER BY col1) AS diff
FROM dbo.T1;

col1        diff
----------- --------------------
2           1
3           1
11          8
12          8
13          8
27          21
33          26
34          26
35          26
42          32
```

You can see that this difference satisfies the two requirements of our group identifier; therefore, you can use it as such. The rest is the same as in the previous solution; namely, you group the rows by the group identifier and return the minimum and maximum *col1* values in each group, like so:

```
SELECT MIN(col1) AS start_range, MAX(col1) AS end_range
FROM (SELECT col1,
        -- the difference is constant and unique per island
        col1 - ROW_NUMBER() OVER(ORDER BY col1) AS grp
      FROM dbo.T1) AS D
GROUP BY grp;
```

Observe how concise and simple the solution is. Of course, it's always a good idea to add comments to help those who see the solution for the first time better understand it.

The solution is also highly efficient. The work involved in assigning the row numbers is negligible compared to the previous solution. It's just a single ordered scan of the index on *col1* and an iterator that keeps incrementing a counter. In a performance test I ran with a sequence with 10,000,000 rows, this query finished in 10 seconds. Other solutions ran for a much longer time.

I hope that this glimpse to solutions using window functions was enough to intrigue you and help you see that they contain immense power. Now we'll get back to studying the technicalities of window functions. Later in the book, you will have a chance to see many more examples.

Elements of Window Functions

The specification of a window function's behavior appears in the function's OVER clause and involves multiple elements. The three core elements are partitioning, ordering, and framing. Not all window functions support all elements. As I describe each element, I'll also indicate which functions support it.

Partitioning

The partitioning element is implemented with a PARTITION BY clause and is supported by all window functions. It restricts the window of the current calculation to only those rows from the result set of the query that have the same values in the partitioning columns as in the current row. For example, if your function uses *PARTITION BY custid* and the *custid* value in the current row is 1, the window with respect to the current row is all rows from the result set of the query that have a *custid* value of 1. If the *custid* value of the current row is 2, the window with respect to the current row is all rows with a *custid* of 2.

If a PARTITION BY clause is not specified, the window is not restricted. Another way to look at it is that inf case explicit partitioning wasn't specified, the default partitioning is to consider the entire result set of the query as one partition.

If it wasn't obvious, let me point out that different functions in the same query can have different partitioning specifications. Consider the query in Listing 1-1 as an example.

LISTING 1-1 Query with Two RANK Calculations

```
SELECT custid, orderid, val,
  RANK() OVER(ORDER BY val DESC) AS rnk_all,
  RANK() OVER(PARTITION BY custid
              ORDER BY val DESC) AS rnk_cust
FROM Sales.OrderValues;
```

Observe that the first RANK function (which generates the attribute *rnk_all*) relies on the default partitioning, and the second RANK function (which generates *rnk_cust*) uses explicit partitioning by *custid*. Figure 1-4 illustrates the partitions defined for a sample of three results of calculations in the query: one *rnk_all* value and two *rnk_cust* values.

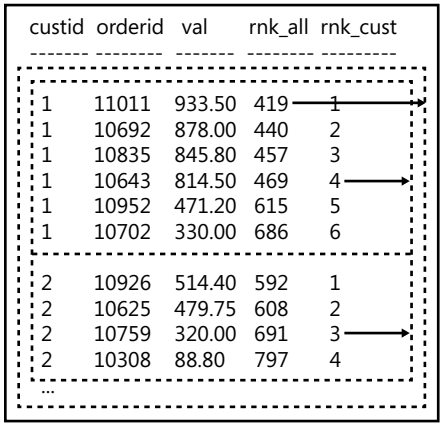

FIGURE 1-4 Window partitioning.

The arrows point from the result values of the functions to the window partitions that were used to compute them.

Ordering

The ordering element defines the ordering for the calculation, if relevant, within the partition. In standard SQL, all functions support an ordering element. As for SQL Server, initially it didn't support the ordering element with aggregate functions; rather, it only supported partitioning. Support for ordering for aggregates was added in SQL Server 2012.

Interestingly, the ordering element has a slightly different meaning for different function categories. With ranking functions, ordering is intuitive. For example, when using descending ordering, the RANK function returns one more than the number of rows in your respective partition that have a greater ordering value than yours. When using ascending ordering, the function returns one more than the number of rows in the pattern with a lower ordering value than yours. Figure 1-5 illustrates the rank calculations from Listing 1-1 shown earlier—this time including the interpretation of the ordering element.

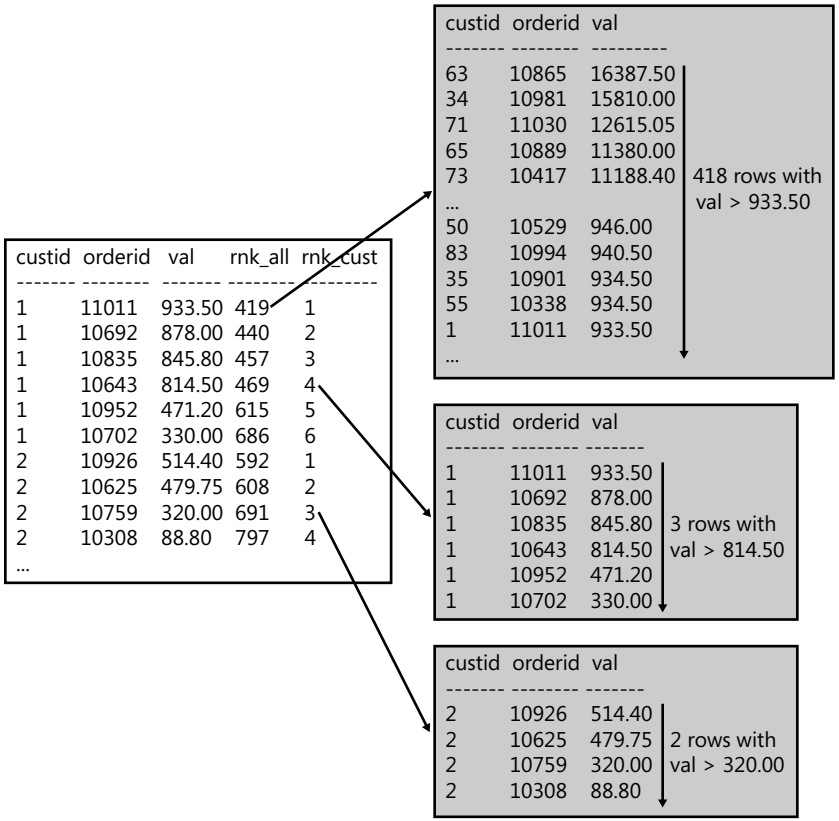

FIGURE 1-5 Window ordering.

Figure 1-5 depicts the windows of only three of the rank calculations. Of course, there are many more—1,660, to be precise. That's because there are 830 rows involved, and for each row, two rank calculations are made. What's interesting to note here is that conceptually it's as if all those windows coexist simultaneously.

Aggregate window functions have a slightly different meaning for *ordering* compared to *ranking* window functions. With aggregates, contrary to what some might think, ordering has nothing to do with the order in which the aggregate is applied; rather, the ordering element gives meaning to the framing options that I will describe next. In other words, the ordering element is an aid to define which rows to restrict in the window.

Framing

Framing is essentially another filter that further restricts the rows in the partition. It is applicable to aggregate window functions as well as to three of the offset functions: FIRST_VALUE, LAST_VALUE, and NTH_VALUE. Think of this windowing element as defining two points in the current row's partition based on the given ordering, framing the rows that the calculation will apply to.

The framing specification in the standard includes a ROWS or RANGE option that defines the starting row and ending row of the frame, as well as a window frame-exclusion option. SQL Server 2012 introduced support for framing, with full implementation of the ROWS option, partial implementation of the RANGE option, and no implementation of the window frame-exclusion option.

The ROWS option allows you to indicate the points in the frame as an offset in terms of the number of rows with respect to the current row. The RANGE option is more dynamic, defining the offsets in terms of a difference between the value of the frame point and the current row's value. The window frame-exclusion option specifies what to do with the current row and its peers in case of ties. This explanation might seem far from clear or sufficient, but I don't want to get into the details just yet. There will be plenty of that later. For now, I just want to introduce the concept and provide a simple example. Following is a query against the EmpOrders view, calculating the running total quantity for each employee and order month:

