Microsoft

The Definitive Guide to DAX

Business intelligence with Microsoft Excel, SQL Server Analysis Services, and Power BI

Marco Russo and Alberto Ferrari

Sample files on the web

Microsoft

The Definitive Guide to DAX: Business intelligence with Microsoft Excel, SQL Server Analysis Services, and Power BI

Marco Russo and Alberto Ferrari

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

Library of Congress Control Number: 2014955069
ISBN: 978-0-7356-9835-2

Printed and bound in the United States of America.

First Printing

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

Acquisitions Editor: Devon Musgrave
Developmental Editor: Carol Dillingham
Project Editor: Carol Dillingham
Editorial Production: Christian Holdener, S4Carlisle Publishing Services
Technical Reviewer: Gerhard Brueckl and Andrea Benedetti; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Copyeditor: Leslie Phillips
Indexer: Maureen Johnson, MoJo's Indexing and Editorial Services
Cover: Twist Creative • Seattle

We dedicate this book to the Picasso team.

Contents at a glance

Table of contents

Foreword

Maybe you don't know our names. We spend our days writing the code for the software you use in your daily job: We are part of the development team of Power BI, SQL Server Analysis Services, and . . . yes, we are among the authors of the DAX language and the VertiPaq engine.

The language you are going to learn using this book is our creature. We spent years working on this language, optimizing the engine, finding ways to improve the optimizer, and trying to build DAX into a simple, clean, and sound language to make your life as a data analyst easier and more productive.

But hey, this is intended to be the foreword of a book, no more words about us! Why are we writing a foreword for a book published by Marco and Alberto, the SQLBI guys? Well, because when you start learning DAX, it is a matter of a few clicks and searches on the web before you find articles written by them. You start reading their papers, learning the language, and hopefully appreciate our hard work. Having met them many years ago, we have great admiration for their deep knowledge of SQL Server Analysis Services. When the DAX adventure started, they were among the first to learn and adopt this new engine and language.

The articles, papers, and blog posts they publish and share on the web became the source of learning for thousands of people. We write the code, but we do not spend much time teaching how to use it; they are the ones who spread knowledge about DAX.

Alberto and Marco's books are among the best sellers on this topic and now, with this new guide to DAX, they truly created a milestone publication about the language we author and love. We write the code, they write the books, and you learn DAX, providing unprecedented analytical power to your business. This is what we love: working all together as a team—we, they, and you—to get better insights from data.

Marius Dumitru, Architect, Power BI CTO's Office

Cristian Petculescu, Chief Architect of Power BI

Jeffrey Wang, Principal Software Engineer Manager

Introduction

We previously wrote about DAX many times: in books about Power Pivot and SSAS Tabular, in blog posts, articles, white papers, and finally in a book dedicated to DAX patterns. So why should we write (and, hopefully, you read) yet another book about DAX? Is there really so much to learn about this language? Of course, we think the answer is definitely yes.

When you write a book, the first thing that the editor wants to know is the number of pages. There are very good reasons why this is important: price, management, allocation of resources, and so on. At the end, nearly everything in a book goes back to the number of pages. As authors, this is somewhat frustrating. In fact, whenever we wrote a book, we had to carefully allocate space to the description of the product (either Power Pivot for Microsoft Excel or SSAS Tabular) and to the DAX language. This always left us with the bitter taste of not having enough pages to describe all we wanted to teach about DAX. After all, you cannot write 1,000 pages about Power Pivot; a book of such a size would be intimidating for anybody.

Thus, for some years we wrote about SSAS Tabular and Power Pivot, and we kept the project of a book completely dedicated to DAX in a drawer. Then we opened the drawer and decided to avoid choosing what to include in the next book: We wanted to explain everything about DAX, with no compromises. The result of that decision is this book.

Here you will not find a description of how to create a calculated column, or which dialog box to use to set some property. This is not a step-by-step book that teaches you how to use Microsoft Visual Studio, Power BI, or Power Pivot for Excel. Instead, this is a deep dive into the DAX language, starting from the beginning and then reaching very technical details about how to optimize your code and model.

We loved each page of this book while we were writing it. We reviewed the content so many times that we had it memorized. We continued adding content whenever we thought there was something important to include, thus increasing the page count and never cutting something because there were no pages left. Doing that, we learned more about DAX and we enjoyed every moment spent doing it.

But there is one more thing. Why should you read a book about DAX?

Come on, you thought this after the first demo of Power Pivot or Power BI. You are not alone, we thought the same the first time we tried it. DAX is so easy! It looks so similar to Excel! Moreover, if you already learned other programming and/or query languages, you are probably used to learning a new language by looking at some examples of the syntax, matching patterns you find to those you already know. We made this mistake, and we would like you to avoid doing the same.

DAX is a strong language, used in a growing number of analytical tools. It is very powerful, but it has a few concepts that are hard to understand by inductive reasoning. The evaluation context, for instance, is a topic that requires a deductive approach: You start with a theory, and then you see a few examples that demonstrate how the theory works. Deductive reasoning is the approach of this book. We know that a number of people do not like learning in this way, because they prefer a more practical approach, learning how to solve specific problems, and then with experience and practice, they understand the underlying theory with an inductive reasoning. If you are looking for that approach, this book is not for you. We wrote a book about DAX patterns, full of examples and without any explanation of why a formula works, or why a certain way of coding is better. That book is a good source for copying and pasting DAX formulas. This book has a different goal: to enable you to really master DAX. All the examples demonstrate a DAX behavior; they do not solve a specific problem. If you find formulas that you can reuse in your models, this is good for you. However, always remember that this is just a side effect, not the goal of the example. Finally, always read any note to make sure there are no possible pitfalls in the code used in examples. For educational purposes we often used code that is not the best practice.

We really hope you will enjoy spending time with us in this beautiful trip to learn DAX, at least in the same way as we enjoyed writing it.

Who this book is for

If you are a casual user of DAX, then this book is probably not the best choice for you. Many books provide a simple introduction to the tools that implement DAX and to the DAX language itself, starting from the ground and reaching a basic level of DAX programming. We know this very well, because we wrote some of those books, too!

If, on the other hand, you are serious about DAX and you really want to understand every detail of this beautiful language, then this is your book. This might be your first book about DAX; in that case you should not expect to benefit from the most advanced topics too early. We suggest you read the book from cover to cover and then read again the most complex parts once you gained some experience; it is very likely that some concepts will become clearer at that point.

DAX is useful to different people, for different purposes: Excel users can leverage DAX to author Power Pivot data models, business intelligence (BI) professionals might need to implement DAX code in BI solutions of any size, casual Power BI users might need to author some DAX formulas in their self-service BI models. In this book, we tried to provide information to all of these different kinds of people. Some of the content (specifically the optimization part) is probably more targeted to BI professionals, because the knowledge needed to optimize a DAX measure is very technical; but we believe that Excel users should understand the different performance of DAX expressions to achieve the best results for their models, too.

Finally, we wanted to write a book to study, not only a book to read. At the beginning, we try to keep it easy and follow a logical path from zero to DAX. However, when the concepts to learn start to become more complex, we stop trying to be simple, and we are realistic. DAX is not a simple language. It took years for us to master it and to understand every detail of the engine. Do not expect to be able to learn all of this content in a few days, by reading casually. This book requires your attention at a very high level. In exchange for that, of course, we offer an unprecedented depth of coverage of all aspects of DAX, giving you the option to become a real DAX expert.

Assumptions about you

We expect our reader to have a basic knowledge of Excel Pivot Tables and some experience in analysis of numbers. If you already have some exposure to the DAX language, then this is good for you, since you will read the first part faster, but knowing DAX is not necessary, of course.

There are some references in the book to MDX and SQL code, but you do not really need to know these languages, because they are just parallels between different ways of writing expressions. If you do not understand those lines of code, it is just fine, it means that that specific topic is not for you.

In the most advanced parts of the book, we discuss parallelism, memory access, CPU usage, and other exquisitely geeky topics that might not be familiar to everybody. Any developer will feel at home there, whereas Excel power users might be a bit intimidated. Nevertheless, when speaking about optimization that information is required. Thus, the most advanced part of the book is aimed more toward BI developers than to Excel users. However, we think that everybody will benefit from reading it.

Organization of this book

The book is designed to flow from introductory chapters to complex ones, in a logical way. Each chapter is written with the assumption that the previous content is fully understood; there is nearly no repetition of concepts explained earlier. For this reason, we strongly suggest that you read it from cover to cover and avoid jumping to more advanced chapters too early.

Once you have read it for the first time, it becomes useful as a reference: If, for example, you are in doubt about the behavior of *ALLSELECTED*, then you can jump straight on to that section and clarify your mind on that. Nevertheless, reading that section without having digested the previous content might result in some frustration or, worse, in an incomplete understanding of the concepts.

With that said, here is the content at a glance:

- Chapter 1 is a brief introduction to DAX, with a few sections dedicated to users who already have some knowledge of other languages, namely SQL, Excel, or MDX. We do not introduce any new concept here, we just give several hints about the difference between DAX and other languages that might be known to the reader.
- Chapter 2 introduces the DAX language itself. We cover basic concepts such as calculated columns, measures, error-handling functions, and we list most of the basic functions of the language.
- Chapter 3 is dedicated to basic table functions. Many functions in DAX work on tables and return tables as a result. In this chapter we cover the most basic functions, whereas we cover advanced ones in Chapter 9.
- Chapter 4 is dedicated to the description of evaluation contexts. Evaluation contexts are the foundation of the DAX language and this chapter, along with the next one, is probably the most important of the entire book.
- Chapter 5 covers only two functions: *CALCULATE* and *CALCULATETABLE*. These are the most important functions in DAX and they strongly rely on a good understanding of evaluation contexts.
- Chapter 6 contains some examples of DAX code. However, you should not consider it as a set of patterns to reuse. Instead, we show how to solve some common scenarios with the basic concepts learned so far.

- Chapter 7 covers time intelligence calculations at a very in-depth level. Year-to-date, month-to-date, values of the previous year, week-based periods, and custom calendars are some of the calculations covered in this chapter.
- Chapter 8 is dedicated to statistical functions such as ranking, financial calculations, and percentiles.
- Chapter 9 is the continuation of Chapter 3, where we introduce the basic table functions. In this chapter, we go forward explaining in great detail the full set of DAX functions that manipulate tables, most of which are very useful in writing DAX queries.
- Chapter 10 brings your knowledge of evaluation context one step further and discusses complex functions such as *ALLSELECTED* and *KEEPFILTERS*, with the aid of the theory of expanded tables. It is a hard chapter, which uncovers most of the secrets of complex DAX expressions.
- Chapter 11 shows you how to perform calculations over hierarchies and how to handle parent/child structures using DAX.
- Chapter 12 is about solving uncommon relationships in DAX. In fact, with the aid of DAX a data model might express any kind of relationship. In this chapter, we show many types of relationships. It is the last chapter about the DAX language; the remaining part of the book covers optimization techniques.
- Chapter 13 shows a detailed description of the VertiPaq engine; it is the most common database engine on top of which DAX runs. Understanding it is essential to learn how to get the best performance in DAX.
- Chapter 14 uses the knowledge of Chapter 13 to show possible optimizations that you can apply at the data model level. When to normalize, how to reduce cardinality of columns, what kind of relationships to set to achieve top performance and low memory usage in DAX.
- Chapter 15 teaches how to read a query plan and how to measure the performance of a DAX query with the aid of tools such as SQL Server Profiler and DAX Studio.
- Chapter 16 shows several optimization techniques, based on the content of the previous chapters about optimization. We show many DAX expressions, measure their performance, and then show and explain optimized formulas.

Conventions

The following conventions are used in this book:

- **Boldface** type is used to indicate text that you type.
- *Italic* type is used to indicate new terms, measures, calculated columns, and database names.
- The first letters of the names of dialog boxes, dialog box elements, and commands are capitalized. For example, the Save As dialog box.
- The names of ribbon tabs are given in ALL CAPS.
- Keyboard shortcuts are indicated by a plus sign (+) separating the key names. For example, Ctrl+Alt+Delete means that you press Ctrl, Alt, and Delete keys at the same time.

About the companion content

We have included companion content to enrich your learning experience. The companion content for this book can be downloaded from the following page:

http://aka.ms/GuidetoDAX/files

The companion content includes the following:

- A SQL Server backup of the Contoso Retail DW database that you can use to build the examples yourself. This is a standard demo database provided by Microsoft, which we enriched with some views, to make it easier to create a data model on top of it.
- A Power BI Desktop model that we used to generate all of the figures in the book. The database is always the same, and then for each chapter there is a document showing the steps required to reproduce the same example we used in the book. You can use this information in case you want to replicate on your system the same scenario we describe in our examples.

Acknowledgments

We have so many people to thank for this book that we know it is impossible to write a complete list. So thanks so much to all of you who contributed to this book—even if you had no idea that you were doing it. Blog comments, forum posts, email discussions, chats with attendees and speakers at technical conferences, analyzing customer scenarios and so much more have been useful to us, and many people have contributed significant ideas to this book.

That said, there are people we have to mention personally, because of their particular contributions.

We want to start with Edward Melomed: He inspired us, and we probably would not have started our journey with the DAX language without a passionate discussion that we had with him several years ago and that ended with the table of contents of our first book about Power Pivot written on a napkin.

We want to thank Microsoft Press and the people who contributed to the project: Carol Dillingham has been a great editor and greatly helped us along the process of book writing. Many others behind the scenes helped us with the complexity of authoring a book: thanks to you all.

The only job longer than writing a book is the studying you must do in preparation for writing it. A group of people that we (in all friendliness) call "ssas-insiders" helped us get ready to write this book. A few people from Microsoft deserve a special mention as well, because they spent precious time teaching us important concepts about Power Pivot and DAX: They are Marius Dumitru, Jeffrey Wang, Akshai Mirchandani, and Cristian Petculescu. Your help has been priceless, guys!

We also want to thank Amir Netz, Ashvini Sharma, Kasper De Jonge, and T. K. Anand for their contributions to the many discussions we had about the product. We feel they helped us in some strategic choices we made in this book and in our career.

Finally, a special mention goes to our technical reviewers: Gerhard Brückl and Andrea Benedetti. They double-checked all the content of our original text, searching for errors and sentences that were not clear; giving us invaluable suggestions on how to improve the book. Without their meticulous work, the book would have been much harder to read! If the book contains fewer errors than our original manuscript, it is only because of them. If it still contains errors, it is our fault, of course.

Thank you so much, folks!

Errata, updates, and book support

We've made every effort to ensure the accuracy of this book and its companion content. If you discover an error, please submit it to us at:

http://aka.ms/GuidetoDAX/errata

If you need to contact the Microsoft Press Support team, please send an email message to *mspinput@microsoft.com*.

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to *http://support.microsoft.com*.

Free ebooks from Microsoft Press

From technical overviews to in-depth information on special topics, the free ebooks from Microsoft Press cover a wide range of topics. These ebooks are available in PDF, EPUB, and Mobi for Kindle formats, ready for you to download at:

http://aka.ms/mspressfree

Check back often to see what is new!

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://aka.ms/tellpress

We know you are busy, so we have kept it short with just a few questions. Your answers go directly to the editors at Microsoft Press. (No personal information will be requested.) Thanks in advance for your input!

Stay in touch

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

CHAPTER 4

Understanding evaluation contexts

At this point in the book, you have learned the basics of the DAX language. You know how to create calculated columns and measures, and you have a good understanding of common functions used in DAX. This is the chapter where you move to the next level in this language: After learning a solid theoretical background of the DAX language, you will be able to become a real DAX champion.

With the knowledge you have gained so far, you can already create many interesting reports, but you will need to learn evaluation contexts in order to create reports that are more complex. Evaluation contexts are the basis of all of the advanced features of DAX.

We want to give a few words of warning to our readers: The concept of evaluation context is an easy one, and you will learn and understand it soon. Nevertheless, you need to thoroughly understand several subtle considerations and details. Otherwise, you will feel lost at a certain point during your DAX learning path. We teach DAX to many users in public and private classes. We know that this is absolutely normal. At a certain point—you have the feeling that formulas work like magic, because they work, but you do not understand why. Do not worry: you will be in good company. Most DAX students reach that point and many others will reach it in the future. It simply means that evaluation contexts are not clear enough to them. The solution, at that point, is an easy one: Come back to this chapter, read it again, and you will probably find something new, which you missed during your first read.

