Expert T-SQL Window Functions in SQL Server 2019

The Hidden Secret to Fast Analytic and Reporting Queries

Second Edition

Kathi Kellenberger
Clayton Groom
Ed Pollack

Apress®

Expert T-SQL Window Functions in SQL Server 2019

The Hidden Secret to Fast Analytic and Reporting Queries

Second Edition

Kathi Kellenberger
Clayton Groom
Ed Pollack

Apress®

Expert T-SQL Window Functions in SQL Server 2019

Kathi Kellenberger
Edwardsville, IL, USA

Clayton Groom
Smithton, IL, USA

Ed Pollack
Albany, NY, USA

ISBN-13 (pbk): 978-1-4842-5196-6
ISBN-13 (electronic): 978-1-4842-5197-3
https://doi.org/10.1007/978-1-4842-5197-3

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Jonathan Gennick
Development Editor: Laura Berendson
Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484251966. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

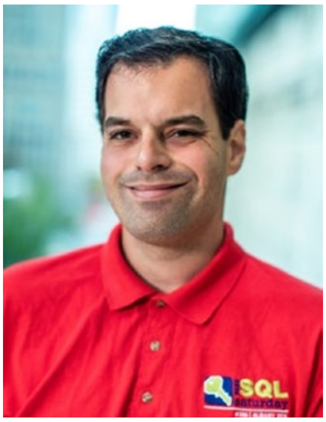
This book is dedicated to the memory of Larry Toothman.

Table of Contents

About the Authors

Kathi Kellenberger is a data platform MVP and the editor of Simple Talk at Redgate Software. She has worked with SQL Server for over 20 years. She is also coleader of the PASS Women in Technology Virtual Group and an instructor at LaunchCode. In her spare time, Kathi enjoys spending time with family and friends, singing, and cycling.

Clayton Groom is a data warehouse and analytics consultant at Clayton Groom, LLC. He has worked with SQL Server for 25 years. His expertise lies in designing and building data warehouse and analytic solutions on the Microsoft technology stack, including Power BI, SQL Server, Analysis Services, Reporting Services, and Excel.



Ed Pollack has over 20 years of experience in database and systems administration and architecture, developing a passion for performance optimization and making things go faster. He has spoken at many SQL Saturdays, 24 Hours of PASS, and PASS Summits and has coordinated SQL Saturday Albany since its inception in 2014.

About the Technical Reviewer

Rodney Landrum went to school to be a poet and a writer. And then he graduated, so that dream was crushed. He followed another path, which was to become a professional in the fun-filled world of information technology. He has worked as a systems engineer, UNIX and network admin, data analyst, client services director, and finally as a database administrator. The old hankering to put words on paper, while paper still existed, got the best of him, and in 2000 he began writing technical articles, some creative and humorous, some quite the opposite. In 2010, he wrote ***SQL Server Tacklebox***, a title his editor disdained, but a book closest to the true creative potential he sought; he still yearned to do a full book without a single screenshot, which he accomplished in 2019 with his first novel, *Chronicles of Shameus*. He currently works from his castle office in Pensacola, FL, as a senior DBA consultant for Ntirety, a division of Hostway/HOSTING.

Foreword

SQL was developed in the 1970s and became standardized through ANSI-approved committees as a formal standard starting in 1986.[1] Today in 2019, SQL has become the most widely used declarative language. Along the way, window functions have come to be an important part of that standard. ANSI does not make standards but plays an important role in documenting and preserving them. The individual software vendors voluntarily decide to comply, and it's the work of the authors of books like this one to explain SQL use in practical terms.

In my own career in data science and advanced analytics, window functions have been an important part of several key projects in the past few years. Several years ago, I made a YouTube video for a user group based on the earlier edition of this book. Since then, as a career architect at Microsoft, I have advised the application for data science. In one project, the input of about 20 features was not yielding adequate results: using window functions, a team under my leadership (and yes, direct coding) quickly grew that number to over 1,000. More than numeric growth, the accuracy rates improved, and on the business story, the organization is saving millions of dollars annually for their question. In the past month, I have encountered an unrelated new project, and a similar story is there: a time-series type of data set and an opportunity to grow from under 20 features to a number much larger.

One wonders whether automated machine learning technologies would make such combinations on their own, and I'm skeptical. Making a robust set of features from window functions requires not just time-series considerations but also clustering knowledge based on knowing the data domain. Even if automated technologies make great progress in this topic, I anticipate the need for any data scientist to have simple knowledge of these functions for the more typical data science investigation which has only a few features and low number of observations.