```
SELECT empid, ordermonth, qty,
  SUM(qty) OVER(PARTITION BY empid
                ORDER BY ordermonth
                ROWS BETWEEN UNBOUNDED PRECEDING
                         AND CURRENT ROW) AS runqty
FROM Sales.EmpOrders;
```

Observe that the window function applies the SUM aggregate to the *qty* attribute, partitions the window by *empid*, orders the partition rows by *ordermonth*, and frames the partition rows based on the given ordering between unbounded preceding (no low boundary point) and the current row. In other words, the result will be the sum of all prior rows in the frame, inclusive of the current row. This query generates the following output, shown here in abbreviated form:

```
empid  ordermonth              qty         run_qty
------ ----------------------- ----------- -----------
1      2006-07-01 00:00:00.000 121         121
1      2006-08-01 00:00:00.000 247         368
1      2006-09-01 00:00:00.000 255         623
1      2006-10-01 00:00:00.000 143         766
1      2006-11-01 00:00:00.000 318         1084
...
2      2006-07-01 00:00:00.000 50          50
2      2006-08-01 00:00:00.000 94          144
2      2006-09-01 00:00:00.000 137         281
2      2006-10-01 00:00:00.000 248         529
2      2006-11-01 00:00:00.000 237         766
...
```

Observe how the window specification is as easy to read as plain English. I will provide much more detail about the framing options in Chapter 2.

Query Elements Supporting Window Functions

Window functions aren't supported in all query clauses; rather, they're supported only in the SELECT and ORDER BY clauses. To help you understand the reason for this restriction, I first need to explain a concept called *logical query processing*. Then I'll get to the clauses that support window functions, and finally I'll explain how to circumvent the restriction with the other clauses.

Logical Query Processing

Logical query processing describes the conceptual way in which a SELECT query is evaluated according to the logical language design. It describes a process made of a series of steps, or phases, that proceed from the query's input tables to the query's final result set. Note that by "logical query processing," I mean the conceptual way in which the query is evaluated—not necessarily the physical way SQL Server processes the query. As part of the optimization, SQL Server can make shortcuts, rearrange the order of some steps, and pretty much do whatever it likes. But that's as long as it guarantees that it will produce the same output as the one defined by logical query processing applied to the declarative query request.

Each step in logical query processing operates on one or more tables (sets of rows) that serve as its input and returns a table as its output. The output table of one step then becomes the input table for the next step.

Figure 1-6 is a flow diagram illustrating the logical query processing flow in SQL Server 2012.

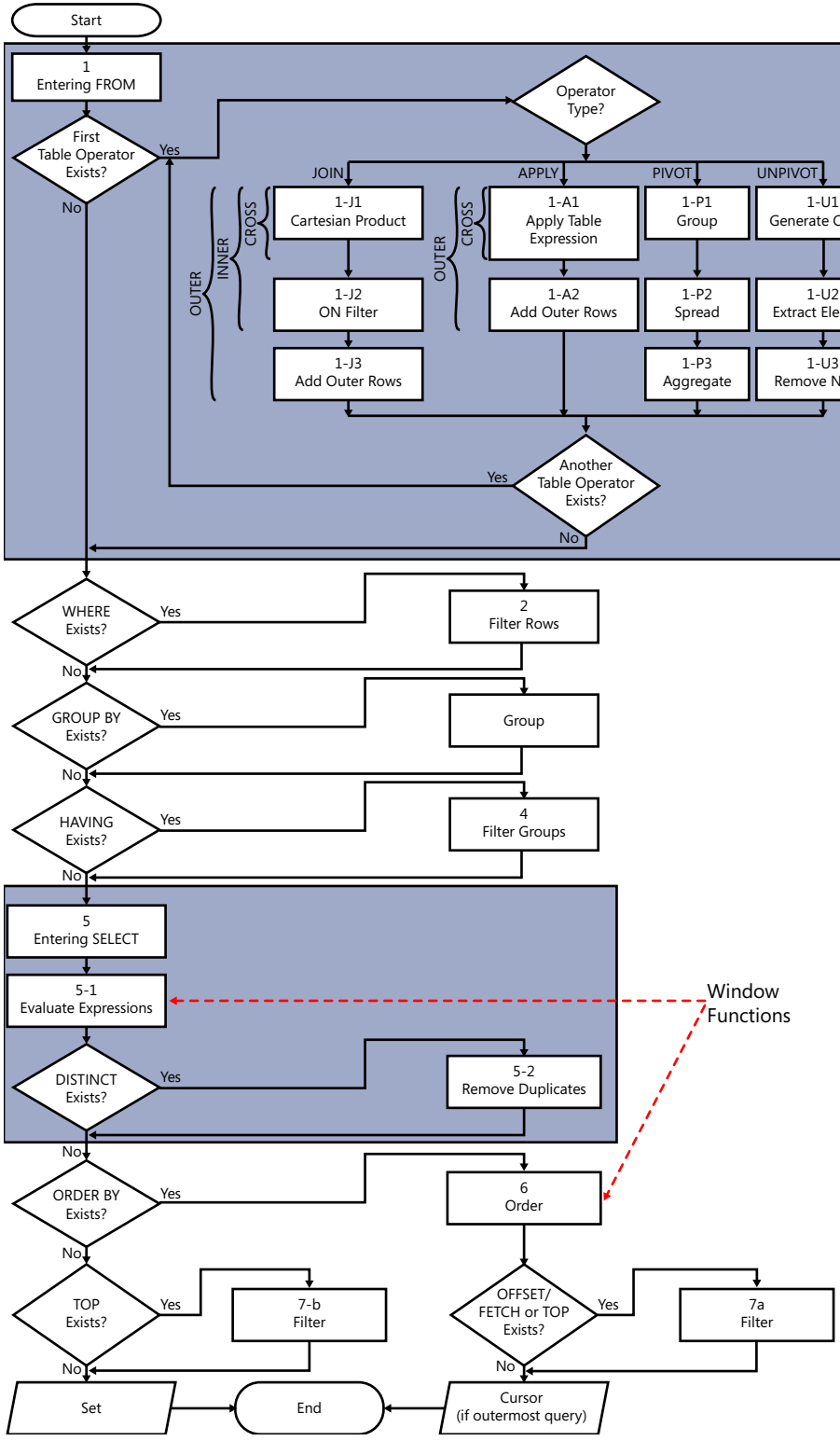

FIGURE 1-6 Logical query processing.

Note that when you write a query, the SELECT clause appears first in terms of the keyed-in order, but observe that in terms of the logical query processing order, it appears almost last—just before the ORDER BY clause is handled.

There's much more to say about logical query processing, but the details are a topic for another book. For the purposes of our discussion, what's important to note is the order in which the various clauses are evaluated. The following list shows the order (with the phases in which window functions are allowed shown in bold):

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
   - 5-1. **Evaluate Expressions**
   - 5-2. Remove Duplicates
6. **ORDER BY**
7. OFFSET-FETCH/TOP

Understanding logical query processing and the logical query processing order enables you to understand the motivation behind restricting window functions to only specific clauses.

Clauses Supporting Window Functions

As illustrated in Figure 1-6, only the query clauses SELECT and ORDER BY support window functions directly. The reason for the limitation is to avoid ambiguity by operating on (almost) the final result set of the query as the starting point for the window. If window functions are allowed in phases previous to the SELECT phase, their initial window could be different than that in the SELECT phase, and therefore, with some query forms, it could be very difficult to figure out the right result. I'll try to demonstrate the ambiguity problem through an example. First run the following code to create the table T1 and populate it with sample data:

```
SET NOCOUNT ON;
USE TSQL2012;
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;
GO

CREATE TABLE dbo.T1
(
  col1 VARCHAR(10) NOT NULL
    CONSTRAINT PK_T1 PRIMARY KEY
);

INSERT INTO dbo.T1(col1)
  VALUES('A'),('B'),('C'),('D'),('E'),('F');
```

Suppose that window functions were allowed in phases prior to the SELECT—for example, in the WHERE phase. Consider then the following query, and try to figure out which *col1* values should appear in the result:

```
SELECT col1
FROM dbo.T1
WHERE col1 > 'B'
  AND ROW_NUMBER() OVER(ORDER BY col1) <= 3;
```

Before you assume that the answer should obviously be the values C, D, and E, consider the all-at-once concept in SQL. The concept of *all-at-once* means that all expressions that appear in the same logical phase are conceptually evaluated at the same point in time. This means that the order in which the expressions are evaluated shouldn't matter. With this in mind, the following query should be semantically equivalent to the previous one:

```
SELECT col1
FROM dbo.T1
WHERE ROW_NUMBER() OVER(ORDER BY col1) <= 3
  AND col1 > 'B';
```

Now, can you figure out what the right answer is? Is it C, D, and E, or is it just C?

That's an example of the ambiguity I was talking about. By restricting window functions to only the SELECT and ORDER BY clauses of a query, this ambiguity is eliminated.

Looking at Figure 1-6, you might have noticed that within the SELECT phase, it's step 5-1 (Evaluate Expressions) that supports window functions, and this step is evaluated before step 5-2 (Remove Duplicates). If you wonder why it is important to know such subtleties, I'll demonstrate why.

Following is a query returning the *empid* and *country* attributes of all employees from the Employees table:

```
SELECT empid, country
FROM HR.Employees;

empid       country
----------- ---------------
1           USA
2           USA
3           USA
4           USA
5           UK
6           UK
7           UK
8           USA
9           UK
```

Next, examine the following query and see if you can guess what its output is before executing it:

```
SELECT DISTINCT country, ROW_NUMBER() OVER(ORDER BY country) AS rownum
FROM HR.Employees;
```

Some expect to get the following output:

```
country         rownum
--------------- --------------------
UK              1
USA             2
```

But in reality you get this:

```
country         rownum
--------------- --------------------
UK              1
UK              2
UK              3
UK              4
USA             5
USA             6
USA             7
USA             8
USA             9
```

Now consider that the ROW_NUMBER function in this query is evaluated in step 5-1 where the SELECT list expressions are evaluated—prior to the removal of the duplicates in step 5-2. The ROW_NUMBER function assigns nine unique row numbers to the nine employee rows, and then the DISTINCT clause has no duplicates left to remove.

When you realize this and understand that it has to do with the logical query processing order of the different elements, you can think of a solution. For example, you can have a table expression defined based on a query that just returns distinct countries and have the outer query assign the row numbers after duplicates are removed, like so:

```
WITH EmpCountries AS
(
  SELECT DISTINCT country FROM HR.Employees
)
SELECT country, ROW_NUMBER() OVER(ORDER BY country) AS rownum
FROM EmpCountries;
```

Can you think of other ways to solve the problem, perhaps even simpler ways than this one?

The fact that window functions are evaluated in the SELECT or ORDER BY phase means that the window defined for the calculation—before applying further restrictions—is the intermediate form of rows of the query after all previous phases—that is, after applying the FROM with all of its table operators (for example, joins), and after the WHERE filtering, the grouping, and the filtering of the groups. Consider the following query as an example:

```
SELECT O.empid,
  SUM(OD.qty) AS qty,
  RANK() OVER(ORDER BY SUM(OD.qty) DESC) AS rnk
FROM Sales.Orders AS O
  JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid
WHERE O.orderdate >= '20070101'
```

```
  AND O.orderdate < '20080101'
GROUP BY O.empid;

empid  qty   rnk
------ ----- ---
4      5273  1
3      4436  2
1      3877  3
8      2843  4
2      2604  5
7      2292  6
6      1738  7
5      1471  8
9      955   9
```

First the FROM clause is evaluated and the join is performed. Then only the rows where the order year is 2007 are filtered. Then the remaining rows are grouped by employee ID. Only then are the expressions in the SELECT list evaluated, including the RANK function, which is calculated based on ordering by the total quantity descending. If there were other window functions in the SELECT list, they would all use the same result set as their starting point. Recall from earlier discussions about alternative options to window functions (for example, subqueries) that they start their view of the data from scratch—meaning that you have to repeat all the logic you have in the outer query in each of your subqueries, leading to much more verbose code.

Circumventing the Limitations

I explained the reasoning behind disallowing the use of window functions in logical query processing phases that are evaluated prior to the SELECT clause. But what if you need to filter by or group by a calculation based on window functions? The solution is to use a table expression such as a CTE or a derived table. Have a query invoke the window function in its SELECT list, assigning the calculation an alias. Define a table expression based on that query, and then have the outer query refer to that alias where you need it.

Here's an example showing how you can filter by the result of a window function using a CTE:

```
WITH C AS
(
  SELECT orderid, orderdate, val,
    RANK() OVER(ORDER BY val DESC) AS rnk
  FROM Sales.OrderValues
)
SELECT *
FROM C
WHERE rnk <= 5;

orderid  orderdate               val       rnk
-------- ----------------------- --------- ----
10865    2008-02-02 00:00:00.000 16387.50  1
10981    2008-03-27 00:00:00.000 15810.00  2
11030    2008-04-17 00:00:00.000 12615.05  3
10889    2008-02-16 00:00:00.000 11380.00  4
10417    2007-01-16 00:00:00.000 11188.40  5
```

With modification statements, window functions are disallowed altogether because those don't support SELECT and ORDER BY clauses. But there are cases where involving window functions in modification statements is needed. Table expressions can be used to address this need as well because T-SQL supports modifying data through table expressions. I'll demonstrate this capability with an UPDATE example. First run the following code to create a table called T1 with columns *col1* and *col2* and populate it with sample data:

```
SET NOCOUNT ON;
USE TSQL2012;
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;
GO

CREATE TABLE dbo.T1
(
  col1 INT NULL,
  col2 VARCHAR(10) NOT NULL
);

INSERT INTO dbo.T1(col2)
  VALUES('C'),('A'),('B'),('A'),('C'),('B');
```

Explicit values were provided in *col2*, and NULLs were used as defaults in *col1*.

Suppose this table represents a situation with data-quality problems. A key wasn't enforced in this table, and therefore it is not possible to uniquely identify rows. You want to assign unique *col1* values in all rows. You're thinking of using the ROW_NUMBER function in an UPDATE statement, like so:

```
UPDATE dbo.T1
  SET col1 = ROW_NUMBER() OVER(ORDER BY col2);
```

But remember that this is not allowed. The workaround is to write a query against T1 returning *col1* and an expression based on the ROW_NUMBER function (call it *rownum*); define a table expression based on this query; finally, have an outer UPDATE statement against the CTE assign *rownum* to *col1*, like so:

```
WITH C AS
(
  SELECT col1, col2,
    ROW_NUMBER() OVER(ORDER BY col2) AS rownum
  FROM dbo.T1
)
UPDATE C
  SET col1 = rownum;
```

Query T1, and observe that all rows got unique *col1* values:

```
SELECT col1, col2
FROM dbo.T1;

col1        col2
----------- ----------
5           C
1           A
3           B
2           A
6           C
4           B
```

Potential for Additional Filters

I provided a workaround in T-SQL that allows you to use window functions indirectly in query elements that don't support those directly. The workaround is a table expression in the form of a CTE or derived table. It's nice to have a workaround, but a table expression adds an extra layer to the query and complicates it a bit. The examples I showed are quite simple, but think about long and complex queries to begin with. Can you have a simpler solution that doesn't require this extra layer?

With window functions, SQL Server doesn't have a solution at the moment. It's interesting, though, to see how others coped with this problem. Teradata for example created a filtering clause it calls *QUALIFY* that is conceptually evaluated after the SELECT clause. This means that it can refer to window functions directly, as in the following example:

```
SELECT orderid, orderdate, val
FROM Sales.OrderValues
QUALIFY RANK() OVER(ORDER BY val DESC) <= 5;
```

Furthermore, you can refer to column aliases defined in the SELECT list, like so:

```
SELECT orderid, orderdate, val,
  RANK() OVER(ORDER BY val DESC) AS rnk
FROM Sales.OrderValues
QUALIFY rnk <= 5;
```

The QUALIFY clause isn't defined in standard SQL; rather, it's a Teradata-specific feature. However, it seems like a very interesting solution, and it would be nice to see both the standard and SQL Server providing a solution to this need.

Reuse of Window Definitions

Suppose that you need to invoke multiple window functions in the same query and part of the window specification (or all of it) is common to multiple functions. If you indicate the complete window specifications in all functions, the code can quickly get lengthy. Here's an example illustrating the problem:

```
SELECT empid, ordermonth, qty,
  SUM(qty) OVER (PARTITION BY empid
                 ORDER BY ordermonth
                 ROWS BETWEEN UNBOUNDED PRECEDING
                          AND CURRENT ROW) AS run_sum_qty,
  AVG(qty) OVER (PARTITION BY empid
                 ORDER BY ordermonth
                 ROWS BETWEEN UNBOUNDED PRECEDING
                          AND CURRENT ROW) AS run_avg_qty,
  MIN(qty) OVER (PARTITION BY empid
                 ORDER BY ordermonth
                 ROWS BETWEEN UNBOUNDED PRECEDING
                          AND CURRENT ROW) AS run_min_qty,
  MAX(qty) OVER (PARTITION BY empid
                 ORDER BY ordermonth
                 ROWS BETWEEN UNBOUNDED PRECEDING
                          AND CURRENT ROW) AS run_max_qty
FROM Sales.EmpOrders;
```

Standard SQL has an answer to this problem in the form of a clause called WINDOW that allows naming a window specification or part of it; then you can refer to that name in other window definitions—ones used by window functions or even by a definition of another window name. This clause is conceptually evaluated after the HAVING clause and before the SELECT clause.

SQL Server doesn't yet support the WINDOW clause. But according to standard SQL, you can abbreviate the preceding query using the WINDOW clause like so:

```
SELECT empid, ordermonth, qty,
  SUM(qty) OVER W1 AS run_sum_qty,
  AVG(qty) OVER W1 AS run_avg_qty,
  MIN(qty) OVER W1 AS run_min_qty,
  MAX(qty) OVER W1 AS run_max_qty
FROM Sales.EmpOrders
WINDOW W1 AS ( PARTITION BY empid
               ORDER BY ordermonth
               ROWS BETWEEN UNBOUNDED PRECEDING
                        AND CURRENT ROW );
```

That's quite a difference, as you can see. In this case, the WINDOW clause assigns the name W1 to a complete window specification with partitioning, ordering, and framing options. Then all four functions refer to W1 as their window specification. The WINDOW clause is actually quite sophisticated. As mentioned, it doesn't have to name a complete window specification; rather, it can even name only part of it. Then a window definition can include a mix of named parts plus explicit parts. As an aside,

the coverage of standard SQL for the WINDOW clause is a striking length of 10 pages! And trying to decipher the details is no picnic.

It would be great to see SQL Server add such support in the future, especially now that it has extensive support for window functions and people are likely to end up with lengthy window specifications.

Summary

This chapter introduced the concept of windowing in SQL. It provided the background to window functions, explaining the motivation for their use. The chapter then provided a glimpse of solving querying tasks using window functions by addressing the task of identifying ranges of existing values in a sequence—a problem also known as *identifying islands*. The chapter then proceeded to explain the design of window functions, covering the elements involved in window specifications: partitioning, ordering, and framing. Finally, this chapter explained how standard SQL addresses the need to reuse a window specification or part of it. The next chapter provides a breakdown of window functions, getting into more detail.

Index

Symbols

+ (concatenation operator), 150

A

aggregate functions. *See also* COUNT function; MAX function; MIN function; SUM function
  described, 3–4, 13, 33–34
  distinct, 51–53
  DISTINCT clause and, 85
  expanding all frame rows, 122
  filtering, 49–51
  framing and, 22, 36–49, 119–128
  general form, 37
  nested, 53–57
  optimizing, 116–128
  ordered set functions and, 81
  ordering and, 22, 36–49, 119–128
  partitioning and, 13–14, 34–36
  SQL Server support, 4
  SQL standard support, 3
  without framing, 116–118
  without ordering, 116–118
aggregation element (pivoting technique), 148
all-at-once concept, 26
Amdahl's Law, 114
APPLY operator
  Packing Intervals T-SQL solution, 192
  parallel APPLY technique, 112–115, 121–122, 127
  Top-N-per-Group T-SQL solution, 152
autogenerating numbers, 62
auxiliary table of numbers, virtual, 133–136
AVG function
  computing cumulative values, 126
  expanding all frame rows, 122
  usage example, 41

B

backward scans, indexes and, 105–108

C

calculating running totals. *See* Running Totals solution (T-SQL)
Cantor, Georg, 7
carry-along-sort technique, 153, 157
CASE expression
  distinct aggregates and, 52
  FILTER clause workaround, 50
  hypothetical set functions and, 84
  Running Totals T-SQL solution, 168
  usage example, 78–79
CHECKSUM_AGG function, 122, 124
CLR (Common Language Runtime)
  Running Totals T-SQL solution, 164–166, 171
  SQL Server support, 34
  user-defined aggregates, 81
COALESCE function, 99, 150
Codd, E. F., 6
columnstore indexes, 108
Common Language Runtime (CLR)
  Running Totals T-SQL solution, 164–166, 171
  SQL Server support, 34
  user-defined aggregates, 81
common table expressions (CTEs)
  distinct aggregates and, 52
  filtering and, 15, 28–30
  Gaps and Islands T-SQL solution, 199–201
  grouped aggregates and, 55
  Max Concurrent Intervals T-SQL solution, 178–180
  Median T-SQL solution, 205
  Mode T-SQL solution, 155–157
  ordered set functions and, 97

common table expressions, *continued*
Packing Intervals T-SQL solution, 184–188
Pivoting T-SQL solution, 148–149
Removing Duplicates T-SQL solution, 147
Running Totals T-SQL solution, 166
Sequences of Keys T-SQL solution, 139
Sorting Hierarchies T-SQL solution, 206–210
Top-N-Per-Group T-SQL solution, 154
usage example, 78–79
Virtual Auxiliary Table of Numbers T-SQL solution, 134
Compute Scalar iterator
computing cumulative values, 127
distribution functions and, 129–131
expanding all frame rows, 123
concatenating strings
carry-along-sort technique and, 153
concatenation operator (+), 150
CONCAT function, 150
ordered set functions and, 81, 98–99
CONCAT function, 150
Conditional Aggregate solution (T-SQL), 204–206
constants
ordering based on, 109
OVER clause and, 62
CONVERT function, 153
COUNT function
about, 3
calculating percentile rank, 70
computing cumulative values, 126–127
expanding all frame rows, 122
grouped aggregates and, 56
Median T-SQL solution, 202
Mode T-SQL solution, 156
ROW_NUMBER function and, 59–60
usage example, 84, 86
COUNT_BIG function
computing cumulative values, 126
expanding all frame rows, 122
CROSS APPLY operator
about, 88
Packing Intervals T-SQL solution, 190
cross joining tables, 133–136
CTEs (common table expressions)
distinct aggregates and, 52
filtering and, 15, 28–30
Gaps and Islands T-SQL solution, 199–201
grouped aggregates and, 55
Max Concurrent Intervals T-SQL solution, 178–180
Median T-SQL solution, 205
Mode T-SQL solution, 155–157
ordered set functions and, 97
Packing Intervals T-SQL solution, 184–188
Pivoting T-SQL solution, 148–149
Removing Duplicates T-SQL solution, 147
Running Totals T-SQL solution, 166
Sequences of Keys T-SQL solution, 139
Sorting Hierarchies T-SQL solution, 206–210
Top-N-Per-Group T-SQL solution, 154
usage example, 78–79
Virtual Auxiliary Table of Numbers T-SQL solution, 134
CUME_DIST function
about, 4, 68–69
hypothetical-set-function form of, 82, 86–87, 89
optimizing, 129
cumulative values, computing, 126–128
CURRENT ROW option
RANGE clause, 43–47, 77
ROWS clause, 37–38
cursor/iterative programming
Max Concurrent Intervals T-SQL solution, 175–177
Running Totals T-SQL solution, 158, 162–164
set-based versus, 6–10

D

data-quality issues solution, 138–139
data warehouses, populating time dimension in, 137
DATEADD function, 195, 197
date and time values
Gaps and Islands T-SQL solution, 193–201
Sequences of Date and Time Values T-SQL solution, 137–138
DATEDIFF function
Gaps and Islands T-SQL solution, 195
Sequences of Date and Time Values T-SQL solution, 137
Dauben, Joseph W., 7
DBCC OPTIMIZER_WHATIF command, 113
degree of parallelism (DOP), 113–114
DELETE clause, 147
DENSE_RANK function
about, 4, 57, 66–67
determinism and, 67

Gaps and Islands T-SQL solution, 196–197
hypothetical-set-function form of, 82, 84–85, 87–89
optimizing, 111–112
determinism
DENSE_RANK function and, 67
RANK function and, 67
ROW_NUMBER function and, 60–64
Discard Results After Execution query option, 136
DISTINCT option
aggregate functions and, 51, 85
usage example, 27, 91
distribution functions. *See also* CUME_DIST function; PERCENTILE_CONT function; PERCENTILE_DISC function; PERCENT_RANK function