Moreover, evaluation contexts play an important role with the usage of the function *CALCULATE*, which is probably the most powerful and hard-to-learn function in DAX. We will introduce *CALCULATE* in Chapter 5, "Understanding *CALCULATE* and *CALCULATETABLE*," and then use it through the rest of the book. Understanding *CALCULATE* without having a solid background on evaluation context is problematic. On the other hand, understanding the importance of evaluation contexts without having ever tried to use *CALCULATE* is nearly impossible. Thus, this chapter and the subsequent one are the two that, in our experience with the previous books we have written, are always marked up and have the corners of pages folded over.

Introduction to evaluation contexts

Let's begin by understanding what an evaluation context is. Any DAX expression is evaluated inside a context. The context is the "environment" under which the formula is evaluated. For example, consider a very simple formula for a measure such as:

```
[Sales Amount] := SUMX ( Sales, Sales[Quantity] * Sales[UnitPrice] )
```

You already know what this formula computes: the sum of all the values of quantity multiplied by price in the Sales table. You can put this measure in a pivot table and look at the results, as you can see in Figure 4-1.

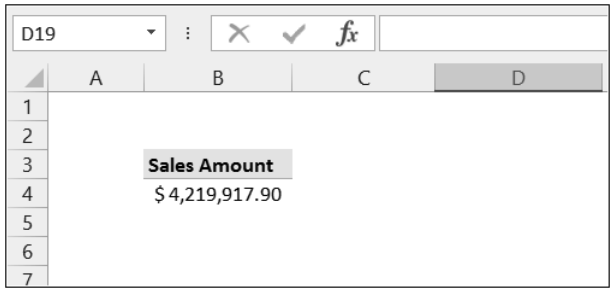


FIGURE 4-1 The measure *Sales Amount*, without a context, shows the grand total of sales.

Well, this number alone does not look interesting at all, does it? But, if you think carefully, the formula computes exactly what it is supposed to compute: the sum of all sales amount, which is a big number with no interesting meaning. This pivot table becomes more interesting as soon as we use some columns to slice the grand total and start investigating it. For example, you can take the product color, put it on the rows, and the pivot table suddenly reveals some interesting business insights, as you can see in Figure 4-2.

The grand total is still there, but now it is the sum of smaller values and each value, together with all the others, has a meaning. However, if you think carefully again, you should note that something weird is happening here: the formula is not computing what we asked.

We supposed that the formula meant "the sum of all sales amount." but inside each cell of the pivot table, the formula is not computing the sum of all sales, it is only computing the sum of sales of products with a specific color. Yet, we never specified that the computation had to work on a subset of the data model. In other words, the formula does not specify that it can work on subsets of data.

Why is the formula computing different values in different cells? The answer is very easy, indeed: because of the evaluation context under which DAX computes the formula. You can think of the evaluation context of a formula as the surrounding area of the cell where DAX evaluates the formula.

Row Labels	Sales Amount
Azure	$12,071.90
Black	$791,735.81
Blue	$294,838.55
Brown	$225,705.83
Gold	$43,292.49
Green	$202,219.08
Grey	$509,990.58
Orange	$55,324.68
Pink	$130,243.02
Purple	$286.00
Red	$126,762.21
Silver	$918,587.35
Silver Grey	$60,366.60
Transparent	$414.54
White	$841,262.48
Yellow	$6,816.78
Grand Total	**$4,219,917.90**

FIGURE 4-2 Sum of *Sales Amount*, sliced by color, looks much more interesting.

Because the product color is on the rows, each row in the pivot table can see, out of the whole database, only the subset of products of that specific color. This is the surrounding area of the formula, that is, a set of filters applied to the database prior to the formula evaluation. When the formula computes the sum of all sales amount, it does not compute it over the entire database, because it does not have the option to look at all the rows. When DAX computes the formula in the row with the value White, only white products are visible and, because of that, it only considers sales pertinent to white products. So the sum of all sales amount, when computed for a row in the pivot table that shows only white products, becomes the sum of all sales amount of white products.

Any DAX formula specifies a calculation, but DAX evaluates this calculation in a context, which defines the final value computed. The formula is always the same, but the value is different because DAX evaluates it against different subsets of data.

The only case where the formula behaves in the way it has been defined is on the grand total. At that level, because no filtering happens, the entire database is visible.

Note In these examples, we are using a pivot table for the sake of simplicity. Clearly, you can define an evaluation context with queries too, and you will learn more about it in future chapters. For now, it is better to keep it simple and think of pivot tables only, so as to have a simplified and visual understanding of the concepts.

Now let's put the year on the columns, to make the pivot table even more interesting. The report is now the one shown in Figure 4-3.

Sales Amount	Column Labels			
Row Labels	CY 2007	CY 2008	CY 2009	Grand Total
Azure	$4,863.10	$4,480.80	$2,728.00	$12,071.90
Black	$282,020.06	$245,539.76	$264,175.99	$791,735.81
Blue	$142,503.18	$76,597.95	$75,737.42	$294,838.55
Brown	$68,737.05	$56,648.58	$100,320.20	$225,705.83
Gold	$9,322.00	$17,475.00	$16,495.49	$43,292.49
Green	$85,164.89	$56,495.65	$60,558.54	$202,219.08
Grey	$200,185.94	$210,735.39	$99,069.25	$509,990.58
Orange	$14,779.34	$20,215.54	$20,329.80	$55,324.68
Pink	$77,438.69	$20,174.08	$32,630.25	$130,243.02
Purple	$224.00		$62.00	$286.00
Red	$57,453.06	$26,079.59	$43,229.56	$126,762.21
Silver	$430,786.04	$216,861.80	$270,939.51	$918,587.35
Silver Grey	$34,776.00	$12,122.60	$13,468.00	$60,366.60
Transparent	$126.42	$79.38	$208.74	$414.54
White	$207,955.56	$274,919.98	$358,386.94	$841,262.48
Yellow	$1,387.28	$4,567.24	$862.26	$6,816.78
Grand Total	**$1,617,722.61**	**$1,242,993.34**	**$1,359,201.95**	**$4,219,917.90**

FIGURE 4-3 Sum of *SalesAmount* is now sliced by color and year.

The rules of the game should be clear at this point: Each cell now has a different value even if the formula is always the same, because both row and column selections of the pivot table define the context. In fact, sales for white products in year 2008 are different from sales for white products in 2007. Moreover, because you can put more than one field in both rows and columns, it is better to say that the set of fields on the rows and the set of fields on the columns define the context. Figure 4-4 makes this more evident.

Sales Amount	Column Labels			
Row Labels	CY 2007	CY 2008	CY 2009	Grand Total
⊟Azure	**$4,863.10**	**$4,480.80**	**$2,728.00**	**$12,071.90**
A. Datum	$4,863.10	$4,480.80	$2,728.00	$12,071.90
⊟Black	**$282,020.06**	**$245,539.76**	**$264,175.99**	**$791,735.81**
A. Datum	$13,691.00	$681.90	$10,455.00	$24,827.90
Adventure Works	$45,584.87	$29,849.86	$29,736.17	$105,170.90
Contoso	$37,330.92	$26,921.86	$55,237.03	$119,489.81
Fabrikam	$22,642.87	$49,585.00	$44,397.04	$116,624.91
Litware	$12,742.44	$9,642.05	$25,524.23	$47,908.72
Northwind Traders			$99.99	$99.99
Proseware	$46,793.84	$22,221.50	$13,912.41	$82,927.75
Southridge Video	$64,377.31	$17,329.31	$16,475.62	$98,182.24
Tailspin Toys	$831.31	$1,752.50	$4,168.63	$6,752.44
The Phone Company	$13,050.00	$21,493.00	$21,043.00	$55,586.00
Wide World Importers	$24,975.50	$66,062.78	$43,126.87	$134,165.15

FIGURE 4-4 The context is defined by the set of fields on rows and on columns.

Each cell has a different value because there are two fields on the rows, color and brand name. The complete set of fields on rows and columns defines the context. For example, the context of the cell highlighted in Figure 4-4 corresponds to color Black, brand Contoso, and Calendar Year 2007.

Note It is not important whether a field is on the rows or on the columns (or on the slicer and/or page filter, or in any other kind of filter you can create with a query). All of these filters contribute to define a single context, which DAX uses to evaluate the formula. Putting a field on rows or columns has just some aesthetic consequences, but nothing changes in the way DAX computes values.

Let's see the full picture now. In Figure 4-5, we added the product category on a slicer, and the month name on a filter, where we selected December.

Month: December

Category: Audio, Cameras and camc..., Cell phones, Computers, Games and Toys, Home Appliances, Music, Movies and ..., TV and Video

Sales Amount / Row Labels	Column Labels: CY 2007	CY 2008	CY 2009	Grand Total
Black	**$2,079.58**	**$4,353.40**	**$5,327.52**	**$11,760.50**
Adventure Works	$479.70	$2,953.50		$3,433.20
Contoso	$899.95			$899.95
Fabrikam			$5,327.52	$5,327.52
Litware	$699.93	$1,399.90		$2,099.83
Blue	**$18,570.30**	**$3,198.00**	**$7,395.00**	**$29,163.30**
Litware		$3,198.00	$7,395.00	$10,593.00
Northwind Traders	$18,570.30			$18,570.30
Brown	**$4,199.85**		**$10,788.00**	**$14,987.85**
Litware	$4,199.85		$10,788.00	$14,987.85
Green			**$16,370.10**	**$16,370.10**
Northwind Traders			$16,370.10	$16,370.10

FIGURE 4-5 In a typical report, the context is defined in many ways, including slicers and filters.

It is clear at this point that the values computed in each cell have a context defined by rows, columns, slicers, and filters. All these filters contribute in the definition of a context that DAX applies to the data model prior to the formula evaluation. Moreover, it is important to learn that not all the cells have the same set of filters, not only in terms of values, but also in terms of fields. For example, the grand total on the columns contains only the filter for category, month, and year, but it does not contain the filter for color and brand. The fields for color and brand are on the rows and they do not filter the grand total. The same applies to the subtotal by color within the pivot table: for those cells there is no filter on the manufacturer, the only valid filter coming from the rows is the color.

We call this context the *Filter Context* and, as its name suggests, it is a context that filters tables. Any formula you ever author will have a different value depending on the filter context that DAX uses to perform its evaluation. This behavior, although very intuitive, needs to be well understood.

Now that you have learned what a filter context is, you know that the following DAX expression should be read as "the sum of all sales amount visible in the current filter context":

```
[Sales Amount] := SUMX ( Sales, Sales[Quantity] * Sales[UnitPrice] )
```

You will learn later how to read, modify, and clear the filter context. As of now, it is enough having a solid understanding of the fact that the filter context is always present for any cell of the pivot table or any value in your report/query. You always need to take into account the filter context in order to understand how DAX evaluates a formula.

Understanding the row context

The filter context is one of the two contexts that exist in DAX. Its companion is the row context and, in this section, you will learn what it is and how it works.

This time, we use a different formula for our considerations:

```
Sales[GrossMargin] = Sales[SalesAmount] - Sales[TotalCost]
```

You are likely to write such an expression in a calculated column, in order to compute the gross margin. As soon as you define this formula in a calculated column, you will get the resulting table, as shown in Figure 4-6.

Unit Price	Unit Cost	SalesAmount	TotalCost	GrossMargin
$32.00	$16.31	32	16.31	15.69
$32.00	$16.31	32	16.31	15.69
$147.00	$67.60	147	67.6	79.4
$308.00	$141.64	308	141.64	166.36
$190.00	$87.37	190	87.37	102.63
$43.00	$21.92	43	21.92	21.08
$43.00	$21.92	43	21.92	21.08
$43.00	$21.92	43	21.92	21.08

FIGURE 4-6 The *GrossMargin* is computed for all the rows of the table.

DAX computed the formula for all the rows of the table and, for each row, it computed a different value, as expected. In order to understand the row context, we need to be somewhat pedantic in our reading of the formula: we asked to subtract two columns, but where did we tell DAX from which row of the table to get the values of the columns? You might say that the row to use is implicit. Because it is a calculated column, DAX computes it row by row and, for each row, it evaluates a different result. This is correct, but, from the point of view of the DAX expression, the information about which row to use is still missing.

In fact, the row used to perform a calculation is not stored inside the formula. It is defined by another kind of context: the *row context*. When you defined the calculated column, DAX started an iteration from the first row of the table; it created a row context containing that row and evaluated the expression. Then it moved on to the second row and evaluated the expression again. This happens for all the rows in the table and, if you have one million rows, you can think that DAX created one million row contexts to evaluate the formula one million times. Clearly, in order to optimize calculations, this is not exactly what happens; otherwise, DAX would be a very slow language. Anyway, from the logical point of view, this is exactly how it works.

Let us try to be more precise. A row context is a context that always contains a single row and DAX automatically defines it during the creation of calculated columns. You can create a row context using other techniques, which are discussed later in this chapter, but the easiest way to explain row context is to look at calculated columns, where the engine always creates it automatically.

There are always two contexts

So far, you have learned what the row context and the filter context are. They are the only kind of contexts in DAX. Thus, they are the only way to modify the result of a formula. Any formula will be evaluated under these two distinct contexts: the row context and the filter context.

We call both contexts "evaluation contexts," because they are contexts that change the way a formula is evaluated, providing different results for the same formula.

This is one point that is very important and very hard to focus on at the beginning: *there are always two contexts and the result of a formula depends on both*. At this point of your DAX learning path, you probably think that this is obvious and very natural. You are probably right. However, later in the book, you will find formulas that will be a challenge to understand if you do not remember about the coexistence of the two contexts, each of which can change the result of the formula.

Testing your evaluation context understanding

Before we move on with more complex discussions about evaluation contexts, we would like to test your understanding of contexts with a couple of examples. Please do not look at the explanation immediately; stop after the question and try to answer it. Then read the explanation to make sense out of it.

Using *SUM* in a calculated column

The first test is a very simple one. What is happening if you define a calculated column, in Sales, with this code?

```
Sales[SumOfSalesAmount] = SUM ( Sales[SalesAmount] )
```

Because it is a calculated column, it will be computed row by row and, for each row, you will obtain a result. What number do you expect to see? Choose one from among these options:

- The value of *SalesAmount* for that row, that is, a different value for each row.
- The total of *SalesAmount* for all the rows, that is, the same value for all the rows.
- An error; you cannot use *SUM* inside a calculated column.

Stop reading, please, while we wait for your educated guess before moving on.

Now, let's elaborate on what is happening when DAX evaluates the formula. You already learned what the formula meaning is: "the sum of all sales amount as seen in the current filter context." As this is in a calculated column, DAX evaluates the formula row by row. Thus, it creates a row context for the first row, and then invokes the formula evaluation and proceeds iterating the entire table. The formula computes the sum of all sales amount values in the current filter context, so the real question is "What is the current filter context?" Answer: It is the full database, because DAX evaluates the formula outside of any pivot table or any other kind of filtering. In fact, DAX computes it as part of the definition of a calculated column, when no filter is active.

Even if there is a row context, *SUM* ignores it. Instead, it uses the filter context and the filter context right now is the full database. Thus, the second option is correct: You will get the grand total of sales amount, the same value for all the rows of Sales, as you can see in Figure 4-7.

Quantity	Unit Price	Unit Cost	SalesAmount	SumOfSalesAmount
1	$9.99	$5.09	9.99	4,219,917.90
1	$9.99	$5.09	9.99	4,219,917.90
1	$9.99	$5.09	9.99	4,219,917.90
1	$9.99	$5.09	9.99	4,219,917.90
1	$9.99	$5.09	9.99	4,219,917.90
1	$9.99	$5.09	9.99	4,219,917.90
1	$9.99	$5.09	9.99	4,219,917.90
1	$9.99	$5.09	9.99	4,219,917.90
1	$9.99	$5.09	9.99	4,219,917.90
1	$9.99	$5.09	9.99	4,219,917.90
1	$9.99	$5.09	9.99	4,219,917.90
1	$9.99	$5.09	9.99	4,219,917.90

FIGURE 4-7 *SUM (Sales[SalesAmount])*, in a calculated column, is computed against the full database.