SQL is central to on-premise and cloud database technologies – and in the data science world, many use Apache Spark (part of SQL Server 2019 and so many other data technologies). This reach into advanced analytics is yet another reason why this topic is

[1]See https://blog.ansi.org/2018/10/sql-standard-iso-iec-9075-2016-ansi-x3-135/

an expert-level subject in the SQL language. The mainstream applications extend from any business analytics SQL query and even into supporting advanced analytics and machine learning algorithms.

Over the years, it's been an honor to individually know Kathi Kellenberger and Clayton Groom as respected peers and professionals and to see how they have each become important leaders to the technical user community through many presentations (for which they typically volunteer their own time) and through the creation and now revision of this book. In this revision, Ed Pollack has applied material on baseball statistics, illustrating that not every time series is about money. It's not enough to have a standard written, but one needs to have expert coaches to explain how these functions describe an approach for business analytics. This book has rich examples and altogether provides a clear path into one of the most mathematically complex and yet practically useful aspects of the SQL language.

Mark Tabladillo, Ph.D.
Cloud Solution Architect, Microsoft

Acknowledgments

The first edition of this book would not have been written except for the suggestion of one of my friends in the data platform community, Larry Toothman. Sadly, Larry passed away shortly after the book was published and before I could get a copy to him. Larry was just getting started with presenting at events and being more involved in the community during the last couple of years of his life. Who knows what he might have accomplished if things had turned out differently. Thanks to Larry's inspiration, people around the world will learn about windowing functions.

They say it takes a village to raise a child, and the same might be true for a book. I would like to thank Jonathan Gennick and Jill Balzano for their help and guidance. There are probably many people at Apress who had a hand in this project that I'll never meet, and I would like to thank them as well.

Clayton Groom and Ed Pollack each wrote about their real-world experience using windowing functions. In each case, the idea for their chapter came from my running into each of them at user group meetings and just talking about my project. Their contributions definitely make this a better and more enjoyable book for you.

Thanks to Rodney Landrum for doing a great job on the technical review and to Mark Tabladillo for the wonderful foreword.

Thank you to my family, especially my husband, Dennis, who takes care of just about everything around the house. He makes my life so much easier when I take on big projects like this.

Finally, thank you dear reader for learning about windowing functions from this book. I hope that you enjoy it and can apply the things you learn right away. I would love to hear from you at events, so don't be shy!

Introduction

Several years ago, I would create a user group presentation for each new version of SQL Server about the new T-SQL features. There was so much to say in 2012 that I decided to build a presentation on just the windowing functions introduced that year. Eventually, I had so much material that it turned into two sessions. Over the years, I have probably presented this information at least 50 times at events around the United States and the United Kingdom. Despite that, most people still are not using windowing functions because they haven't heard about them or do not realize the benefits.

What's in This Book?

This book covers each type of windowing function beginning with the ranking functions introduced with SQL Server 2005 through the statistical functions introduced in 2012. Each chapter explains how to use the functions along with any options and provides a few simple examples of how to use them. Unfortunately, the last time that Microsoft added any new windowing functions was 2012, but there have been some performance improvements more recently. One chapter is dedicated to the performance of windowing functions.

Finally, the last two chapters in the book cover some real-world examples. In Chapter 9, you'll learn how to analyze a large data set, over 100 years of baseball statistics. Chapters 10 and 11 show how windowing functions can be used in data warehouse calculations instead of building a cube.

Intended Audience

This book is meant for people who already have good T-SQL skills. They know how to join tables, use subqueries and CTEs, and write aggregate queries. Despite these skills, they occasionally run into problems that are not easy to solve in a set-based manner. Without windowing functions, some of these problems can only be solved by using cursors or expensive triangular joins. By using the concepts taught in this book, your T-SQL skills will improve to the next level. Once you start using windowing functions, you'll find even more reasons to learn them.

Contacting the Author

Great care was taken to ensure that the information presented is correct, but sometimes readers come up with a better way to write a query or find an error. You can contact me at `kathi.kellenberger@outlook.com` with any comments or questions.

CHAPTER 1

Looking Through the Window

SQL Server is a powerful database platform with a versatile query language called T-SQL. The most exciting T-SQL enhancement over the years, in my opinion, is the window functions. Window functions enable you to solve query problems in new, easier ways and with better performance most of the time over traditional techniques. They are a great tool for analytics. You may hear these called “windowing” or “windowed” functions as well. The three terms are synonymous when talking about this feature.