about, 4, 68
inverse distribution, 68, 71–73, 90–94, 129–132
optimization of, 128–132
ordering and, 68
partitioning and, 68
rank distribution, 68–71, 82–90, 128–129
SQL Server support, 4, 68
SQL standard support, 3
DOP (degree of parallelism), 113–114
duplicate data, removing, 145–148

E

equality filters, 105
EXCLUDE CURRENT ROW option, 47
EXCLUDE GROUP option, 47
EXCLUDE NO OTHERS option, 47
EXCLUDE TIES option, 47
Extended Event, 126

F

fast-track case, 119–122
FILTER clause, 49–51
filtering
CTEs and, 15, 28–30
equality filters, 105
FILTER clause and, 49–51
Max Concurrent Intervals T-SQL solution, 174–175
OFFSET/FETCH option, 134–136, 144–145, 151–153
Packing Intervals T-SQL solution, 187
Paging T-SQL solution, 143–145
QUALIFY clause and, 30
Running Totals T-SQL solution, 160–162
Top-N-per-Group T-SQL solution, 151–154
TOP option, 134–136, 148, 151–153
WHERE clause and, 117
Filter iterator, 117
FIRST_VALUE function
about, 4, 76–79
expanding all frame rows, 122, 124
framing element and, 22
ordered-set-function form of, 94–96
Flanaghan, Ben, 179
FLOOR function, 188
forward scans, indexes and, 105
FOR XML PATH('') option, 99
frame rows, expanding, 122–126
framing
about, 22–23
aggregate functions and, 22, 36–49, 119–128
offset functions and, 22, 119–128
ordering and, 36
RANGE option, 22, 37, 43–47
ROWS option, 22, 37–43
FROM clause
processing order, 54
usage example, 27–28

G

gaps
Gaps and Islands T-SQL solution, 193–201
Sequences of Keys T-SQL solution, 138–142
GetNums function
creating, 102–103
Gaps and Islands T-SQL solution, 198
Sequences of Date and Time Values T-SQL solution, 137
Virtual Auxiliary Table of Numbers T-SQL solution, 136
GROUP BY clause
nested aggregates, 53
processing order, 54
grouped queries
drawbacks of, 11–12
grouped aggregates and, 54
ordered set functions and, 81
grouping element (pivoting technique), 148

H

Hash partitioning, 114
HAVING clause, 54
hypothetical set functions
  CUME_DIST, 82, 86–87, 89
  DENSE_RANK, 82, 84–85, 87–89
  general solution for, 87–90
  as ordered set functions, 82–90
  PERCENT_RANK, 82, 85–87, 89
  RANK, 82–84, 87–89

I

identifying islands problem, 15–19, 195–201
INCLUDE clause, 104
indexed sequential access method (ISAM), 7
indexes
  backward scans, 105–108
  columnstore indexes, 108
  indexing guidelines, 103
  Max Concurrent Intervals T-SQL solution, 171–180
  Mode T-SQL solution, 154–158
  Packing Intervals T-SQL solution, 183–184, 188–189
  Paging T-SQL solution, 143–145
  POC indexes, 104–105, 151–152, 161
  Running Totals T-SQL solution, 161, 168
  Top-N-per-Group T-SQL solution, 151–152, 158
Index Scan iterator
  backward scans, 107
  Max Concurrent Intervals T-SQL solution, 174
  ROW_NUMBER function optimization and, 110
Index Seek iterator, 174–175
inverse distribution functions (percentiles). *See also* PERCENTILE_COUNT function; PERCENTILE_DISC function
  about, 68, 71–73
  Median T-SQL solution, 202–204
  optimization of, 129–132
  as ordered set functions, 81, 90–94
ISAM (indexed sequential access method), 7
islands
  Gaps and Islands T-SQL solution, 193–201
  identifying islands problem, 15–19, 195–201
iterative/cursor programming
  Max Concurrent Intervals T-SQL solution, 175–177
  Running Totals T-SQL solution, 158, 162–164
  set-based versus, 6–10

J

joins
  cross, 133–136
  Max Concurrent Intervals T-SQL solution, 174
  ON clause, 161
  Running Totals T-SQL solution, 161–162, 170

K

keys
  Sequences of Keys T-SQL solution, 138–142
  surrogate, 141
Kyte, Tom, 51

L

LAG function
  about, 4, 74–76
  converting to LAST_VALUE function, 122
  expanding all frame rows, 124
  Gaps and Islands T-SQL solution, 194, 199
  NULL return value, 75
LAST_VALUE function
  about, 4, 76–79
  expanding all frame rows, 122, 124
  framing element and, 22
  ordered-set-function form of, 94–96
LEAD function
  about, 4, 74–76
  converting to LAST_VALUE function, 122
  expanding all frame rows, 124
  Gaps and Islands T-SQL solution, 194–195, 199
  NULL return value, 75
Linoff, Gordon, 205
logical query processing
  about, 23–25
  clause ordering and, 57

M

Machanic, Adam, 92, 112
Max Concurrent Intervals solution (T-SQL)
about, 171–173
cursor-based solution, 175–177
performance benchmark, 180
solutions based on window functions, 178–180
traditional set-based solution, 173–175
MAX function
about, 3
expanding all frame rows, 122, 124
usage example, 39, 96–97, 117
Median solution (T-SQL), 202–204
Merge Join iterator, 176, 178
Microsoft SQL Server
hypothetical set functions and, 84–86
logical query processing and, 23
optimization and, 5, 14
ordered set functions and, 81
ordering element support, 21
parallelism considerations, 112–113
StreamInsight feature, 51
TOP option, 134–136, 148, 151–153
WINDOW clause and, 31–32
Microsoft SQL Server 2005
aggregate functions support, 4
ranking functions support, 4, 57
window functions support, 1
Microsoft SQL Server 2012
aggregate functions support, 4
DISTINCT option and, 51
distribution functions support, 4, 68
FILTER clause and, 49
hypothetical set functions and, 84, 87
indexing support, 108
logical query processing and, 23–24
NEXT VALUE FOR function and, 62
OFFSET/FETCH option, 134–136, 144–145, 151–153
offset functions support, 4, 74, 76, 78, 94
RANGE option, 45
rank distribution functions and, 70
window frame-exclusion option, 22, 47
window functions support, 1
MIN function
about, 3
expanding all frame rows, 122, 124
usage example, 96–97
Mode solution (T-SQL), 154–158

N

nested aggregates, 53–57
nested iterations, 166–167
Nested Loops join, 174, 190
Nested Loops join iterator, 115, 118
NEXT VALUE FOR function, 62
NOT EXISTS predicate, 89
NTH_VALUE function
about, 4, 76–79
framing element and, 22
ordered-set-function form of, 94, 96
NTILE function
about, 4, 57, 63–66
optimizing, 110–111

O

OFFSET/FETCH option
Paging T-SQL solution, 144–145
Top-N-per-Group T-SQL solution, 151–153
Virtual Auxiliary Table of Numbers T-SQL solution, 134–136
offset functions. *See also* FIRST_VALUE function; LAG function; LAST_VALUE function; LEAD function; NTH_VALUE function
about, 4, 74–79
carry-along-sort technique, 153, 157
framing and, 22, 119–128
optimizing, 116–128
ordered set functions and, 94–98
ordering and, 74, 119–128
partitioning and, 74
SQL Server support, 4, 74, 76, 78, 94
SQL standard support, 3
OLAP functions, 3
ON clause, 161
optimization of window functions
aggregate functions, 116–128
distribution functions, 128–132
indexing guidelines, 103–108
logical query processing and, 23
offset functions, 116–128
parallel APPLY technique and, 112–115
ranking functions, 108–112
sample data, 101–103
SQL and, 5, 14
ORDER BY clause
about, 7
backward scans and, 107–108

ORDER BY clause, *continued*
Max Concurrent Intervals T-SQL solution, 176
modification statements and, 29
Paging T-SQL solution, 144
presentation ordering and, 58
processing order, 54
ranking functions and, 61
Sequences of Keys T-SQL solution, 142
Virtual Auxiliary Table of Numbers T-SQL solution, 135
window functions support, 8, 23, 25–28
ordered set functions
about, 71, 81