This example shows that the two contexts exist together. They both work on the result of a formula, but in different ways. Aggregate functions like *SUM*, *MIN*, and *MAX* used in calculated columns use the filter context only and ignore the row context, which DAX uses only to determine column values. If you have chosen the first answer, as many students typically do, it is perfectly normal. The point is that you are not yet thinking that the two contexts are working together to change the formula result in different ways. The first answer is the most common, when using intuitive logic, but it is the wrong one, and now you know why.

Using columns in a measure

The second test we want to do with you is slightly different. Imagine you want to define the formula for gross margin in a measure instead of in a calculated column. You have a column with the sales amount, another column for the product cost, and you might write the following expression:

```
[GrossMargin] := Sales[SalesAmount] - Sales[ProductCost]
```

What result should you expect if you try to author such a measure?

1. The expression works correctly, we will need to test the result in a report.
2. An error, you cannot even author this formula.
3. You can define the formula, but it will give an error when used in a pivot table or in a query.

As before, stop reading, think about the answer, and then read the following explanation.

In the formula, we used *Sales[SalesAmount]*, which is a column name, that is, the value of *SalesAmount* in the Sales table. Is this definition lacking something? You should recall, from previous arguments, that the information missing here is the row from where to get the current value of *SalesAmount*. When you write this code inside a calculated column, DAX knows the row to use when it computes the expression, thanks to the row context. However, what happens for a measure? There is no iteration, there is no current row, that is, there is no row context.

Thus, the second answer is correct. You cannot even write the formula; it is syntactically wrong and you will receive an error when you try to enter it.

Remember that a column does not have a value by itself. Instead, it has a different value for each row of a table. Thus, if you want a single value, you need to specify the row to use. The only way to specify the row to use is the row context. Because inside this measure there is no row context, the formula is incorrect and DAX will refuse it.

The correct way to specify this calculation in a measure is to use aggregate functions, as in:

```
[GrossMargin] := SUM ( Sales[SalesAmount] ) - SUM ( Sales[ProductCost] )
```

Using this formula, you are now asking for an aggregation through *SUM*. Therefore, this latter formula does not depend on a row context; it only requires a filter context and it provides the correct result.

Creating a row context with iterators

You learned that DAX automatically creates a row context when you define a calculated column. In that case, the engine evaluates the DAX expression on a row-by-row basis. Now, it is time to learn how to create a row context inside a DAX expression by using iterators.

You might recall from Chapter 2, "Introducing DAX," that all the X-ending functions are iterators, that is, they iterate over a table and evaluate an expression for each row, finally aggregating the results using different algorithms. For example, look at the following DAX expression:

```
[IncreasedSales] := SUMX ( Sales, Sales[SalesAmount] * 1.1 )
```

SUMX is an iterator, it iterates the Sales table and, for each row of the table, it evaluates the sales amount adding 10 percent to its value, finally returning the sum of all these values. In order to evaluate the expression for each row, *SUMX* creates a row context on the Sales table and uses it during the iteration. DAX evaluates the inner expression (the second parameter of *SUMX*) in a row context containing the currently iterated row.

It is important to note that different parameters of *SUMX* use different contexts during the full evaluation flow. Let's look closer at the same expression:

```
= SUMX (
    Sales,                          ← External contexts
    Sales[SalesAmount] * 1.1        ← External contexts + new Row Context
)
```

The first parameter, Sales, is evaluated using the context coming from the caller (for example, it might be a pivot table cell, another measure, or part of a query), whereas the second parameter (the expression) is evaluated using both the external context plus the newly created row context.

All iterators behave in the same way:

1. Create a new row context for each row of the table received as the first parameter.
2. Evaluate the second parameter inside the newly created row context (plus any other context which existed before the iteration started), for each row of the table.
3. Aggregate the values computed during step 2.

It is important to remember that the original contexts are still valid inside the expression: Iterators only add a new row context; they do not modify existing ones in any way. This rule is usually valid, but there is an important exception: If the previous contexts already contained a row context for the same table, then the newly created row context hides the previously existing row context. We are going to discuss this in more detail in the next section.

Using the *EARLIER* function

The scenario of having many nested row contexts on the same table might seem very rare, but, in reality, it happens quite often. Let's see the concept with an example. Imagine you want to count, for each product, the number of other products with a higher price. This will produce a sort of ranking of the product based on price.

To solve this exercise, we use the *FILTER* function, which you learned in the previous chapter. As you might recall, *FILTER* is an iterator that loops through all the rows of a table and returns a new

table containing only the ones that satisfy the condition defined by the second parameter. For example, if you want to retrieve the table of products with a price higher than US$100, you can use:

```
= FILTER ( Product, Product[UnitPrice] > 100 )
```

Note The careful reader will have noted that *FILTER* needs to be an iterator because the expression Product[UnitPrice]>100 can be evaluated if and only if a valid row context exists for Product; otherwise the effective value of Unit Price would be indeterminate. *FILTER* is an iterator function that creates a row context for each row of the table in the first argument, which makes it possible to evaluate the condition in the second argument.

Now, let's go back to our original example: creating a calculated column that counts the number of products that have a higher price than the current one. If you would name the price of the current product **PriceOfCurrentProduct**, then it is easy to see that this pseudo-DAX formula would do what is needed:

```
Product[UnitPriceRank] =
COUNTROWS (
    FILTER (
        Product,
        Product[UnitPrice] > PriceOfCurrentProduct
    )
)
```

FILTER returns only the products with a price higher than the current one and *COUNTROWS* counts those products. The only remaining issue is a way to express the price of the current product, replacing PriceOfCurrentProduct with a valid DAX syntax. With "current," we mean the value of the column in the current row when DAX computes the column. It is harder than you might expect.

You define this new calculated column inside the Product table. Thus, DAX evaluates the expression inside a row context. However, the expression uses a *FILTER* that creates a new row context on the same table. In fact, *Product[UnitPrice]* used in the fifth row of the previous expression is the value of the unit price for the current row iterated by *FILTER - our inner iteration*. Therefore, this new row context hides the original row context introduced by the calculated column. Do you see the issue? You want to access the current value of the unit price but not use the last introduced row context. Instead, you want to use the previous row context, that is, the one of the calculated column.

DAX provides a function that makes it possible: *EARLIER*. *EARLIER* retrieves the value of a column by using the previous row context instead of the last one. So you can express the value of **PriceOfCurrentProduct** using *EARLIER(Product[UnitPrice])*.

EARLIER is one of the strangest functions in DAX. Many users feel intimidated by *EARLIER*, because they do not think in terms of row contexts and they do not take into account the fact that you can

nest row contexts by creating multiple iterations over the same table. In reality, *EARLIER* is a very simple function that will be useful many times. The following code finally solves the scenario:

```
Product[UnitPriceRank] =
COUNTROWS (
    FILTER (
        Product,
        Product[UnitPrice] > EARLIER ( Product[UnitPrice] )
    )
) + 1
```

In Figure 4-8 you can see the calculated column defined in the Product table, which has been sorted using *Unit Price* in a descending order.

Product Code	Product Name	Unit Price	UnitPriceRank
0802079	Litware Refrigerator L1200 Orange	$3,199.99	1
0802073	Litware Refrigerator 24.7CuFt X980 Grey	$3,199.99	1
0802067	Litware Refrigerator 24.7CuFt X980 Blue	$3,199.99	1
0802061	Litware Refrigerator 24.7CuFt X980 Green	$3,199.99	1
0802055	Litware Refrigerator 24.7CuFt X980 Silver	$3,199.99	1
0802049	Litware Refrigerator 24.7CuFt X980 Brown	$3,199.99	1
0802043	Litware Refrigerator 24.7CuFt X980 White	$3,199.99	1
0802037	Fabrikam Refrigerator 24.7CuFt X9800 Orange	$3,199.99	1
0802031	Fabrikam Refrigerator 24.7CuFt X9800 Grey	$3,199.99	1
0802025	Fabrikam Refrigerator 24.7CuFt X9800 Blue	$3,199.99	1
0802019	Fabrikam Refrigerator 24.7CuFt X9800 Green	$3,199.99	1
0802013	Fabrikam Refrigerator 24.7CuFt X9800 Silver	$3,199.99	1
0802007	Fabrikam Refrigerator 24.7CuFt X9800 Brown	$3,199.99	1
0802001	Fabrikam Refrigerator 24.7CuFt X9800 White	$3,199.99	1
0201033	Adventure Works 52" LCD HDTV X590 Brown	$2,899.99	15
0201032	Adventure Works 52" LCD HDTV X590 White	$2,899.99	15
0201031	Adventure Works 52" LCD HDTV X590 Black	$2,899.99	15
0201030	Adventure Works 52" LCD HDTV X590 Silver	$2,899.99	15
0801036	NT Washer & Dryer 27in L2700 Green	$2,652.90	19
0801031	NT Washer & Dryer 27in L2700 Blue	$2,652.90	19

FIGURE 4-8 *UnitPriceRank* is a useful example of how *EARLIER* is useful to navigate in nested row contexts.

Because there are fourteen products with the same unit price, their rank is always one; the fifteenth product has a rank of 15, shared with other products with the same price. We suggest you study and understand this small example closely, because it is a very good test to check your ability to use and understand row contexts, how to create them using iterators (*FILTER*, in this case), and how to access values outside of them through the usage of *EARLIER*.

Note *EARLIER* accepts a second parameter, which is the number of steps to skip, so that you can skip two or more row contexts. Moreover, there is also a function named *EARLIEST* that lets you access directly the outermost row context defined for a table. To be honest, neither the second parameter of *EARLIER* nor *EARLIEST* is used often: while having two nested row contexts is a common scenario, having three or more of them is something that happens rarely.

Before leaving this example, it is worth noting that, if you want to transform this value into a better ranking (that is, a value that starts with 1 and grows of one, creating a sequence 1, 2, 3…) then counting the prices instead of counting the products is sufficient. Here, the *VALUES* function, which you learned in the previous chapter, comes to help:

```
Product[UnitPriceRankDense] =
COUNTROWS (
    FILTER (
        VALUES ( Product[UnitPrice] ),
        Product[UnitPrice] > EARLIER ( Product[UnitPrice] )
    )
) + 1
```

In Figure 4-9 you can see the new calculated column.

Product Name	Unit Price	UnitPriceRank	UnitPriceRankDense
Litware Refrigerator 24.7CuFt X980 Grey	$3,199.99	1	1
Litware Refrigerator 24.7CuFt X980 Blue	$3,199.99	1	1
Litware Refrigerator 24.7CuFt X980 Green	$3,199.99	1	1
Litware Refrigerator 24.7CuFt X980 Silver	$3,199.99	1	1
Litware Refrigerator 24.7CuFt X980 Brown	$3,199.99	1	1
Litware Refrigerator 24.7CuFt X980 White	$3,199.99	1	1
Fabrikam Refrigerator 24.7CuFt X9800 Orange	$3,199.99	1	1
Fabrikam Refrigerator 24.7CuFt X9800 Grey	$3,199.99	1	1
Fabrikam Refrigerator 24.7CuFt X9800 Blue	$3,199.99	1	1
Fabrikam Refrigerator 24.7CuFt X9800 Green	$3,199.99	1	1
Fabrikam Refrigerator 24.7CuFt X9800 Silver	$3,199.99	1	1
Fabrikam Refrigerator 24.7CuFt X9800 Brown	$3,199.99	1	1
Fabrikam Refrigerator 24.7CuFt X9800 White	$3,199.99	1	1
Adventure Works 52" LCD HDTV X590 Brown	$2,899.99	15	2
Adventure Works 52" LCD HDTV X590 White	$2,899.99	15	2
Adventure Works 52" LCD HDTV X590 Black	$2,899.99	15	2
Adventure Works 52" LCD HDTV X590 Silver	$2,899.99	15	2
NT Washer & Dryer 27in L2700 Green	$2,652.90	19	3
NT Washer & Dryer 27in L2700 Blue	$2,652.90	19	3

FIGURE 4-9 *UnitPriceRankDense* shows a better ranking, because it counts the prices, not the products.

We strongly suggest you learn and understand *EARLIER* thoroughly, because you will use it very often. Nevertheless, it is important to note that variables can be used—in many scenarios—to avoid the use of *EARLIER*. Moreover, a careful use of variables makes the code much easier to read. For example, you can compute the previous calculated column using this expression:

```
Product[UnitPriceRankDense] =
VAR
    CurrentPrice = Product[UnitPrice]
RETURN
    COUNTROWS (
        FILTER (
            VALUES ( Product[UnitPrice] ),
            Product[UnitPrice] > CurrentPrice
        )
    ) + 1
```

In this final example, using a variable, you store the current unit price in the *CurrentPrice* variable, which you use later to perform the comparison. Giving a name to the variable, you make the code easier to read, without having to traverse the stack of row contexts every time you read the expression to make sense of the evaluation flow.

Understanding *FILTER*, *ALL*, and context interactions

In the preceding example, we have used *FILTER* as a convenient way of filtering a table. *FILTER* is a very common function to use, whenever you want to apply a filter that further restricts the existing context.

Imagine that you want to create a measure that counts the number of red products. With the knowledge you gained so far, the formula is an easy one:

```
[NumOfRedProducts] :=
COUNTROWS (
    FILTER (
        Product,
        Product[Color] = "Red"
    )
)
```

This formula works fine and you can use it inside a pivot table; for example, putting the brand on the rows to produce the report shown in Figure 4-10.

Row Labels	NumOfRedProducts
Adventure Works	6
Contoso, Ltd	36
Fabrikam, Inc.	12
Litware, Inc.	12
Northwind Traders	3
Proseware, Inc.	7
Southridge Video	13
Tailspin Toys	6
Wide World Importers	4
Grand Total	**99**

FIGURE 4-10 You can easily count the number of red products using the *FILTER* function.

Before moving on with this example, it is useful to stop for a moment and think carefully how DAX computed these values. The brand is a column of the Product table. The engine evaluates *NumOfRedProducts* inside each cell, in a context defined by the brand on the rows. Thus, each cell shows the number of red products that also have the brand indicated by the corresponding row. This happens because, when you ask to iterate over the Product table, you are really asking to iterate the Product table as it is visible in the current filter context, which contains only products with that specific brand. It might seem trivial, but it is better to remember it multiple times than take a chance of forgetting it.

This is more evident if you put a slicer on the worksheet containing the color. In Figure 4-11 we have created two identical pivot tables with the slicer on color. You can see that the left one has the color Red selected, and the numbers are the same as in Figure 4-10, whereas in the right one the pivot table is empty because the slicer has the color Green selected.

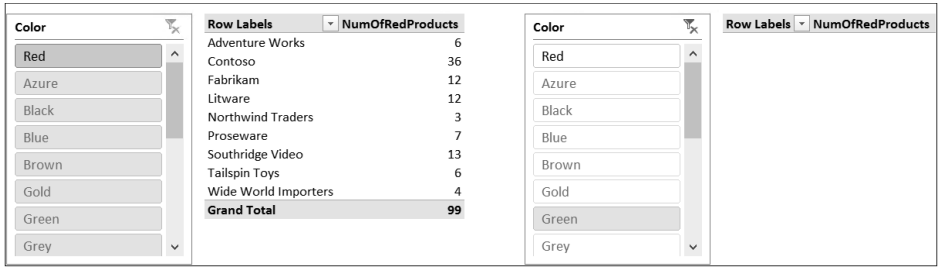


FIGURE 4-11 DAX evaluates *NumOfRedProducts* taking into account the outer context defined by the slicer.

In the right pivot table, the Product table passed into *FILTER* contains only Green products and, because there are no products that can be red and green at the same time, it always evaluates to *BLANK* (that is, *FILTER* does not return any row that *COUNTROWS* can work on).

The important part of this example is the fact that, in the same formula, there are both a filter context coming from the outside (the pivot table cell, which is affected by the slicer selection) and a row context introduced in the formula. Both contexts work at the same time and modify the formula result. DAX uses the filter context to evaluate the Product table, and the row context to filter rows during the iteration.