After the release of SQL Server 2000, SQL Server enthusiasts waited 5 long years for the next version of SQL Server to arrive. Microsoft delivered an entirely new product with SQL Server 2005. This version brought SQL Server Management Studio, SQL Server Integration Services, snapshot isolation, and database mirroring. Microsoft also enhanced T-SQL with many great features, such as common table expressions (CTEs). The most exciting T-SQL enhancement of all with 2005 was the introduction of window functions.

That was just the beginning. Window functions are part of the standard ANSI SQL specification beginning with ANSI SQL2003. More functionality according to the standard was released with version 2012 of SQL Server. In 2019, Microsoft gave some of the window functions a performance boost with batch mode processing, a feature once reserved for column store indexes. You’ll see how this performance feature works in Chapter 8. Even now, the functionality falls short of the entire specification, so there is more to look forward to in the future.

This chapter provides a first look at two T-SQL window functions, `LAG` and `ROW_NUMBER`. You will learn just what the window is and how to define it with the `OVER` clause. You will also learn how to divide the windows into smaller sections called partitions.

K. Kellenberger et al., *Expert T-SQL Window Functions in SQL Server 2019*,
https://doi.org/10.1007/978-1-4842-5197-3_1

Discovering Window Functions

Window functions do not let you do anything that was impossible to do with earlier functionality, and they have nothing to do with the Microsoft Windows API. Using previously available methods, such as self-joins, correlated subqueries, and cursors, you can solve just about any T-SQL problem if you work at it long and hard enough. The main benefit of window functions is the ease with which you can solve these tricky queries. Most of the time, you also realize a big boost in performance over the older methods. You can often use a window function to change a solution involving many statements or subqueries to one easier statement.

I like to divide window functions into several categories that do not exactly match up with the way Microsoft defines them: ranking functions, window aggregates, accumulating window aggregates, offset functions, and statistical functions. (Microsoft refers to the four offset and four statistical functions as “analytic” functions.) You can use these functions to assign a rank to each row, calculate summary values without grouping, calculate running totals, include columns from different rows in your results, and calculate percentages over a group. You’ll learn about these functions as you read this book.

My favorite T-SQL function which also happens to be a window function is called `LAG`. It is one of the offset functions, which you will learn about in Chapter 6. `LAG` allows you to include columns from different rows in your results. Using `LAG` is easier and performs better than older methods that do the same thing.

Within the same year (just a few months apart), two different people approached me for help with essentially the same problem: using data from the stock market, how can one compare the closing price of a stock from one day to the next? The traditional solution requires that each row of the data be joined to the prior row to get the closing price from the previous day. By using the `LAG` function, the solution is not only simpler to write, it also performs much better.

Note If you would like to follow along with this example, a sample script to create the StockAnalysisDemo database and generated stock market data can be found along with the code for this chapter on the Apress site.

For a quick look at how to solve this problem first by using one of the traditional methods and then by using `LAG`, review and run Listing 1-1.