hypothetical set functions, 81–90
inverse distribution functions, 68, 71–73, 81, 90–94
offset functions, 94–98
SQL Server and, 81
string concatenation, 98–99
ordering (sort order)
about, 21–22
aggregate functions and, 22, 36–49, 119–128
based on constants, 109
distribution functions and, 68
elements of sets and, 7
framing and, 36
logical query processing and, 25
Max Concurrent Intervals T-SQL solution, 176
Median T-SQL solution, 203–204
offset functions and, 74, 119–128
POC concept and, 104–106
RANK function, 3
ranking functions and, 21, 58, 109
Running Totals T-SQL solution, 161
Sorting Hierarchies T-SQL solution, 206–210
total ordering, 15
OVER clause
about, 1–3, 10
constants and, 62
usage example, 2, 13, 36

P

Packing Intervals solution (T-SQL)
about, 181–183
solutions based on window functions, 184–193
traditional set-based solution, 183–184
paging
Paging T-SQL solution, 143–145
tiling versus, 64
parallelism
backward scans and, 105
Packing Intervals T-SQL solution, 189
parallel APPLY technique, 112–115, 121–122, 127, 152
Top-N-per-Group T-SQL solution, 152
Parallelism (Distribute Streams) exchange iterator, 115
Parallelism (Gather Streams) exchange iterator, 114
Parallelism (Redistribute Streams) exchange iterator, 114
PARTITION BY clause
about, 20
usage example, 36
partitioning
about, 13–14, 20–21
aggregate functions and, 13–14, 34–36
distribution functions and, 68
offset functions and, 74
Packing Intervals T-SQL solution, 181
parallel APPLY technique and, 113
POC concept and, 104–106
ranking functions and, 58
Running Totals T-SQL solution, 160–161
PERCENTILE_CONT function
about, 72–73
distribution-function form of, 68
Median T-SQL solution, 202
optimizing, 129–131
ordered-set-function form of, 90–94
SQL Server support, 4
PERCENTILE_DISC function
about, 71–72
distribution-function form of, 68
optimizing, 129–132
ordered-set-function form of, 90–92
SQL Server support, 4
percentiles (inverse distribution functions). *See also* PERCENTILE_COUNT function; PERCENTILE_DISC function
about, 68, 71–73
Median T-SQL solution, 202–204
optimization of, 129–132
as ordered set functions, 81, 90–94
PERCENT_RANK function
about, 4, 68–69
hypothetical-set-function form of, 82, 85–87, 89
optimizing, 128–129

performance benchmarks
Max Concurrent Intervals T-SQL solution, 180
Running Totals T-SQL solution, 169–171
Pivoting solution (T-SQL), 148–151
POC indexes
about, 104–105
Running Totals T-SQL solution, 161
Sort iterator and, 109
Top-N-per-Group T-SQL solution, 151–152

Q

QUALIFY clause, 30
Query Options dialog box, 136

R

RANGE clause
about, 22
aggregate functions and, 37
CURRENT ROW option, 43–47, 77
UNBOUNDED option, 43–46, 77
window frame extent part and, 43–47
window frame units part and, 120
rank distribution functions. *See also* CUME_DIST function; PERCENT_RANK function
about, 68–71, 90
as hypothetical set functions, 82–90
optimization of, 128–129
RANK function
about, 4, 57, 66–67
calculating percentile rank, 70
determinism and, 67
hypothetical-set-function form of, 82–84, 87–89
Mode T-SQL solution, 156
optimizing, 111–112
ordering element and, 21
partitioning element and, 20
Removing Duplicates T-SQL solution, 147
usage example, 3, 9, 28
ranking functions. *See also* DENSE_RANK function; NTILE function; RANK function; ROW_NUMBER function
about, 4, 57
as hypothetical set functions, 82–90
optimization of, 108–112
ordering and, 21, 58, 109
partitioning and, 58
SQL Server support, 4, 57
SQL standard support, 3
RDBMSs (relational database management systems), 6
relational database management systems (RDBMSs), 6
relational model
about, 6–7
ordering and, 8
Removing Duplicates solution (T-SQL), 145–148
Rincón, Eladio, 113
ROW_NUMBER function
about, 4, 57, 58–63
COUNT function and, 59–60
determinism and, 60–64
distinct aggregates and, 52
Gaps and Islands T-SQL solution, 197
islands problem, 18–19
Max Concurrent Intervals T-SQL solution, 178–180
Median T-SQL solution, 202
Mode T-SQL solution, 154–156
modification statements and, 29
optimizing, 109–110
Packing Intervals T-SQL solution, 184
Paging T-SQL solution, 143–145
Pivoting T-SQL solution, 149
Removing Duplicates T-SQL solution, 145–147
Running Totals T-SQL solution, 166
Sequences of Keys T-SQL solution, 139, 141–142
Sorting Hierarchies T-SQL solution, 206–210
Top-N-per-Group T-SQL solution, 152
usage example, 27, 92–93
Virtual Auxiliary Table of Numbers T-SQL solution, 135
row pattern recognition, 51
ROWS clause
about, 22
converting RANGE option to, 120–121
CURRENT ROW option, 37–38
UNBOUNDED FOLLOWING option, 37–38, 77
UNBOUNDED PRECEDING option, 37–38
window frame extent part and, 37–43
Running Totals solution (T-SQL)
about, 158–160
CLR-based solution, 164–166
cursor-based solution, 162–164
multirow UPDATE with variables, 167–169

Running Totals solution (T-SQL), *continued*
nested iterations, 166–167
performance benchmark, 169–171
set-based solutions using subqueries or joins, 161–162
set-based solution using window functions, 160–161

S

scheduling applications, 137
Schulz, Brad, 107
Segment iterator
computing cumulative values, 127
computing ranking functions, 108
distribution functions and, 129–130
expanding all frame rows, 123
fast-track case, 120
parallel APPLY technique, 114
SELECT clause
CTE filtering example, 28–30
logical query processing and, 23–25
modification statements and, 29
processing order, 54
Removing Duplicates T-SQL solution, 146–147
Running Totals T-SQL solution, 169
window functions support, 23, 25–28
Sequence Project iterator
computing cumulative values, 127
computing ranking functions, 108, 111, 112
distribution functions and, 129–130
expanding all frame rows, 123
fast-track case, 120
parallel APPLY technique, 114
sequences
about, 62
Date and Time Values T-SQL solution, 137–138
Gaps and Islands T-SQL solution, 193–201
Sequences of Date and Time Values T-SQL solution, 137–138
Sequences of Keys T-SQL solution, 138–142
set-based programming
iterative/cursor versus, 6–10
Max Concurrent Intervals T-SQL solution, 173–175
Packing Intervals T-SQL solution, 183–184
Running Totals T-SQL solution, 158–162
sets. *See also* ordered set functions
about, 7
logical query processing and, 23
ordering elements of, 7, 81
Sorting Hierarchies solution (T-SQL), 206–210
Sort iterator
about, 104–105
parallel APPLY technique and, 114
POC index and, 109
sort order (ordering)
about, 21–22
aggregate functions and, 22, 36–49, 119–128
based on constants, 109
distribution functions and, 68
elements of sets and, 7
framing and, 36
logical query processing and, 25
Max Concurrent Intervals T-SQL solution, 176
Median T-SQL solution, 203–204
offset functions and, 74, 119–128
POC concept and, 104–106
RANK function, 3
ranking functions and, 21, 58, 109
Running Totals T-SQL solution, 161
Sorting Hierarchies T-SQL solution, 206–210
total ordering, 15
spreading element (pivoting technique), 148
SQL standard
about, 3
additional resources, 3
all-at-once concept, 26
optimization and, 5
relational model and, 6
row pattern recognition, 51
SQL:1999 standard, 3
SQL:2003 standard, 3
SQL:2008 standard
filtering and, 50
window functions support, 3