At this point, you might want to define another formula that returns the number of red products regardless of the selection done on the slicer. Thus, you want to ignore the selection made on the slicer and always return the number of the red products.

You can easily do this by using the *ALL* function. *ALL* returns the content of a table ignoring the filter context, that is, it always returns all the rows of a table. You can define a new measure, named *NumOfAllRedProducts*, by using this expression:

```
[NumOfAllRedProducts] :=
COUNTROWS (
    FILTER (
        ALL ( Product ),
        Product[Color] = "Red"
    )
)
```

This time, instead of referring to Product only, we use *ALL* (Product), meaning that we want to ignore the existing filter context and always iterate over all products. The result is definitely not what we would expect, as you can see in Figure 4-12.

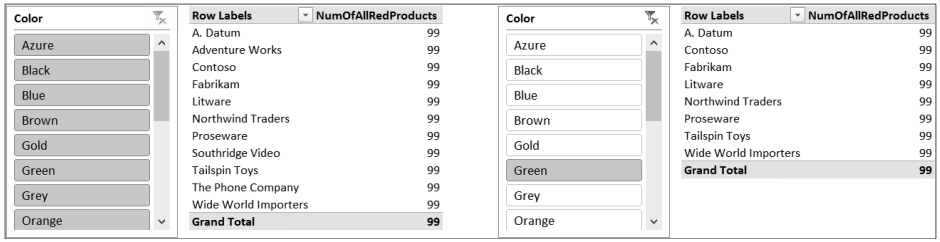


FIGURE 4-12 The *NumOfAllRedProducts* returns strange results.

There are a couple of interesting things to note in here:

- The result is always 99, regardless of the brand selected on the rows.
- The brands in the left pivot table are different from the ones in the right one.

Let us investigate both topics. First, 99 is the total number of red products in the database. Having used *ALL*, we have removed all the filters from the Product table, that is, we have removed both the filter on color and the filter on brand. This is an unwanted effect but, unfortunately, we do not have any other option with the limited knowledge we have of DAX as of now. *ALL* is very powerful, but it is an all-or-nothing function: if used, it removes all of the filters; it has no options to remove only part of them. To be more specific, we wanted to remove only the filter on color, leaving all other filters untouched. In the next chapter, you will learn how to solve this issue with the introduction of *CALCULATE*.

The second point is easier to understand: Because we have selected Green, we are seeing the manufacturers of green products, not the manufacturers of all the products. Thus, the rightmost pivot table shows green product manufacturers with the total of red products in the database. This happens because the list of manufacturers, used to populate the axis of the pivot table, is computed in the original filter context, which contains a filter on color equal to green. Once the axes have been computed, then the values are computed, always returning 99 as a result.

Note This behavior of DAX is called "AutoExists logic." It looks very natural, because it hides nonexisting values, despite some internal complexity. In Chapter 10, "Advanced evaluation context," we will dedicate a section to describe the AutoExists behavior in full detail.

We do not want to solve this scenario right now. The solution will come later when you learn *CALCULATE*, which has specific features to solve scenarios like this one. As of now, we have used this example to show that you might find strange results coming from relatively simple formulas because of context interactions and the coexistence, in the same expression, of filter and row contexts.

Working with many tables

We just started learning contexts, and this led us to some interesting (and surprising) results up to now. You might have noticed that we deliberately used only one table: Product. With only one table, you need to face only interactions between row context and filter context in the same expression.

Very few data models contain just one single table. It is most likely that in your data model you will have many tables linked by relationships. Thus, an interesting question is "*How do the two contexts behave regarding relationships?*" Moreover, because relationships have a direction, we need to understand what happens on the one and on the many side of a relationship. Finally, to make things a bit harder, please recall that relationships can be unidirectional or bidirectional, depending on how you defined the cross-filter direction on relationship itself.

If you create a row context on a table on the many side of the relationship, do you expect it to let you use columns of the one side? Moreover, if you create a row context on the one side of the relationship, do you expect to be able to access columns from the table on the many side? In addition, what about the filter context? Do you expect to put a filter on the many table and see it propagated to the table on the one side? Any answer could be the correct one, but we are interested in learning how DAX behaves in these situations, that is, understand how the DAX language defines propagation of contexts through relationships. As you are going to learn, there are some subtle interactions between contexts and relationships and learning them requires some patience.

In order to examine the scenario, we use a data model containing six tables, which you can see in Figure 4-13.

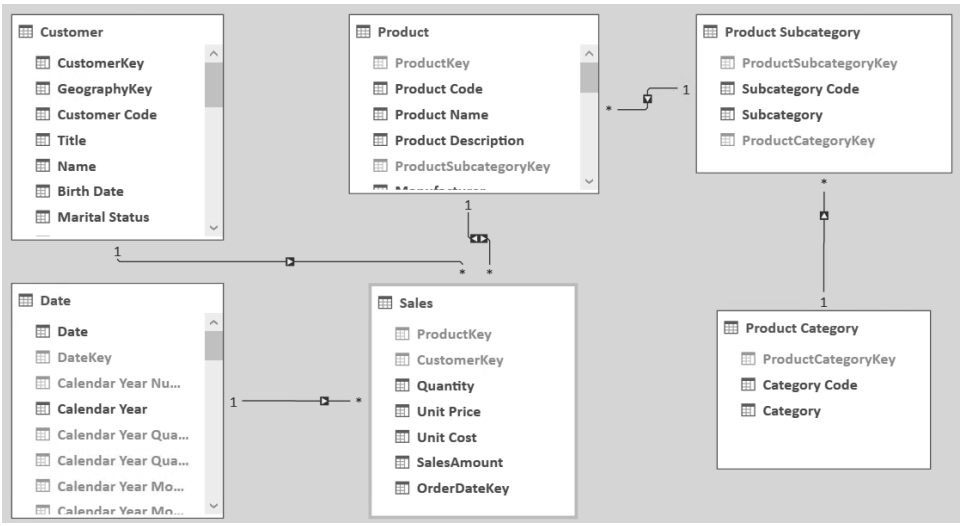


FIGURE 4-13 Here you can see the data model used to learn interaction between contexts and relationships.

There are a couple of things to note about this model:

- There is a chain of one-to-many relationships starting from *Sales* and reaching *Product Category*, through *Product* and *Product Subcategory*.
- The only bidirectional relationship is the one between *Sales* and *Product*. All remaining relationships are set to be one-way cross-filter direction.

Now that we have defined the model, let's start looking at how the contexts behave by looking at some DAX formulas.

Row contexts and relationships

The interaction of row contexts and relationships is very easy to understand, because there is nothing to understand: they do not interact in any way, at least not automatically.

Imagine you want to create a calculated column in the Sales table containing the difference between the unit price stored in the fact table and the product list price stored in the Product table. You could try this formula:

```
Sales[UnitPriceVariance] = Sales[UnitPrice] - Product[UnitPrice]
```

This expression uses two columns from two different tables and DAX evaluates it in a row context that iterates over Sales only, because you defined the calculated column within that table (Sales). Product is on the one side of a relationship with Sales (which is on the many side), so you might expect to be able to gain access to the unit price of the related row (the product sold). Unfortunately, this does not happen. The row context in Sales does not propagate automatically to Product and DAX returns an error if you try to create a calculated column by using the previous formula.

If you want to access columns on the one side of a relationship from the table on the many side of the relationship, as is the case in this example, you must use the *RELATED* function. *RELATED* accepts a column name as the parameter and retrieves the value of the column in a corresponding row that is found by following one or more relationships in the many-to-one direction, starting from the current row context.

You can correct the previous formula with the following code:

```
Sales[UnitPriceVariance] = Sales[UnitPrice] - RELATED ( Product[UnitPrice] )
```

RELATED works when you have a row context on the table on the many side of a relationship. If the row context is active on the one side of a relationship, then you cannot use it because many rows would potentially be detected by following the relationship. In this case, you need to use *RELATEDTABLE*, which is the companion of *RELATED*. You can use *RELATEDTABLE* on the one side of the relationship and it returns all the rows (of the table on the many side) that are related with the

current one. For example, if you want to compute the number of sales of each product, you can use the following formula, defined as a calculated column on Product:

```
Product[NumberOfSales] = COUNTROWS ( RELATEDTABLE ( Sales ) )
```

This expression counts the number of rows in the Sales table that correspond to the current product. You can see the result in Figure 4-14.

Product Code	Product Name	NumberOfSales
0101001	Contoso 512MB MP3 Player E51 Silver	12
0101002	Contoso 512MB MP3 Player E51 Blue	9
0101003	Contoso 1G MP3 Player E100 White	4
0101004	Contoso 2G MP3 Player E200 Silver	
0101005	Contoso 2G MP3 Player E200 Red	8
0101006	Contoso 2G MP3 Player E200 Black	
0101007	Contoso 2G MP3 Player E200 Blue	10
0101008	Contoso 4G MP3 Player E400 Silver	74

FIGURE 4-14 *RELATEDTABLE* is very useful when you have a row context on the one side of the relationship.

It is worth noting that both, *RELATED* and *RELATEDTABLE,* can traverse a long chain of relationships to gather their result; they are not limited to a single hop. For example, you can create a column with the same code as before but, this time, in the Product Category table:

```
'Product Category'[NumberOfSales] = COUNTROWS ( RELATEDTABLE ( Sales ) )
```

The result is the number of sales for the category, which traverses the chain of relationships from Product Category to Product Subcategory, then to Product to finally reach the Sales table.

Note The only exception to the general rule of *RELATED* and *RELATEDTABLE* is for one-to-one relationships. If two tables share a 1:1 relationship, then you can use both *RELATED* and *RELATEDTABLE* in both tables and you will get as a result either a column value or a table with a single row, depending on the function you have used.

The only limitation—with regards to chains of relationships—is that all the relationships need to be of the same type (that is, one-to-many or many-to-one), and all of them going in the same direction. If you have two tables related through one-to-many and then many-to-one, with an intermediate bridge table in the middle, then neither *RELATED* nor *RELATEDTABLE* will work. A 1:1 relationship behaves at the same time as a one-to-many and as a many-to-one. Thus, you can have a 1:1 relationship in a chain of one-to-many without interrupting the chain.

Let's make this concept clearer with an example. You might think that Customer is related with Product because there is a one-to-many relationship between Customer and Sales, and then a many-to-one relationship between Sales and Product. Thus, a chain of relationships links the two tables. Nevertheless, the two relationships are not in the same direction.

We call this scenario a many-to-many relationship. In other words, a customer is related to many products (the ones bought) and a product is related to many customers (the ones who bought the product). You will learn the details of how to make many-to-many relationships work later; let's focus on row context, for the moment. If you try to apply *RELATEDTABLE* through a many-to-many relationship, the result could be not what you might expect. For example, consider a calculated column in Product with this formula:

```
Product[NumOfBuyingCustomers] = COUNTROWS ( RELATEDTABLE ( Customer ) )
```

You might expect to see, for each row, the number of customers who bought that product. Unexpectedly, the result will always be 18869, that is, the total number of customers in the database, as you can see in Figure 4-15.

Product Code	Product Name	NumberOfCustomers
0101001	Contoso 512MB MP3 Player E51 Silver	18869
0101002	Contoso 512MB MP3 Player E51 Blue	18869
0101003	Contoso 1G MP3 Player E100 White	18869
0101004	Contoso 2G MP3 Player E200 Silver	18869
0101005	Contoso 2G MP3 Player E200 Red	18869
0101006	Contoso 2G MP3 Player E200 Black	18869
0101007	Contoso 2G MP3 Player E200 Blue	18869
0101008	Contoso 4G MP3 Player E400 Silver	18869

FIGURE 4-15 *RELATEDTABLE* does not work if you try to traverse a many-to-many relationship.

RELATEDTABLE cannot follow the chain of relationships because they are not in the same direction: one is one-to-many, the other one is many-to-one. Thus, the filter from Product cannot reach Customers. It is worth noting that if you try the formula in the opposite direction, that is, you count, for each of the customers, the number of bought products, the result will be correct: a different number for each row representing the number of products bought by the customer. The reason for this behavior is not the propagation of a filter context but, rather, the context transition created by a hidden *CALCULATE* inside *RELATEDTABLE*. We added this final note for the sake of completeness. It is not yet time to elaborate on this: You will have a better understanding of this after reading Chapter 5, "Understanding *CALCULATE* and *CALCULATETABLE*."

Filter context and relationships

You have learned that row context does not interact with relationships and that, if you want to traverse relationships, you have two different functions to use, depending on which side of the relationship you are on while accessing the target table.

Filter contexts behave in a different way: They interact with relationships in an automatic way and they have different behaviors depending on how you set the filtering of the relationship. The general rule is that the filter context propagates through a relationship if the filtering direction set on the relationship itself makes propagation feasible.

This behavior is very easy to understand by using a simple pivot table with a few measures. In Figure 4-16 you can see a pivot table browsing the data model we have used so far, with three very simple measures defined as follows:

```
[NumOfSales]     := COUNTROWS ( Sales )
[NumOfProducts]  := COUNTROWS ( Product )
[NumOfCustomers] := COUNTROWS ( Customer )
```

Row Labels	NumOfSales	NumOfProducts	NumOfCustomers
Azure	43	14	18,869
Black	3,086	602	18,869
Blue	714	200	18,869
Brown	318	77	18,869
Gold	106	50	18,869
Green	262	74	18,869
Grey	1,113	283	18,869
Orange	126	55	18,869
Pink	435	84	18,869
Purple	8	6	18,869
Red	647	99	18,869
Silver	2,595	417	18,869
Silver Grey	103	14	18,869
Transparent	100	1	18,869
White	2,679	505	18,869
Yellow	200	36	18,869
Grand Total	**12,535**	**2,517**	**18,869**

FIGURE 4-16 Here you can see the behavior of filter context and relationships.

The filter is on the product color. Product is the source of a one-to-many relationship with Sales, so the filter context propagates from Product to Sales, and you can see this because the *NumOfSales* measure counts only the sales of products with the specific color. *NumOfProducts* shows the number of products of each color, and a different value for each row (color) is what you would expect, because the filter is on the same table where we are counting.

On the other hand, *NumOfCustomers*, which counts the number of customers, always shows the same value, that is, the total number of customers. This is because the relationship between Customer and Sales, as you can see Figure 4-17, has an arrow in the direction of Sales.

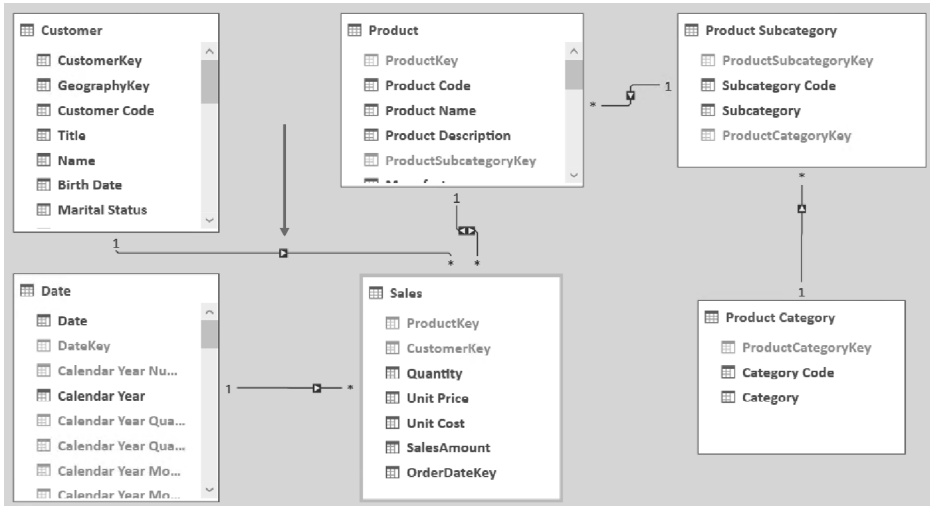


FIGURE 4-17 The relationship between *Customer* and *Sales* is a one-way relationship.