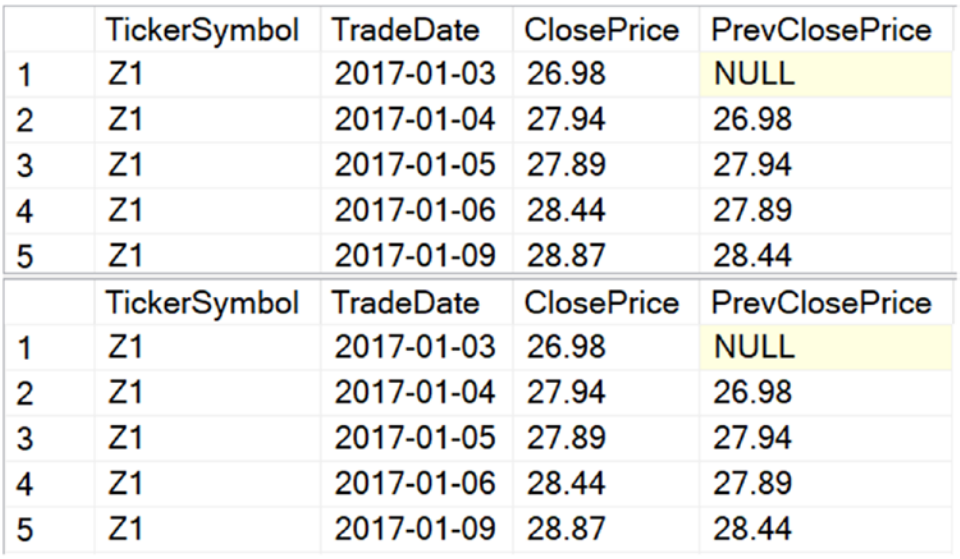
Listing 1-1. Two Approaches to Solving the Stock Market Problem

```
USE StockAnalysisDemo;
GO
--1-1.1 Using a subquery
SELECT TickerSymbol, TradeDate, ClosePrice,
    (SELECT TOP(1) ClosePrice
    FROM StockHistory AS SQ
    WHERE SQ.TickerSymbol  = OQ.TickerSymbol
        AND SQ.TradeDate < OQ.TradeDate
    ORDER BY TradeDate DESC) AS PrevClosePrice
FROM StockHistory AS OQ
ORDER BY TickerSymbol, TradeDate;

--1-1.2 Using LAG
SELECT TickerSymbol, TradeDate, ClosePrice,
    LAG(ClosePrice) OVER(PARTITION BY TickerSymbol
           ORDER BY TradeDate) AS PrevClosePrice
FROM StockHistory
ORDER BY TickerSymbol, TradeDate;
```

The partial results are shown in Figure 1-1. Since the data is randomly generated, the values of ClosePrice and PrevClosePrice in the image will not match your values. Query 1 uses a correlated subquery, the old method, to select one `ClosePrice` for every outer row. By joining the `TickerSymbol` from the inner query to the outer query you ensure that you are not comparing two different stocks. The inner and outer queries are also joined by the `TradeDate`, but the `TradeDate` for the inner query must be less than the outer query to make sure you get the prior day. The inner query must also be sorted to get the row that has the latest data but still less than the current date. This query took over a minute to run on my laptop, which has 16GB of RAM and is using SSD storage. Almost 700,000 rows were returned.

Query 2 uses the window function `LAG` to solve the same problem and produces the same results. Don't worry about the syntax at this point; you will be an expert by the end of this book. The query using `LAG` took just 13 seconds to run on my laptop.



	TickerSymbol	TradeDate	ClosePrice	PrevClosePrice
1	Z1	2017-01-03	26.98	NULL
2	Z1	2017-01-04	27.94	26.98
3	Z1	2017-01-05	27.89	27.94
4	Z1	2017-01-06	28.44	27.89
5	Z1	2017-01-09	28.87	28.44

	TickerSymbol	TradeDate	ClosePrice	PrevClosePrice
1	Z1	2017-01-03	26.98	NULL
2	Z1	2017-01-04	27.94	26.98
3	Z1	2017-01-05	27.89	27.94
4	Z1	2017-01-06	28.44	27.89
5	Z1	2017-01-09	28.87	28.44

Figure 1-1. *Partial results of the stock market problem*

By just looking at the code in Listing 1-1, you can see that Query 2 using LAG is much simpler to write, even though you may not understand the syntax just yet. It also runs much faster because it is just reading the table once instead of once per row like Query 1. As you continue reading this book and running the examples, you will learn how window functions like LAG will make your life easier and your customers happier!

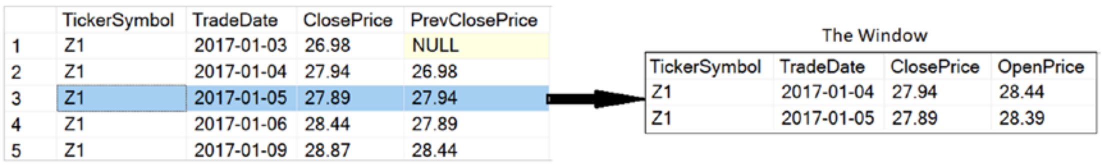
Thinking About the Window

Window functions are different than regular functions because they operate over a set of rows, also called a *window*. This may sound similar to how aggregate functions work. Aggregate functions, such as SUM and AVG, operate on groups of rows and provide summary values. When you write an aggregate query, you lose the detail columns except for those in the GROUP BY clause.