SQL:2011 standard, 3
SQLDataReader class, 164, 166
SQL Server (Microsoft)
hypothetical set functions and, 84–86
logical query processing and, 23
optimization and, 5, 14
ordered set functions and, 81
ordering element support, 21
parallelism considerations, 112–113
StreamInsight feature, 51

TOP option, 134–136, 148, 151–153
WINDOW clause and, 31–32
SQL Server 2005 (Microsoft)
aggregate functions support, 4
ranking functions support, 4, 57
window functions support, 1
SQL Server 2012 (Microsoft)
aggregate functions support, 4
DISTINCT option and, 51
distribution functions support, 4, 68
FILTER clause and, 49
hypothetical set functions and, 84, 87
indexing support, 108
logical query processing and, 23–24
NEXT VALUE FOR function and, 62
OFFSET/FETCH option, 134–136, 144–145, 151–153
offset functions support, 4, 74, 76, 78, 94
RANGE option, 45
rank distribution functions and, 70
window frame-exclusion option, 22, 47
window functions support, 1
SQL Server Management Studio (SSMS), 126
SQL windowing
about, 1
background of window functions, 2–15
elements of window functions, 19
QUALIFY clause and, 30
query elements supporting window functions, 23–30
reusing window definitions, 31–32
solutions using window functions, 15–19
SSMS (SQL Server Management Studio), 126
STATISTICS IO option, 121, 125–126
STDEV function
computing cumulative values, 126
expanding all frame rows, 122
STDEVP function
computing cumulative values, 126
expanding all frame rows, 122
stored procedures
Running Totals T-SQL solution, 164–165
Sequences of Keys T-SQL solution, 140
Stream Aggregate iterator
about, 119
computing cumulative values, 127
distribution functions and, 129, 131–132
expanding all frame rows, 123
fast-track case, 120
MAX function and, 117
string concatenation
carry-along-sort technique and, 153
concatenation operator (+), 150
CONCAT function, 150
ordered set functions and, 81, 98–99
STUFF function, 99
subqueries
islands problem, 16–17
Running Totals T-SQL solution, 161–162, 170
as window function alternative, 12–14
SUM function
about, 3
computing cumulative values, 126–127
expanding all frame rows, 122–123
framing element and, 22
grouped aggregates and, 54–55
optimizing, 132
Packing Intervals T-SQL solution, 191–192
RANGE option, 22
ROWS option, 22
Running Totals T-SQL solution, 160
surrogate keys, 141

T

tables
applying operators, 14
cross joining, 133–136
Gaps and Islands T-SQL solution, 193–201
logical query processing and, 23
Max Concurrent Intervals T-SQL solution, 171–180
Median T-SQL solution, 202–204
Mode T-SQL solution, 154–158
modification statements and, 29
Packing Intervals T-SQL solution, 181–193
Pivoting T-SQL solution, 148–151
Removing Duplicates T-SQL solution, 145–148
Running Totals T-SQL solution, 158–171
Sequences of Keys T-SQL solution, 138–142
sets and, 7
Sorting Hierarchies T-SQL solution, 206–210
Top-N-per-Group T-SQL solution, 151–154, 157
Virtual Auxiliary Table of Numbers T-SQL solution, 133–136
Table Spool iterator
distribution functions and, 130
ranking functions and, 111

threads
DOP considerations, 114
moving data between, 114
tiling versus paging, 64
time and date values
Gaps and Islands T-SQL solution, 193–201
Sequences of Date and Time Values T-SQL solution, 137–138
Top-N-per-Group solution (T-SQL), 151–154, 157
TOP option
Removing Duplicates T-SQL solution, 148
Top-N-per-Group T-SQL solution, 151–153
Virtual Auxiliary Table of Numbers T-SQL solution, 134–136
total ordering, 15
totals, calculating. *See* Running Totals solution (T-SQL)
T-SQL
approach to querying tasks, 6–10
Conditional Aggregate solution, 204–206
Gaps and Islands solution, 193–201
islands problem, 17–18
Max Concurrent Intervals solution, 171–180
Median solution, 202–204
Mode solution, 154–158
modification statements and, 29
Packing Intervals solution, 181–193
Paging solution, 143–145
Pivoting solution, 148–151
Removing Duplicates solution, 145–148
Running Totals solution, 158–171
Sequences of Date and Time Values solution, 137–138
Sequences of Keys solution, 138–142
Sorting Hierarchies solution, 206–210
Top-N-per-Group solution, 151–154, 157
Virtual Auxiliary Table of Numbers solution, 133–136

U

UDAs (user-defined aggregates), 81
UNBOUNDED option
RANGE clause, 43–46, 77
ROWS clause, 37–38, 77
UNBOUNDED FOLLOWING option, 37–38, 77
UNBOUNDED PRECEDING option
RANGE clause, 77
ROWS clause, 37–38
Running Totals T-SQL solution, 160
window frame extent part, 119–123
UPDATE clause
modification statements and, 29
Running Totals T-SQL solution, 167–169
Sequences of Keys T-SQL solution, 139
user-defined aggregates (UDAs), 81

V

VAR function
computing cumulative values, 126
expanding all frame rows, 122
VARP function
computing cumulative values, 126
expanding all frame rows, 122
VertiPaq technology, 108
Virtual Auxiliary Table of Numbers solution (T-SQL), 133–136

W

WHERE clause
filtering and, 117
processing order, 54
Running Totals T-SQL solution, 161
usage example, 27
window
about, 2
defined by OVER clause, 3
frame-exclusion option, 22
WINDOW clause, 31–32
window-frame-exclusion option, 22, 37, 47–49
window frame extent part
about, 37
RANGE clause and, 43–47
ROWS clause and, 37–43
UNBOUNDED PRECEDING option, 119–122
window frame units part
about, 37
RANGE clause, 120
window functions. *See also* specific window functions
about, 1–2
background of, 2–5
drawbacks of alternatives to, 11–15
elements of, 19
Max Concurrent Intervals T-SQL solution, 178–180

optimization of, 5
Packing Intervals T-SQL solution, 184–193
QUALIFY clause and, 30
query elements supporting, 23–30
reusing window definitions, 31–32
Running Totals T-SQL solution, 160–161
solutions using, 15–19
T-SQL approach to, 6–10
windowing (SQL)
about, 1
background of window functions, 2–15
elements of window functions, 19
QUALIFY clause and, 30
query elements supporting window functions, 23–30
reusing window definitions, 31–32
solutions using window functions, 15–19
Window Spool iterator
computing cumulative values, 127
distribution functions and, 129
expanding all frame rows, 122–124
fast-track case, 119–120
window_spool_ondisk_warning extended event, 126
WITHIN GROUP clause, 68, 71

About the Author

ITZIK BEN-GAN is a mentor with and co-founder of SolidQ. A SQL Server Microsoft MVP since 1999, Itzik has taught 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 Pro* as well as articles and white papers for MSDN and *The SolidQ Journal*. Itzik's speaking engagements include Tech-Ed, SQL PASS, SQL Server Connections, presentations to various SQL Server user groups, and SolidQ events. Itzik is a subject-matter expert within SolidQ for its T-SQL related activities. He authored SolidQ's Advanced T-SQL and T-SQL Fundamentals courses and delivers them regularly worldwide.

What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Tell us how well this book meets your needs—what works effectively, and what we can do better. Your feedback will help us continually improve our books and learning resources for you.

Thank you in advance for your input!

Microsoft® Press

SurvPage_Corp_02.indd 1 5/19/2011 4:18:12 PM