The filter started from *Product*, then propagated to *Sales* (following the arrow from *Product* to *Sales*, which is enabled) but then, when it tried to propagate to *Customer*, it did not find the arrow letting it continue. Thus, it stopped. One-way relationships permit propagation of the filter context in a single direction, not in both.

You can think that the arrows on the relationships are like semaphores. If they are enabled, then the semaphore light is green and the propagation happens. If, on the other hand, the arrow is not enabled, then the semaphore light is red and the filter cannot be propagated.

The arrow is always enabled from the one side to the many side of any relationship. You have the option of enabling it from the many side to the one side as well. If you let the arrow disable, then the propagation will not happen from the many to the one side.

You can better appreciate the behavior if you look at the pivot table shown in Figure 4-18. Instead of using the product color on the rows, this time we slice by customer education.

Row Labels	NumOfSales	NumOfProducts	NumOfCustomers
(blank)	9,826	1,039	385
Bachelors	665	119	5,356
Graduate Degree	418	72	3,189
High School	573	121	3,294
Partial College	746	121	5,064
Partial High School	307	78	1,581
Grand Total	**12,535**	**2,517**	**18,869**

FIGURE 4-18 Filtering by customer education, the Product table is filtered too.

This time the filter starts from *Customer*. It can reach the Sales table, because the arrow is enabled in the corresponding relationship. Then, from *Sales*, it can further propagate to *Product* because the relationship between *Sales* and *Product* is bidirectional.

Now you add to the model a similar measure that counts the number of subcategories, such as the following one:

```
NumOfSubcategories := COUNTROWS ( 'Product Subcategory' )
```

Adding it to the report, you will see that the number of subcategories is not filtered by the customer education, as shown in Figure 4-19.

Calendar	November 2007			
Row Labels	**NumOfSales**	**NumOfProducts**	**NumOfCustomers**	**NumOfSubcategories**
(blank)	177	33	385	44
Bachelors	30	13	5,356	44
Graduate Degree	7	5	3,189	44
High School	28	13	3,294	44
Partial College	42	14	5,064	44
Partial High School	15	7	1,581	44
Grand Total	**299**	**51**	**18,869**	**44**

FIGURE 4-19 If the relationship is unidirectional, customers cannot filter subcategories.

This is because the relationship between Product and Product Subcategory is unidirectional, that is it lets the filter propagate in a single direction. As soon as you enable the arrow starting from Product and going to Product Subcategory, you will see that the filter propagates, as you can see in Figure 4-20.

Calendar	November 2007			
Row Labels	**NumOfSales**	**NumOfProducts**	**NumOfCustomers**	**NumOfSubcategories**
(blank)	177	33	385	16
Bachelors	30	13	5,356	8
Graduate Degree	7	5	3,189	4
High School	28	13	3,294	9
Partial College	42	14	5,064	9
Partial High School	15	7	1,581	7
Grand Total	**299**	**51**	**18,869**	**21**

FIGURE 4-20 If the relationship is bidirectional, customers can filter subcategories too.

As it happened with the row context, it is not important how many steps you need to traverse to reach a table: as long as there is a chain of enabled relationships, automatic propagation happens. For example, if you put a filter on Product Category, the filter propagates to Product Subcategory, then to Product, and finally to Sales.

It is important to note that there are no functions available to access columns or values from tables following the chain of filter contexts, because propagation of the filter context in a DAX expression happens automatically, whereas propagation of row contexts does not and it is required to specify the propagation using *RELATED* and *RELATEDTABLE*.

Introducing *VALUES*

The previous example is very interesting, because it shows how to compute the number of customers who bought a product by using the direction of filtering. Nevertheless, if you are interested only in counting the number of customers, then there is an interesting alternative that we take as an opportunity to introduce as another powerful function: *VALUES*.

VALUES is a table function that returns a table of one column only, containing all the values of a column currently visible in the filter context. There are many advanced uses of *VALUES*, which we will introduce later. As of now, it is helpful to start using *VALUES* just to be acquainted with its behavior.

In the previous pivot table, you can modify the definition of *NumOfCustomers* with the following DAX expression:

```
[NumOfCustomers] := COUNTROWS ( VALUES ( Sales[CustomerKey] ) )
```

This expression does not count the number of customers in the Customer table. Instead, it counts the number of values visible in the current filter context for the *CustomerKey* column in Sales. Thus, the expression does not depend on the relationship between Sales and Customers; it uses only the Sales table.

When you put a filter on Products, it also always filters Sales, because of the propagation of the filter from Product to Sales. Therefore, not all the values of *CustomerKey* will be visible, but only the ones present in rows corresponding to sales of the filtered products.

The meaning of the expression is "count the number of customer keys for the sales related to the selected products." Because a customer key represents a customer, the expression effectively counts the number of customers who bought those products.

Note You can achieve the same result using *DISTINCTCOUNT*, which counts the number of distinct values of a column. As a general rule, it is better to use *DISTINCTCOUNT* than *COUNTROWS* of *VALUES*. We used *COUNTROWS* and *VALUES*, here, for educational purposes, because *VALUES* is a useful function to learn even if its most common usages will be clear in later chapters.

Using *VALUES* instead of capitalizing on the direction of relationships comes with both advantages and disadvantages. Certainly setting the filtering in the model is much more flexible, because it uses relationships. Thus, you can count not only the customers using the *CustomerKey*, but also any other attribute of a customer (number of customer categories, for example). With that said, there might be reasons that force you to use one-way filtering or you might need to use *VALUES* for performance reasons. We will discuss these topics in much more detail in Chapter 12, "Advanced relationship handling."

Introducing *ISFILTERED*, *ISCROSSFILTERED*

Two functions are very useful and might help you gain a better understanding of the propagation of filter contexts. Moreover, learning them is a good way to introduce one of the most interesting concepts of pivot table computation, that is, detection of the cell for which you are computing a value from inside DAX.

These two functions aim to let you detect whether all the values of a column are visible in the current filter context or not; they are:

- *ISFILTERED*: returns *TRUE* or *FALSE*, depending on whether the column passed as an argument has a direct filter on it, that is, it has been put on rows, columns, on a slicer or filter and the filtering is happening for the current cell.
- *ISCROSSFILTERED* returns TRUE or FALSE depending on whether the column has a filter because of automatic propagation of another filter and not because of a direct filter on it.

In this section, we are interested in using the functions to understand the propagation of filter contexts. Thus, we are going to create dummy expressions, which are only useful as learning tools.

If you create a new measure with this definition:

```
[CategoryFilter] := ISFILTERED ( 'Product Category'[Category] )
```

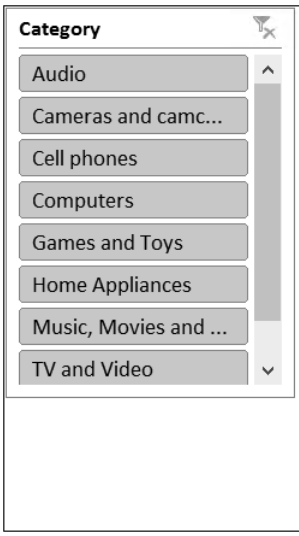
This simple measure returns the value of the *ISFILTERED* function applied to the product category name. You can then create a second measure that makes the same test with the product color. So the code will be:

```
[ColorFilter] := ISFILTERED ( Product[ColorName] )
```

If you add both measures to a pivot table, placing the categories in a slicer and the colors on the rows, the result will be similar to Figure 4-21.

The interesting part is that the category is never filtered because, even if we added a slicer, we did not make a selection on it. The color, on the other hand, is always filtered on rows, because each row has a specific color, but not in the grand total, because the filter context there does not include any selection of products.

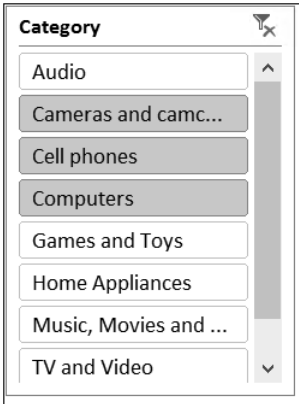
Note This behavior of the grand total, that is, no filter is applied from the ones coming from rows and columns, is very useful whenever you want to modify the behavior of a formula so that, at the grand total level, it shows a different value. In fact, you will check *ISFILTERED* for an attribute present in the pivot table report in order to understand whether the cell you are evaluating is in the inner part of the pivot table or if it is at the grand total level.



Row Labels	CategoryFilter	ColorFilter
Azure	FALSE	TRUE
Black	FALSE	TRUE
Blue	FALSE	TRUE
Brown	FALSE	TRUE
Gold	FALSE	TRUE
Green	FALSE	TRUE
Grey	FALSE	TRUE
Orange	FALSE	TRUE
Pink	FALSE	TRUE
Purple	FALSE	TRUE
Red	FALSE	TRUE
Silver	FALSE	TRUE
Silver Grey	FALSE	TRUE
Transparent	FALSE	TRUE
White	FALSE	TRUE
Yellow	FALSE	TRUE
Grand Total	**FALSE**	**FALSE**

FIGURE 4-21 You can see that Category is never filtered and Color is filtered everywhere but on the grand total.

If you now select some values from the Category slicer, the result changes. Now the category always has a filter, as you can see in Figure 4-22. In fact, the filter context introduced by the slicer is effective even at the grand total level of the pivot table.



Row Labels	CategoryFilter	ColorFilter
Azure	TRUE	TRUE
Black	TRUE	TRUE
Blue	TRUE	TRUE
Brown	TRUE	TRUE
Gold	TRUE	TRUE
Green	TRUE	TRUE
Grey	TRUE	TRUE
Orange	TRUE	TRUE
Pink	TRUE	TRUE
Purple	TRUE	TRUE
Red	TRUE	TRUE
Silver	TRUE	TRUE
Silver Grey	TRUE	TRUE
Transparent	TRUE	TRUE
White	TRUE	TRUE
Yellow	TRUE	TRUE
Grand Total	**TRUE**	**FALSE**

FIGURE 4-22 The filter introduced by the slicer works at the grand total level too.

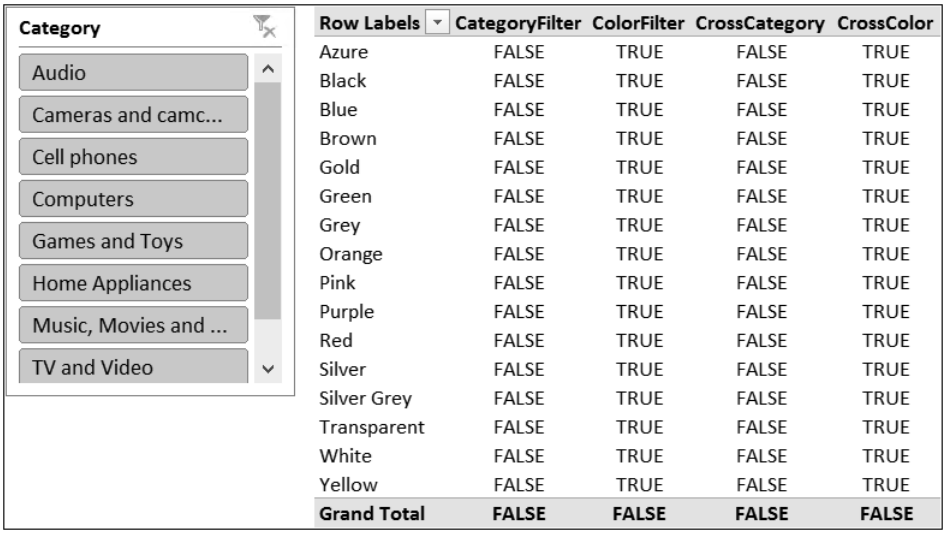
ISFILTERED is useful to detect when a direct filter is working on a column. There are situations where a column does not show all of its values, not because you are filtering the column, but because you placed a filter on another column. For example, if you filter the color and ask for the *VALUES* of

the product brand, you will get as a result only the brands of products of that specific color. When a column is filtered because of a filter on another column, we say that the column is cross-filtered and the *ISCROSSFILTERED* function detects this scenario.

If you add these two new measures to the data model that check, this time, the *ISCROSSFILTERED* of color and category:

```
[CrossCategory] := ISCROSSFILTERED ( 'Product Category'[Category] )
[CrossColor] := ISCROSSFILTERED ( Product[Color] )
```

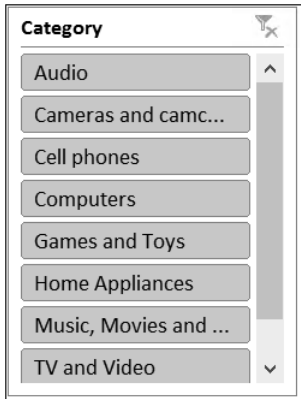
Then you will see the result shown in Figure 4-23.



Row Labels	CategoryFilter	ColorFilter	CrossCategory	CrossColor
Azure	FALSE	TRUE	FALSE	TRUE
Black	FALSE	TRUE	FALSE	TRUE
Blue	FALSE	TRUE	FALSE	TRUE
Brown	FALSE	TRUE	FALSE	TRUE
Gold	FALSE	TRUE	FALSE	TRUE
Green	FALSE	TRUE	FALSE	TRUE
Grey	FALSE	TRUE	FALSE	TRUE
Orange	FALSE	TRUE	FALSE	TRUE
Pink	FALSE	TRUE	FALSE	TRUE
Purple	FALSE	TRUE	FALSE	TRUE
Red	FALSE	TRUE	FALSE	TRUE
Silver	FALSE	TRUE	FALSE	TRUE
Silver Grey	FALSE	TRUE	FALSE	TRUE
Transparent	FALSE	TRUE	FALSE	TRUE
White	FALSE	TRUE	FALSE	TRUE
Yellow	FALSE	TRUE	FALSE	TRUE
Grand Total	**FALSE**	**FALSE**	**FALSE**	**FALSE**

FIGURE 4-23 Cross-filtering is visible using the *ISCROSSFILTERED* function.

You can see that color is cross-filtered and category is not. An interesting question, at this point, is "Why is the category not filtered?" When you filter a color, you might expect to see only the categories of product of that specific color. To answer the question you need to remember that the category is not a column of the Product table. Instead, it is part of Product Category and the arrows on the relationship do not let the relationship propagate. If you change the data model, enabling bidirectional filtering on the full chain of relationships from Product to Product Category, then the result will be different, as is visible in Figure 4-24.



Row Labels	CategoryFilter	ColorFilter	CrossCategory	CrossColor
Azure	FALSE	TRUE	TRUE	TRUE
Black	FALSE	TRUE	TRUE	TRUE
Blue	FALSE	TRUE	TRUE	TRUE
Brown	FALSE	TRUE	TRUE	TRUE
Gold	FALSE	TRUE	TRUE	TRUE
Green	FALSE	TRUE	TRUE	TRUE
Grey	FALSE	TRUE	TRUE	TRUE
Orange	FALSE	TRUE	TRUE	TRUE
Pink	FALSE	TRUE	TRUE	TRUE
Purple	FALSE	TRUE	TRUE	TRUE
Red	FALSE	TRUE	TRUE	TRUE
Silver	FALSE	TRUE	TRUE	TRUE
Silver Grey	FALSE	TRUE	TRUE	TRUE
Transparent	FALSE	TRUE	TRUE	TRUE
White	FALSE	TRUE	TRUE	TRUE
Yellow	FALSE	TRUE	TRUE	TRUE
Grand Total	**FALSE**	**FALSE**	**FALSE**	**FALSE**

FIGURE 4-24 Enabling two-way filtering shows that now the category is cross-filtered, even if not filtered directly.

In this section, you have seen some examples of *ISFILTERED* and *ISCROSSFILTERED*, mainly for educational purposes, because you used them only to get a better understanding of how a filter context propagates through relationships. In following chapters, by writing advanced DAX code, you will learn why these two functions are so useful.

Evaluation contexts recap

Let's recap all what we have learned about evaluation contexts.