When adding a GROUP BY clause, instead of returning a summary along with all the rows, you will see a summary row, one row for each unique set of GROUP BY columns. For example, to get a count of the all the rows using an aggregate query, you must leave out the other columns. Once you add columns into the SELECT and GROUP BY, you get a count for each unique grouping, not the entire set of results.

Queries with window functions are much different than traditional aggregate queries. There are no restrictions to the columns that appear in the SELECT list, and no GROUP BY clause is required. You can also add window functions to aggregate queries, and that will be discussed in Chapter 3. Instead of summary rows being returned, all the details are returned and the result of the expression with the window function is included as just another column. In fact, by using a window function to get the overall count of the rows, you could still include all of the columns in the table.

Imagine looking through a window to see a specific set of rows while your query is running. You have one last chance to perform an operation, such as grabbing one of the columns from another row. The result of the operation is added as an additional column. You will learn how window functions really work as you read this book, but the idea of looking through the window has helped me understand and explain window functions to audiences at many SQL Server events. Figure 1-2 illustrates this concept.



	TickerSymbol	TradeDate	ClosePrice	PrevClosePrice
1	Z1	2017-01-03	26.98	NULL
2	Z1	2017-01-04	27.94	26.98
3	Z1	2017-01-05	27.89	27.94
4	Z1	2017-01-06	28.44	27.89
5	Z1	2017-01-09	28.87	28.44

The Window

TickerSymbol	TradeDate	ClosePrice	OpenPrice
Z1	2017-01-04	27.94	28.44
Z1	2017-01-05	27.89	28.39

Figure 1-2. *Looking through the window to perform an operation on a set of rows*

The window is not limited to the columns found in the SELECT list of the query. For example, if you take a look at the StockHistory table, you will see that there is also an OpenPrice column. The OpenPrice from one day is not the same as the ClosePrice from the previous day. If you wanted to, you could use LAG to include the previous OpenPrice in the results even though it is not included in the SELECT list originally.

In the stock history example using LAG, each row has its own window where it finds the previous close price. When the calculation is performed on the third row of the data, the window consists of the second and third rows. When the calculation is performed on the fourth row, the window consists of the third and fourth rows.

What would happen if the rows for 2017-12-02 were removed from the query by a WHERE clause? Does the window contain filtered-out rows? The answer is “No,” which brings up two very important concepts to understand when using window functions: where window functions may be used in the query and the logical order of operations.

Window functions may only be used in the SELECT list and ORDER BY clause. You cannot filter or group on window functions. In situations where you must filter or group on the results of a window function, the solution is to separate the logic. You could use a temp table, derived table subquery, or a CTE and then filter or group in the outer query.

Window functions operate after the `FROM`, `WHERE`, `GROUP BY`, and `HAVING` clauses. They operate before the `TOP` and `DISTINCT` clauses are evaluated. You will learn more about how `DISTINCT` and `TOP` affect queries with window functions in the "Uncovering Special Case Windows" section later in this chapter.

The window is defined by the `OVER` clause. Notice in Query 2 of Listing 1-1 that the `LAG` function is followed by an `OVER` clause. Each type of window function has specific requirements for the `OVER` clause. The `LAG` function must have an `ORDER BY` expression and may have a `PARTITION BY` expression.

Understanding the OVER Clause

One thing that sets window functions apart is the `OVER` clause, which defines the window or set. With one exception I'll explain in Chapter 7, window functions will have an `OVER` clause, and learning how to use the `OVER` clause is required to understand window functions. In some cases, the `OVER` clause will be empty. You will see empty `OVER` clauses in Chapter 3 when working with window aggregate functions.

Note There is one situation in which you will see the OVER keyword in a query not following a window function, and that is with the sequence object. The sequence object, introduced with SQL Server 2008, is a bucket containing incrementing numbers often used in place of an identity column.

For any type of expression in the `SELECT` list of a query, a calculation is performed for each row in the results. For example, if you had a query with the expression `Col1 + Col2`, those two columns would be added together once for every row returned. A calculation is performed for row 1, row 2, row 3, and so on. Expressions with window functions must also be calculated once per row. In this case, however, the expressions operate over a set of rows that can be different for each row where the calculation is performed.

The `OVER` clause determines which rows make up the window. The `OVER` clause has three possible components: `PARTITION BY`, `ORDER BY`, and the frame. The `PARTITION BY` expression divides up the rows, and it's optional depending on what you are trying to accomplish. The `ORDER BY` expression is required for some types of window functions.

Where it is used, it determines the order in which the window function is applied. Finally, the frame is used for some specific types of window functions to provide even more granularity. You'll learn about framing in Chapter 5.

Many T-SQL developers and database professionals have used the ROW_NUMBER function. They may not have even realized that this is one of the window functions. There are many situations where adding a row number to the query is a step along the way to solving a complex query problem.

ROW_NUMBER supplies an incrementing number, starting with one, for each row. The order in which the numbers are applied is determined by the columns specified in the ORDER BY expression, which is independent of an ORDER BY clause found in the query itself. Run the queries in Listing 1-2 to see how this works.

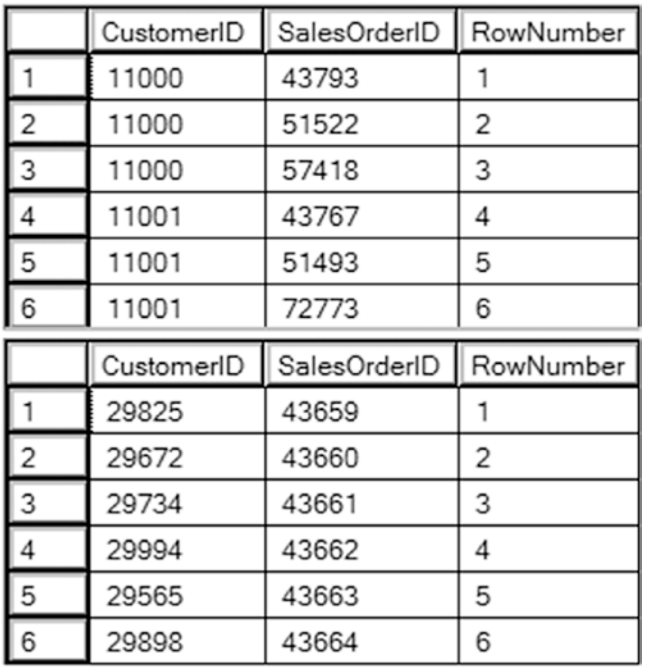
Listing 1-2. Applying the Row Numbers to Different Columns

```
USE AdventureWorks;
GO
--1-2.1 Row numbers applied by CustomerID
SELECT CustomerID, SalesOrderID,
    ROW_NUMBER() OVER(ORDER BY CustomerID) AS RowNumber
FROM Sales.SalesOrderHeader;

--1-2.2 Row numbers applied by SalesOrderID
SELECT CustomerID, SalesOrderID,
    ROW_NUMBER() OVER(ORDER BY SalesOrderID) AS RowNumber
FROM Sales.SalesOrderHeader;
```

Note The AdventureWorks and AdventureWorksDW databases are used in many of the examples throughout this book. You can use any version starting with 2014 or later to follow along, and the 2017 version was the latest available at the time of this writing. Just be sure to adjust the USE statement when it's included to fit your version of the databases.

The OVER clause follows the ROW_NUMBER function. Inside the OVER clause, you will see ORDER BY followed by one or more columns. The difference between Queries 1 and 2 is just the ORDER BY expression within the OVER clause. Notice in the partial results shown in Figure 1-3 that the row numbers end up applied in the order of the column found in the ORDER BY expression of the OVER clause, which is also the order that the data is returned. Since the data must be sorted to apply the row numbers, it is easy for the data to stay in that order, but it is not guaranteed. The only way to ever actually guarantee the order of the results is to add an ORDER BY to the query.



	CustomerID	SalesOrderID	RowNumber
1	11000	43793	1
2	11000	51522	2
3	11000	57418	3
4	11001	43767	4
5	11001	51493	5
6	11001	72773	6

	CustomerID	SalesOrderID	RowNumber
1	29825	43659	1
2	29672	43660	2
3	29734	43661	3
4	29994	43662	4
5	29565	43663	5
6	29898	43664	6

Figure 1-3. *Partial results of using ROW_NUMBER with different OVER clauses*

If the query itself has an ORDER BY clause, it can be different than the ORDER BY within OVER. Listing 1-3 demonstrates this.

Listing 1-3. Using ROW_NUMBER with a Different ORDER BY in the OVER Clause

```
--1-3.1 Row number with a different ORDER BY
SELECT CustomerID, SalesOrderID,
    ROW_NUMBER() OVER(ORDER BY CustomerID) AS RowNumber
FROM Sales.SalesOrderHeader
ORDER BY SalesOrderID;
```