- Evaluation context is the context that modifies the value of a DAX expression by filtering the data model and providing the concept of current row, when needed to access a column value.
- The evaluation context consists of two parts: the row context and the filter context. They co-exist and they are present for all the formulas. In order to understand a formula's behavior, you always need to take into account both contexts, because they operate at the same time.
- DAX creates a row context automatically when you define a calculated column. You can also create a row context programmatically by using iterator functions. All iterators define a row context.
- You can nest row contexts and, in such a case, the *EARLIER* function is useful to get access to the previous row context.
- DAX creates a filter context when you use a pivot table by using fields on rows, columns, slicers, and filters. There is a way to programmatically create filter contexts by using *CALCULATE*, but we still have not learned it yet. We hope that at this point you should be very curious to learn more about it!

- Row context does not propagate through relationships automatically. Propagation happens manually by using *RELATED* and *RELATEDTABLE*. You need to use these functions on the correct side of a one-to-many relationship: *RELATED* on the many side, *RELATEDTABLE* on the one side.
- Filter context automatically propagates following the filtering of the relationship. It always propagates from the one side of the relationship to the many side. In addition, you also have the option of enabling the propagation from the many side to the one side. No functions are available to force the propagation: Everything happens inside the engine in an automatic way, according to the definition of relationships in the data model.
- *VALUES* returns a table containing a one-column table with all the unique values of the column that are visible in the current filter context. You can use the resulting table as a parameter to any iterator.

At this point, you have learned the most complex conceptual topics of the DAX language. These points rule all the evaluation flows of your formulas and they are the pillars of the DAX language. Whenever you encounter an expression that does not compute what you want, there's a huge chance that was because you have not fully understood these rules.

As we said in the introduction, at a first reading all these topics look very simple. In fact, they are. What makes them complex is the fact that in a complex expression you might have several evaluation contexts active in different parts of the formula. Mastering evaluation context is a skill that you will gain with experience, and we will try to help you on this by showing many examples in the next chapters. After some DAX formulas on your own, you will intuitively know which contexts are used and which functions they require and you will finally master the DAX language.

Creating a parameter table

In this chapter, you learned many theoretical concepts about evaluation contexts. It is now time to use some of them to solve an interesting scenario and learn a very useful technique, that is, the use of parameter tables.

The idea of a parameter table is to create a table that is unrelated to the rest of the data model, but you will use it internally in DAX expressions to modify their behavior. An example might help to clarify this. Imagine you have created a report that shows the sum of sales amount and, because your company sells many goods, the numbers shown in the report are very large. Because our sample database does not suffer from this problem, instead of using the *SalesAmount* column, we have created a measure that sums the *SalesAmount* cubed, so that numbers are bigger and the described scenario is more realistic. In Figure 4-25 you can see this report.

Sales Amount	Column Labels			
Row Labels	CY 2007	CY 2008	CY 2009	Grand Total
Azure	1,960,392,451.90	705,729,904.55	316,870,144.00	2,982,992,500.45
Black	1,852,957,833,776.85	766,823,353,494.27	460,804,127,759.50	3,080,585,315,030.62
Blue	3,110,738,344,016.30	229,604,213,839.81	320,590,304,143.49	3,660,932,861,999.60
Brown	2,238,683,519,356.14	111,814,853,374.13	2,394,507,848,899.20	4,745,006,221,629.46
Gold	5,328,362,350.00	11,083,737,753.00	18,668,829,287.01	35,080,929,390.01
Green	1,384,243,448,780.78	587,763,173,943.58	527,494,290,700.17	2,499,500,913,424.52
Grey	363,642,216,140.14	1,129,486,253,197.86	121,678,591,398.79	1,614,807,060,736.79
Orange	13,053,660,871.52	9,023,532,983.06	10,442,825,120.49	32,520,018,975.07
Pink	882,892,638,974.98	24,076,918,659.84	44,266,723,593.98	951,236,281,228.80
Purple	702,464.00		61,256.00	763,720.00
Red	277,716,888,210.38	36,410,685,023.30	43,485,805,695.03	357,613,378,928.71
Silver	1,626,446,100,120.32	599,520,302,874.75	1,805,674,285,019.79	4,031,640,688,014.86
Silver Grey	62,115,060,924.00	4,721,772,262.90	13,668,050,062.00	80,504,883,248.90
Transparent	6,124.34	3,125.70	10,647.71	19,897.74
White	849,267,695,977.16	2,923,461,779,978.27	6,665,808,316,354.88	10,438,537,792,310.30
Yellow	4,851,727.20	786,150,633.51	1,631,445.55	792,633,806.26
Grand Total	**12,669,051,722,266.00**	**6,435,282,461,048.52**	**12,427,408,571,527.60**	**31,531,742,754,842.20**

FIGURE 4-25 Reading reports with big numbers is sometimes difficult.

The issue with this report is that the numbers are large and they tend to be hard to read. Are they millions, billions, trillions? Moreover, they use a lot of space in the report, without carrying much information. A common request, for this kind of report, is to show the numbers using a different scale. For example, you might want to show the values divided by a thousand or a million, so that they result in smaller numbers, still carrying the useful information.

You can easily solve this scenario by modifying the measure and dividing it by a thousand. The only problem is that, depending on the relative scale of numbers, you might want to see them as real values (if they are small enough), divided by thousands or divided by millions. Creating three measures seems cumbersome at this point and we want to find a better solution that removes the need of creating many different measures.

The idea is to let the user decide which scale to apply in the report when using a slicer. In Figure 4-26 you can see an example of the report we want to build.

ScaledSalesAmount	Column Labels			
Row Labels	CY 2007	CY 2008	CY 2009	Grand Total
Azure	1,960.39	705.73	316.87	2,982.99
Black	1,852,957.83	766,823.35	460,804.13	3,080,585.32
Blue	3,110,738.34	229,604.21	320,590.30	3,660,932.86
Brown	2,238,683.52	111,814.85	2,394,507.85	4,745,006.22
Gold	5,328.36	11,083.74	18,668.83	35,080.93
Green	1,384,243.45	587,763.17	527,494.29	2,499,500.91
Grey	363,642.22	1,129,486.25	121,678.59	1,614,807.06
Orange	13,053.66	9,023.53	10,442.83	32,520.02
Pink	882,892.64	24,076.92	44,266.72	951,236.28
Purple	0.70		0.06	0.76
Red	277,716.89	36,410.69	43,485.81	357,613.38
Silver	1,626,446.10	599,520.30	1,805,674.29	4,031,640.69
Silver Grey	62,115.06	4,721.77	13,668.05	80,504.88
Transparent	0.01	0.00	0.01	0.02
White	849,267.70	2,923,461.78	6,665,808.32	10,438,537.79
Yellow	4.85	786.15	1.63	792.63
Grand Total	**12,669,051.72**	**6,435,282.46**	**12,427,408.57**	**31,531,742.75**

FIGURE 4-26 The slicer does not filter values here. It is used to change the way numbers are shown.

The interesting idea of the report is that you do not use the *ShowValueAs* slicer to filter data. Instead, you will use it to change the scale used by the numbers. When the user selects Real Value, the actual numbers will be shown. If Thousands is selected, then the actual numbers are divided by one thousand and are shown in the same measure without having to change the layout of the pivot table. The same applies to Millions and Billions.

To create this report, the first thing that you need is a table containing the values you want to show on the slicer. In our example, made with Excel, we use an Excel table to store the scales. In a more professional solution, it would be better to store the table in an SQL database. In Figure 4-27 you can see the content of such a table.

ShowValueAs	DivideBy
Real Value	1
Thousands	1,000
Millions	1,000,000
Billions	1,000,000,000

FIGURE 4-27 This Excel table will be the source for the slicer in the report.

Obviously, you cannot create any relationship with this table, because *Sales* does not contain any column that you can use to relate to this table. Nevertheless, once the table is in the data model, you can use the *ShowValueAs* column as the source for a slicer. Yes, you end up with a slicer that does nothing, but some DAX code will perform the magic of reading user selections and further modifying the content of the repor.

The DAX expression that you need to use for the measure is the following:

```
[ScaledSalesAmount] :=
IF (
    HASONEVALUE ( Scale[DivideBy] ),
    DIVIDE ( [Sales Amount], VALUES ( Scale[DivideBy] ) ),
    [Sales Amount]
)
```

There are two interesting things to note in this formula:

- The condition tested by the *IF* function is: *HASONEVALUE (Scale[ShowValueAs])*. This pattern is very common: you check whether the column of the Scale table has only one value visible. If the user did not select anything in the slicer, then all of the values of the column are visible in the current filter context; that is, *HASONEVALUE* will return *FALSE* (because the column has many different values). If, on the other hand, the user selected a single value, then only that one is visible and *HASONEVALUE* will return *TRUE*. Thus, the condition reads as: "if the user has selected a single value for *ShowValueAs* attribute."

- If a single value is selected, then you know that a single row is visible. Thus, you can compute *VALUES (Scale[DivideBy])* and you are sure that the resulting table contains only one column and one row (the one visible in the filter context). DAX will convert the one-row-one-column table returned by *VALUES* in a scalar value. If you try to use *VALUES* to read a single value when the result is a table with multiple rows, you will get an error. However, in this specific scenario, you are sure that the value returned will be only one, because of the previous condition tested by the *IF* function.

Therefore, you can read the expression as: "If the user has selected a single value in the slicer, then show the sales amount divided by the corresponding denominator, otherwise show the original sales amount." The result is a report that changes the values shown interactively, using the slicer as if it was a button. Clearly, because the report uses only standard DAX formulas, it will work when deployed to SharePoint or Power BI, too.

Parameter tables are very useful in building reports. We have shown a very simple (yet very common) example, but the only limit is your imagination. You can create parameter tables to modify the way a number is computed, to change parameters for a specific algorithm, or to perform other complex operations that change the value returned by your DAX code.

CHAPTER 13

The VertiPaq engine

At this point in the book, you have a solid understanding of the DAX language. The next step, apart from the necessary experience that you need to gain by yourself, is not only being able to write DAX, but also to write efficient DAX. Writing efficient DAX, that is, code that runs at its best possible speed, requires you to understand the internals of the engine. The next chapters aim to provide the essential knowledge to measure and improve performance of DAX code.

More specifically, this chapter is dedicated to the internal architecture of the VertiPaq engine: the in-memory columnar database that stores and hosts your model.

Before continuing with the dissertation, it is worth mentioning a quick note. The official name of the engine on top of which DAX runs is "xVelocity in-memory Analytical Engine." The name appeared later, when the engine was ready to market. During its development, it was code-named "VertiPaq." Because of the late change in the name, many white papers referred to the engine as the VertiPaq engine, and all the early adopters learned its name as VertiPaq. Moreover, internally, the engine is still known as VertiPaq (in fact, as you learn later, its query engine executes VertiPaq queries, not xVelocity queries). In order to avoid confusion in sentences such as "the xVelocity engine executes a VertiPaq query," which mixes both names in a single sentence, we decided to use VertiPaq only.

There is another important note to our readers. Starting from this chapter, we somewhat deviate from DAX and begin to discuss some low-level technical details about the implementation of DAX and the VertiPaq engine. Although this is an important topic, you need to be aware of two facts:

- Implementation details change often. We did our best to show information at a level which is not likely to change soon, carefully balancing detail level and usefulness with consistency over time. The most up-to-date information will always be available in blog posts and articles on the web.
- All the considerations about the engine and optimization techniques are useful if you rely on the VertiPaq engine. In case you are using DirectQuery, then the content of the last chapters of this book is nearly useless in your specific scenario. However, we suggest that you read and understand it anyway, because it shows many details that will help you in choosing the best engine for your analytical scenario.

Understanding database processing

DAX runs in SQL Server Analysis Services (SSAS) Tabular, Power BI service (both on server and on the local Power BI Desktop), and in the Power Pivot for Microsoft Excel add-in. Technically, Power Pivot for Excel runs a local instance of SSAS Tabular, whereas Power BI uses a separate process running a special instance of Analysis Services. Thus, speaking about different engines is somewhat artificial: Power Pivot is SSAS, even if it runs in a "hidden mode" inside Excel. In this book, we make no differences between these engines and, when we speak about SSAS, you might always mentally replace SSAS with Power Pivot or Power BI. If there are differences worth highlighting, then we will note them in the specific section.

When SSAS loads the content of a source table in memory, we say that it processes the table. This happens during the process operation of SSAS or the data refresh in Power Pivot for Excel and Power BI. During processing, the engine reads the content of your data source and transforms it in the internal VertiPaq data structure.

The steps that happen during processing are as follows:

1. Reading of the source dataset, transformation into a columnar data structure of VertiPaq, encoding and compressing each column.
2. Creation of dictionaries and indexes for each column.
3. Creation of the data structures for relationships.
4. Computation and compression of all the calculated columns.

The last two steps are not necessarily sequential. In fact, you can create a relationship based on a calculated column, or have calculated columns that depend on a relationship because they use *RELATED* or *CALCULATE*; SSAS creates a complex graph of dependencies to execute the steps in the correct order.

In the next sections, you learn many more details about these steps and the format of internal structures created by SSAS during the transformation of the data source into the VertiPaq model.

Introduction to columnar databases

VertiPaq is an in-memory columnar database. Being in-memory means that all of the data handled by a model reside in RAM, and it is an easy concept to learn. We briefly introduced the concept of a columnar database in Chapter 5, "Understanding *CALCULATE* and *CALCULATETABLE*." Now it is time to perform a deeper analysis of it.

We think of a table as a list of rows, where each row is divided into columns. Let's take, as an example, the Product table in Figure 13-1.

Product

ID	Name	Color	Unit Price
1	Camcorder	Red	112.25
2	Camera	Red	97.50
3	Smartphone	White	100.00
4	Console	Black	112.25
5	TV	Blue	1,240.85
6	CD	Red	39.99
7	Touch screen	Blue	45.12
8	PDA	Black	120.25
9	Keyboard	Black	120.50

FIGURE 13-1 The figure shows the Product table, with four columns and nine rows.

If you think of a table as a set of rows, then you are using the most natural visualization of a table structure, also known as a *row store*. In a row store, data is organized in rows and, when stored in memory, you might think that the value of the column *Name* in the first row is adjacent to the columns *ID* and *Color* in the same row. On the other hand, the column *Name* in the second row is slightly farther from *Name* in the first row because, in the middle, there are *Color*, *Unit Price* in the first row, and *ID* in the second row.

For example, if you need to compute the sum of *Unit Price*, you have to scan the entire table, reading many values that you are not interested in seeing. You can imagine that scanning the memory of the database sequentially: in order to read the first value of *Unit Price*, you have to read (and skip) *ID*, *Name,* and *Color* of the first row. Only then you will find an interesting value. The same process repeats for all the rows: You need to read and ignore many columns to find the interesting values to sum.

Reading and ignoring values takes time. In fact, if we asked you to compute the sum of *Unit Price*, you would not follow that algorithm. Instead, as a human being, you would probably scan the first row searching for the position of *Unit Price*, and then move your eyes vertically, reading only the values one at a time and mentally accumulating their values to produce the sum. The reason for this very natural behavior is that you save time by reading vertically instead of on a row-by-row basis.

In a columnar database, data is organized in such a way to optimize vertical scanning. To obtain this result, you need a way to make the different values of a column adjacent one to the other. In Figure 13-2 you can see the same Product table as organized by a columnar database.

Product Columns

ID	Name	Color	Unit Price
1	Camcorder	Red	112.25
2	Camera	Red	97.50
3	Smartphone	White	100.00
4	Console	Black	112.25
5	TV	Blue	1,240.85
6	CD	Red	39.99
7	Touch screen	Blue	45.12
8	PDA	Black	120.25
9	Keyboard	Black	120.50

FIGURE 13-2 The Product table organized on a column-by-column basis.

When stored in a columnar database, each column has its own data structure, and it is physically separated from the others. Thus, the different values of *Unit Price* are adjacent one to the other and distant from *Color*, *Name*, and *ID*.

On this data structure, computing the sum of *Unit Price* is much easier, because you immediately go to the column containing the *Unit Price* and then find, very near, all the values you need to perform the computation. In other words, you do not have to read and ignore other column values: In a single scan, you obtain only useful numbers and you can quickly aggregate them.

Now, imagine that, instead of asking you the sum of *Unit Price*, we asked you to compute the sum of *Unit Price* for only the Red products. Try that before you continue reading; it will help in understanding the algorithm.

This is not so easy anymore, because you cannot obtain such a number by simply scanning the *Unit Price* column. What you probably did is a scan of the *Color* column and, whenever it was Red, you grabbed the corresponding value of *Unit Price*. At the end, you summed up all the values to compute the result. This algorithm, although very natural, required you to constantly move your eyes from one column to another, possibly guiding your finger to keep the last scanned position of *Color*. It is definitely not an optimized way of computing the value! A better way, that only computers use, is to first scan the *Color* column, find the row numbers where the color is Red and then, once you know the row numbers, scan the *Unit Price* column summing only the rows you identified in the previous step.

This last algorithm is much better, because it lets you perform one scan of the first column and one scan of the second column, always accessing memory locations that are adjacent one to the other (apart from the jump between the scan of the first and second column).

For a more complex expression, such as the sum of all products that are either Blue or Black Console with a price higher than USD 50, things are even worse. This time, you have no chance of

scanning the column one at a time, because the condition depends on many columns. As usual, if you try it on paper, it helps in understanding it better.

The simplest algorithm to produce such a result is to scan the table not on a column basis but, instead, on a row basis. You probably scanned the table row by row, even if the storage organization is column by column. Although it is a very simple operation when executed on paper by a human, the same operation is extremely expensive if executed by a computer in RAM, because it requires a lot of random reads of memory, leading to a worse performance than if computed doing a sequential scan.

Columnar databases provide very quick access to a single column but, as soon as you need a calculation that involves many columns, they need to spend some time—after having read the column content—to reorganize the information in such a way that the final expression can be computed. Even if this example was a very simple one, it is already very useful to highlight the most important characteristics of column stores:

- Single-column access is very fast, because it reads a single block of memory, and then it computes whatever aggregation you need on that memory block.
- If an expression uses many columns, the algorithm is more complex because it requires the engine to access different memory areas at different times, keeping track of the progress in some temporary area.
- The more columns you need to compute an expression, the harder it becomes to produce a final value, up to a point where it is easier to rebuild the row storage out of the column store to compute the expression.

Column stores aim to reduce the read time. However, they spend more CPU cycles to rearrange the data when many columns from the same table are used. Row stores, on the other hand, have a more linear algorithm to scan data, but they result in many useless reads. As a general rule, reducing reads at the cost of increasing CPU usage is a good deal, because with modern computers it is always easier (and cheaper) to increase the CPU speed versus reducing I/O (or memory access) time.

Moreover, as you learn in the next sections, columnar databases have more options to reduce the amount of time spent scanning data, that is, compression.

Understanding VertiPaq compression

In the previous section, you learned that VertiPaq stores each column in a separate data structure. This simple fact allows the engine to implement some extremely important compressions and encoding that you are about to learn in this section.

Note Please note that the actual details of the compression algorithm of VertiPaq are proprietary and, of course, we cannot publish them in a book. Yet what we explain in this chapter is already a good approximation of what really happens in the engine and you can use it, to all effects, to understand how the VertiPaq engine stores data.

VertiPaq compression algorithms aim to reduce the memory footprint of your data model. Reducing the memory usage is a very important task for two very good reasons:

- A smaller model makes a better use of your hardware. Why spend money on 1 TB of RAM when the same model, once compressed, can be hosted in 256 GB? Saving RAM is always a good option, if feasible.
- A smaller model is faster to scan. As simple as this rule is, it is very important when speaking about performance. If a column is compressed, the engine will scan less RAM to read its content, resulting in better performance.

Understanding value encoding

Value encoding is the first kind of encoding that VertiPaq might use to reduce the memory of a column. Imagine you have a column containing the price of products, stored as integer values. The column contains many different values and, to represent all of them, you need a defined number of bits.

In the Figure 13-3 example, the maximum value of *Unit Price* is 216. Therefore, you need at least 8 bits to store each value. Nevertheless, by using a simple mathematical operation, you can reduce the storage to 5 bits.

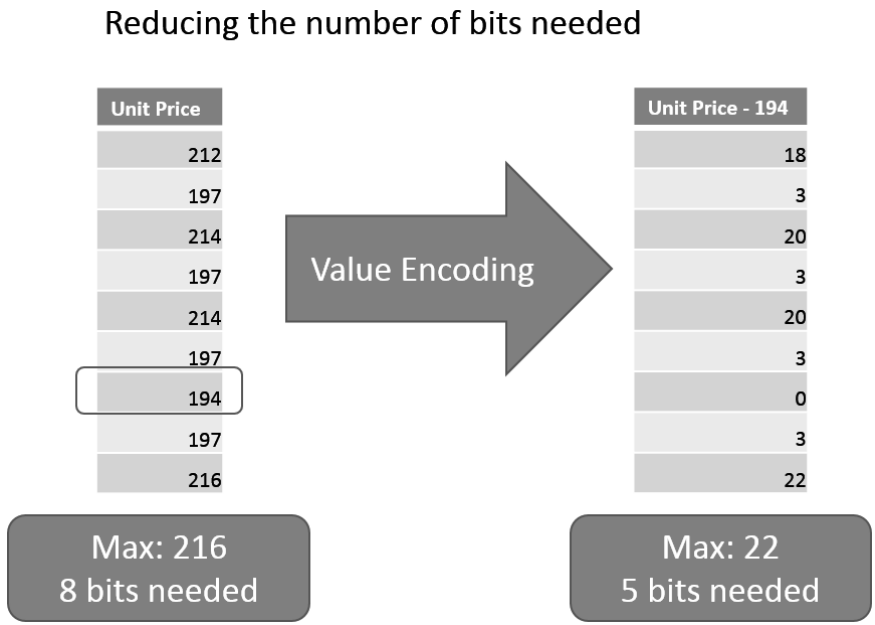


FIGURE 13-3 By using simple mathematical operations, VertiPaq can reduce the number of bits needed for a column.

In the example, VertiPaq discovered that by subtracting the minimum value (194) from all the values of the column, it could modify the range of the column, reducing it to a range from 0 to 22. Storing numbers up to 22 requires less bits than storing numbers up to 216. While 3 bits might seem a very small saving, when you multiply this for a few billion rows, it is easy to see that the difference can be an important one.

The VertiPaq engine is much more sophisticated than this. It can discover mathematical relationships between the values of a column and, when it finds them, it can use them to modify the storage, reducing its memory footprint. Obviously, when using the column, it has to re-apply the transformation in the opposite direction to again obtain the original value (depending on the transformation, this can happen before or after aggregating the values). Again, this will increase the CPU usage and reduce the amount of reads, which, as we already discussed, is a very good option.

Value encoding happens only for integer columns because, obviously, it cannot be applied on strings or floating-point values. Please consider that VertiPaq stores the currency data type of DAX in an integer value.

Understanding dictionary encoding

Dictionary encoding is another technique used by VertiPaq to reduce the number of bits required to store a column. Dictionary encoding builds a dictionary of the distinct values of a column and then it replaces the column values with indexes to the dictionary. Let's see this with an example. In Figure 13-4 you can see the *Color* column, which uses strings and, thus, cannot be value-encoded.

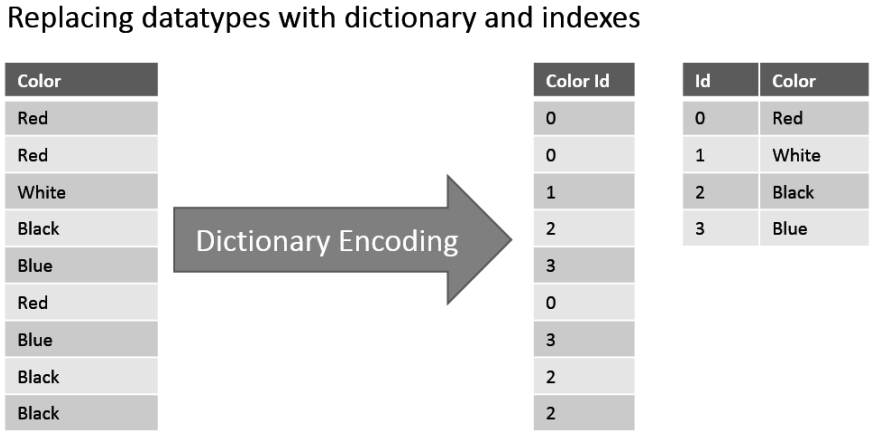


FIGURE 13-4 Dictionary encoding consists of building a dictionary and replacing values with indexes.

When VertiPaq encodes a column with dictionary encoding, it

- Builds a dictionary, containing the distinct values of the column.
- Replaces the column values with integer numbers, where each number is the dictionary index of the original value.

There are some advantages in using dictionary encoding:

- All columns contain only integer values; this makes it simpler to optimize the internal code of the engine. Moreover, it basically means that VertiPaq is datatype independent.
- The number of bits used to store a single value is the minimum number of bits necessary to store an index entry. In the example provided, having only four different values, 2 bits are sufficient.

These two aspects are of paramount importance for VertiPaq. It does not matter whether you use a string, a 64-bit integer, or a floating point to represent a value. All these datatypes will be dictionary encoded, providing the same performance, both in terms of speed of scanning and of storage space. The only difference might be in the size of the dictionary, which is typically very small when compared with the size of the column itself.

The primary factor to determine the column size is not the datatype, but it is the number of distinct values of the column. We refer to the number of distinct values of a column as its *cardinality*. Repeating a concept so important is always a good thing: Of all the various factors of an individual column, the most important one when designing a data model is its cardinality.

The lower the cardinality, the smaller the number of bits required to store a single value and, as a consequence, the smaller the memory footprint of the column. If a column is smaller, not only will it be possible to store more data in the same amount of RAM, but it will also be much faster to scan it whenever you need to aggregate its values in a DAX expression.

Understanding Run Length Encoding (RLE)

Dictionary encoding and value encoding are two very good alternative compression techniques. However, there is another complementary compression technique used by VertiPaq: Run Length Encoding (RLE). This technique aims to reduce the size of a dataset by avoiding repeated values. For example, consider a column containing the calendar quarter of a sale, stored in the Sales table. This column might have the string "Q1" repeated many times in contiguous rows, for all the sales in the same quarter. In such a case, VertiPaq avoids storing repeating values and replaces them with a slightly more complex structure that contains the value only once, with the number of contiguous rows having the same value, as you can see in Figure 13-5.

In Figure 13-5, you see *Quarter*, *Start,* and *Count*. In reality, *Start* is not required because VertiPaq can compute it by summing all the previous values of *Count*, again saving precious bytes of RAM.

RLE's efficiency strongly depends on the repetition pattern of the column. Some columns will have the same value repeated for many rows, resulting in a great compression ratio. Some others, with quickly changing values, will produce a lower compression ratio. Sorting of data is extremely important in order to improve the compression ratio of RLE, as you will see later in this chapter.

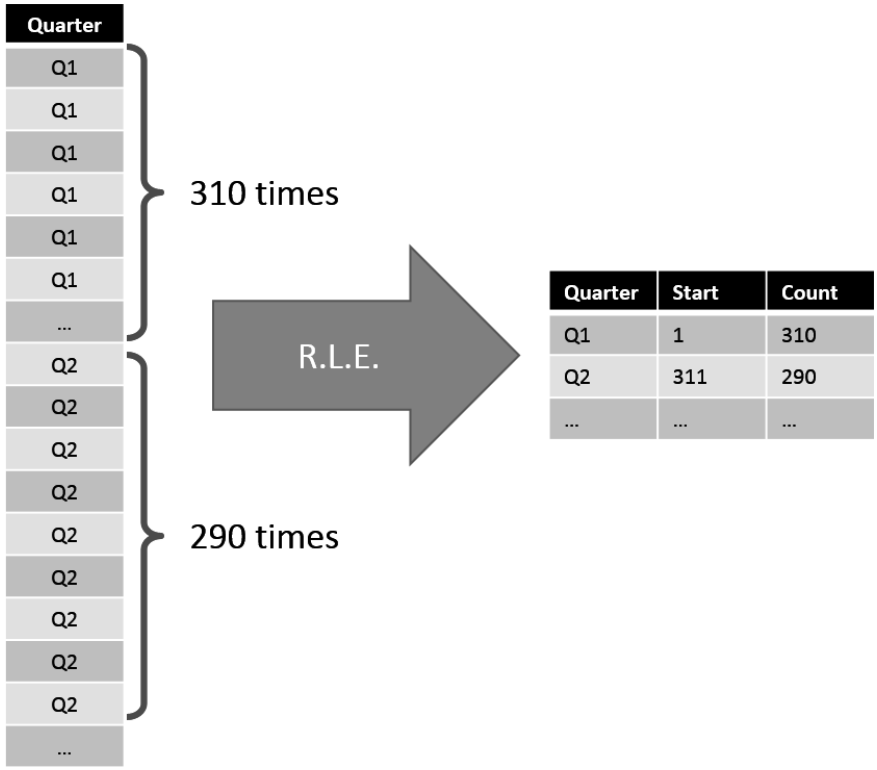


FIGURE 13-5 RLE replaces repeating values with the number of contiguous rows with the same value.

Finally, you might have a column in which content changes so often that, if you try to compress it using RLE, you end up using more space than its original one. Think, for example, of the primary key of a table. It has a different value for each row, resulting in an RLE version larger than the column itself. In such a case, VertiPaq skips the RLE compression and stores the column as it is. Thus, the VertiPaq storage of a column will never exceed the original column size. At worst, it is the same.

In the example, we have shown RLE working on the *Quarter* column containing strings. In reality, RLE processes the already dictionary-encoded version of the column. In fact, each column can have both RLE and dictionary or value encoding. Therefore, the VertiPaq storage for a column compressed with dictionary encoding consists of two distinct entities: the dictionary and the data rows. The latter is the RLE-encoded result of the dictionary-encoded version of the original column, as you can see in Figure 13-6.

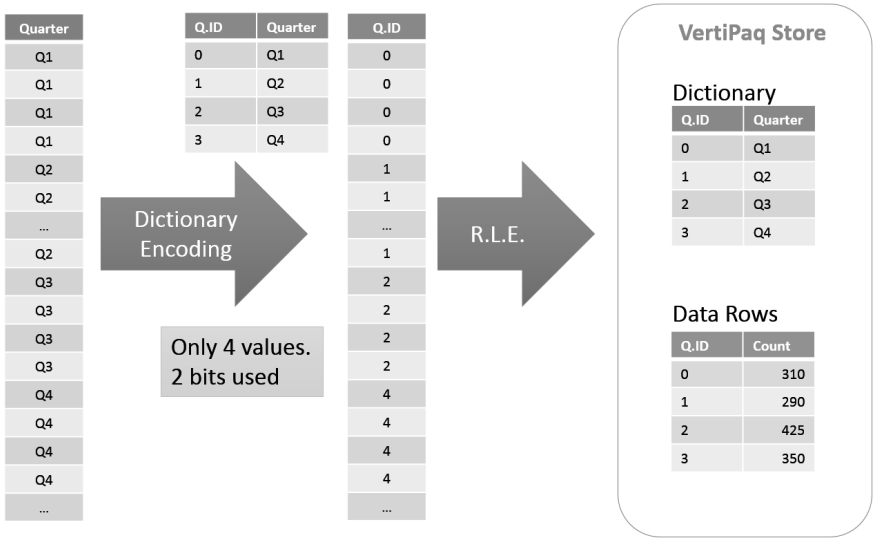


FIGURE 13-6 RLE is applied to the dictionary-encoded version of a column.

VertiPaq applies RLE also to value-encoded columns. In this case, obviously, the dictionary is missing because the column already contains value-encoded integers.

The factors to consider working with a Tabular model, regarding its compression ratio, are, in order of importance:

- The cardinality of the column, which defines the number of bits used to store a value.
- The number of repetitions, that is, the distribution of data in a column. A column with many repeated values will be compressed more than a column with very frequently changing ones.
- The number of rows in the table.
- The datatype of the column (it affects only the dictionary size).

Given all these considerations, you can see that it is nearly impossible to predict the compression ratio of a table. Moreover, while you have full control over some aspects of a table (you can limit the number of rows and change the datatypes), they are the least important ones. Yet as you will learn in the next chapter, you can work on cardinality and repetitions too, to improve the performance of a model.

Finally, it is worth noting that if you reduce the cardinality of a column, you are also increasing the chances of repetitions. For example, if you store a time column at the second granularity, then you have up to 86,400 distinct values in the column. If, on the other hand, you store the same time column at the hour granularity, then you have not only reduced the cardinality, but you have also introduced repeating values (3.600 seconds converts to the same hour), resulting in a much better compression ratio. However, changing the datatype from DateTime to Integer or also String has an irrelevant impact on the column size.

Understanding re-encoding

SSAS has to decide which algorithm to use in order to encode each column. Specifically, it needs to decide whether to use value or dictionary encoding. In reality, the patented algorithms used by SSAS are much more complex and this description is a simplification of them, yet it is enough to get a solid understanding. Anyway, how does SSAS choose the best way to compress a column? It reads a sample of rows during the first scan of the source and, depending on the values found, it chooses an algorithm.

- If the datatype of the column is not integer, then the choice is straightforward: it goes for dictionary encoding.
- For integer values, it uses some heuristics, for example:
  - If the numbers in the column increase linearly, it is probably a primary key, and value encoding is the best option.
  - If all numbers are in a defined range of values, then value encoding is the way to go.
  - If the numbers are in a very wide range of values, with values very different from another, then dictionary encoding is the best choice.

Once the decision is made, SSAS starts to compress the column using the chosen algorithm. Unfortunately, it might have made the wrong decision and discover it only very late during processing. For example, SSAS might read a few million rows where the values are in the range 100–201, so that value encoding is the best choice. After those millions of rows, suddenly an outlier appears, such as a large number like 60,000,000. Obviously, the choice was wrong because the number of bits needed to store such a large number is huge. What to do then? Instead of continuing with the wrong choice, SSAS can decide to re-encode the column. This means that the whole column is re-encoded using, in this case, dictionary encoding. This process might last for a long time, because it needs to reprocess the whole column.

For very large datasets, where processing time is important, a best practice is to provide to SSAS a good sample of data distribution in the first set of rows it reads, to reduce re-encoding to a minimum. You do so by providing a good sample in the first partition processed.

Finding the best sort order

As we already said in the previous pages, RLE's efficiency strongly depends on the sort order of the table. Obviously, all the columns in the same table are sorted in the same way because, at some point during the querying, VertiPaq might have to match different columns for the same row. So in large tables it could be important to determine the best sorting of your data to improve efficiency of RLE and reduce the memory footprint of your model.

When SSAS reads your table, it tries different sort orders to improve the compression. In a table with many columns, this is a very expensive operation. SSAS then sets an upper limit to the time it can spend finding the best sort order. The default can change with different versions of the engine, currently it is 10 seconds per million rows. You can modify its value in the *ProcessingTimeboxSecPerMRow* entry in the configuration file of the SSAS service. If using Power Pivot, you cannot change this value.

Note SSAS searches for the best sort order in data using a heuristic algorithm that certainly also considers the physical order of the rows it receives. For this reason, even if you cannot force the sort order used by VertiPaq for RLE, you can provide to the engine data sorted in an arbitrary way. The VertiPaq engine will certainly include such a sort order in the options to consider.

In order to obtain the maximum compression, you can set the value to 0, which means SSAS stops searching only when it finds the best compression factor. The benefit in terms of space usage and query speed can be very high but, at the same time, processing will take much longer.

Generally, you should try to put the least changing columns first in the sort order, because they are likely to generate many repeating values. Keep in mind, anyway, that finding the best sort order is a very complex task, and it makes sense to spend time on it only when your data model is really a large one (in the order of a few billion rows). Otherwise, the benefit you get from these extreme optimizations is limited.

Once all the columns are compressed, SSAS completes the processing by building calculated columns, hierarchies, and relationships. Hierarchies and relationships are additional data structures needed by VertiPaq to execute queries, whereas calculated columns are added to the model by using DAX expressions.

Calculated columns, like all other columns, are compressed after they are computed. Nevertheless, they are not exactly the same as standard columns. In fact, they are compressed during the final stage of processing, when all the other columns have already finished their compression. Consequently, VertiPaq does not consider them when choosing the best sort order for your table.

Imagine you create a calculated column that results in a Boolean value. Having only two values, it can be compressed very well (1 bit is enough to store a Boolean value) and it is a very good candidate to be first in the sort order list, so that the table shows first all the *TRUE* values and later only the *FALSE* ones. But, being a calculated column, the sort order is already defined and it might be the case that, with the defined sort order, the column frequently changes its value. In such a case, the column results in less-than-optimal compression.

Whenever you have the chance to compute a column in DAX or in SQL, keep in mind that computing it in SQL results in slightly better compression. Obviously, many other factors may drive you to choose DAX instead of SQL to calculate the column. For example, the engine automatically computes a calculated column in a large table depending on a column in a small table, whenever such a small table has a partial or full refresh. This happens without having to reprocess the entire large table, which would be necessary if the computation was in SQL. If you are seeking for optimal compression, this is something you have to consider.

Understanding hierarchies and relationships

As we said in the previous sections, at the end of table processing, SSAS builds two additional data structures: hierarchies and relationships.

Hierarchies are of two types: attribute hierarchies and user hierarchies. They are data structures used to improve performance of MDX queries. Because DAX does not have the concept of hierarchy in the language, hierarchies are not interesting for the topics of this book.

Relationships, on the other hand, play an important role in the VertiPaq engine and, for some extreme optimizations, it is important to understand how they work. Later in this chapter, we will cover the role of relationships in a query. Here we are only interested in defining what relationships are, in terms of VertiPaq.

A relationship is a data structure that maps IDs in one table to row numbers in another table. For example, consider the columns *ProductKey* in Sales and *ProductKey* in Products, used to build a relationship between the two tables. *Product[ProductKey]* is a primary key. You know that it used value encoding and no compression at all, because RLE could not reduce the size of a column without duplicated values. On the other end, *Sales[ProductKey]* is likely dictionary encoded and compressed, because it probably contains many repetitions. The data structures of the two columns are completely different.

Moreover, because you created the relationship, VertiPaq knows that you are likely to use it very often, placing a filter on Product and expecting to filter Sales, too. If every time it needs to move a filter from Product to Sales, VertiPaq had to retrieve values of *Product[ProductKey]*, search them in the dictionary of *Sales[ProductKey]*, and finally retrieve the IDs of *Sales[ProductKey]* to place the filter, then it would result in slow queries.

To improve query performance, VertiPaq stores relationships as pairs of IDs and row numbers. Given the ID of a *Sales[ProductKey]*, it can immediately find the corresponding rows of Product that match the relationship. Relationships are stored in memory, as any other data structure of VertiPaq. In Figure 13-7 you can see how the relationship between Sales and Product is stored.

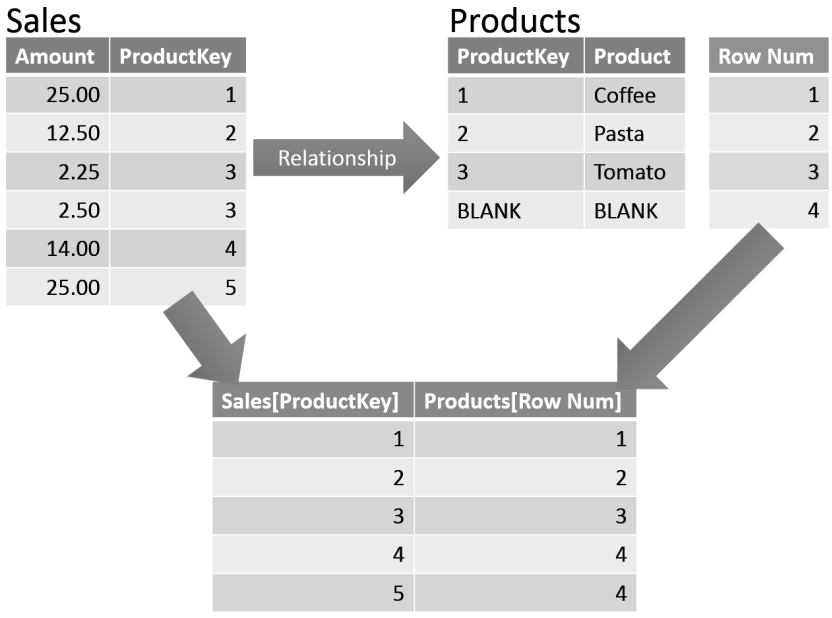


FIGURE 13-7 The figure shows the relationship between Sales and Product.

Understanding segmentation and partitioning

As you might imagine, compressing a table of several billion rows in a single step would be extremely memory-intensive and time-consuming. In fact, the table is not processed as a whole. Instead, during processing SSAS reads it into segments that, by default, contain 8 million rows each. When a segment is completely read, the engine starts to compress it while, in the meantime, it reads the next segment.

You can configure the segment size on SSAS using the *DefaultSegmentRowCount* entry in the configuration file of the service (or in the server properties in Management Studio). In Power Pivot, the segment size has a fixed value of 1 million rows. You cannot change it, because Power Pivot is optimized for smaller datasets.

Segmentation is important for several reasons:

- When querying a table, VertiPaq uses the segments as the basis for parallelism: it uses one core per segment when scanning a column. By default, SSAS always uses one single thread to scan a table with 8 million rows or less. You start observing parallelism in action only on much larger tables.
- The larger the segment, the better the compression. Having the option of analyzing more rows in a single compression step, VertiPaq can achieve better compression levels. On very large tables, it is important to test different segment sizes and measure the memory usage, so to achieve optimal compression. Keep in mind that increasing the segment size can negatively affect processing time: the larger the segment, the slower the processing.
- Although the dictionary is global to the table, bit-sizing happens at the segment level. Thus, if a column has 1,000 distinct values but, in a specific segment, only two of them are used, then that column will be compressed to a single bit for that segment.
- If segments are too small, then the parallelism at query time is increased. This is not always a good thing. In fact, while it is true that scanning the column is faster, VertiPaq needs more time at the end of the scan to aggregate partial results computed by the different threads. If a partition is too small, then the time required for managing task switching and final aggregation is more than the time needed to scan the data, with a negative impact to the overall query performance.

During processing, the first segment has a special treatment if the table has only one partition. In fact, the first segment can be larger than *DefaultSegmentRowCount*. VertiPaq reads twice the size of *DefaultSegmentRowCount* and starts to segment a table only if it contains more rows (but remember that this does not apply to a table with more than one partition). Therefore, a table with 10 million rows will be stored as a single segment, whereas a table with 20 million rows will use three segments: two containing 8 million rows, and one with only 4 million rows.

Segments cannot exceed the partition size. If you have a partitioning schema on your model that creates partitions of only 1 million rows, then all of your segments will be smaller than 1 million rows (namely, they will be same as the partition size). Over-partitioning of tables is a very common mistake of naïve VertiPaq users: remember that creating too many small partitions can only lower the performance.

Using Dynamic Management Views

SSAS lets you discover all the information about the data model using Dynamic Management Views (DMV). DMVs are extremely useful to explore how your model is compressed, the space used by different columns and tables, the number of segments in a table, or the number of bits used by columns in different segments.

You can run DMVs from inside SQL Server Management Studio or, better, using DAX Studio. Moreover, DAX Studio offers you the list of all DMV in a simpler way, without the need to remember them or to reopen this book looking for the DMV name you forgot. You can also use the free tool VertiPaq Analyzer (*http://www.sqlbi.com/tools/vertipaq-analyzer/*) to automatically retrieve data from DMVs and see them in useful reports.

Although DMVs use an SQL-like syntax, you cannot use full SQL syntax to query them, because they do not run inside SQL Server, they are a convenient way to discover the status of SSAS and to gather information about data models. Moreover, DMVs were created when SSAS supported only Multidimensional, so the information provided is not optimized for Tabular. For example, when you query the column information, you get as a result CUBE_NAME, MEASURE_GROUP_NAME, and DIMENSION_NAME, although in VertiPaq there is no concept of cube, measure group, or dimension.

There are different DMVs, divided in two main categories:

- SCHEMA views. These return information about SSAS metadata, such as database names, tables, and individual columns. They do not provide statistical information. Instead, they are used to gather information about datatypes, names, and similar data.
- DISCOVER views. They are intended to gather information about the SSAS engine and/or discover statistics information about objects in a database. For example, you can use views in the discover area to enumerate the DAX keywords, the number of connections and sessions that are currently open, or the traces running.

In this book, we do not want to describe the details of all those views, because they would be off-topic. If you need more information, you can find it in Microsoft documentation on the web. Instead, we want to give some hints and point out the most useful DMVs related to databases used by DAX, which are in the DISCOVER area.

Moreover, while many DMVs report useful information in many columns, in this book we describe the most interesting ones related to the internal structure. For example, there are DMVs to discover the datatypes of all columns, which is not interesting information from the modeling point of view (it might be useful for client tools, but it is useless for the modeler of a solution). On the other hand, knowing the number of bits used for a column in a segment is very technical and definitely useful to optimize a model, so we highlighted it.

Using DISCOVER_OBJECT_MEMORY_USAGE

The first, and probably the most useful DMV lets you discover the memory usage of all the objects in the SSAS instance. This DMV returns information about all the objects in all the databases in the SSAS instance, and it is not limited to the current database.

```
SELECT * FROM $SYSTEM.DISCOVER_OBJECT_MEMORY_USAGE
```

The output of the DMV is a table containing many rows that are very hard to read, because the structure is a parent/child hierarchy that starts with the instance name and ends with individual column information.

The most useful columns in the dataset are as follows:

- OBJECT_ID: Is the ID of the object of which it is reporting memory usage. By itself, it is not a key for the table. You need to combine it with the OBJECT_PARENT_PATH to make it a unique value working as a key.
- OBJECT_PARENT_PATH: Is the full path of the parent in the parent/child hierarchy.
- OBJECT_MEMORY_NON_SHRINKABLE: Is the amount of memory used by the object.

As we said, the raw dataset is nearly impossible to read. However, you can build a Power Pivot data model on top of this query, implementing the parent/child hierarchy structure and browse the full memory map of your instance. Kasper De Jonge published a workbook on his blog that does exactly this, and you can find it here: *http://www.powerpivotblog.nl/what-is-using-all-that-memory-on-my-analysis-server-instance/*

Using DISCOVER_STORAGE_TABLES

The DISCOVER_STORAGE_TABLES DMV is useful to quickly discover tables in a model. It returns only the tables of the current model. In reality, despite its name, it returns tables, hierarchies, and relationships, but the important information is the set of tables.

The most important columns are as follows:

- DIMENSION_NAME: Even if it is named "dimension," for Tabular models it is the table name.
- TABLE_ID: The internal ID of the table, which might be useful to create relationships, because it contains the GUID used by SSAS as a suffix on most table names. Hierarchies and relationships, reported by the same DMV, have an ID starting with H$ and R$, respectively.
- TABLE_PARTITIONS_COUNT: This represents the number of partitions of the table.
- ROWS_COUNT: It is the total number of rows of the table.

A typical usage of this DMV is to run a query similar to the following one, which returns table name and number of rows for only the tables (by checking the first characters of DIMENSION_NAME and TABLE_ID).

```
SELECT
    DIMENSION_NAME AS TABLE_NAME,
    ROWS_COUNT AS ROWS_IN_TABLE
FROM  $SYSTEM.DISCOVER_STORAGE_TABLES
WHERE DIMENSION_NAME = LEFT ( TABLE_ID, LEN ( DIMENSION_NAME ) )
ORDER BY DIMENSION_NAME
```

Note The previous query might not work on Power BI Designer and Power Pivot, because the *LEFT* and *LEN* functions could be not supported. Please consider filtering the result of the DMV in a different way in case you cannot use this technique to obtain only the rows you want.

Using DISCOVER_STORAGE_TABLE_COLUMNS

This DMV gives you detailed information about individual columns, either in tables or in relationships and hierarchies. It is useful to discover, for example, the size of the dictionary, its datatype, and the kind of encoding used for the column.

Most of the information is useful for columns, while it is of less use for hierarchies (either user or system hierarchies). Specifically, for hierarchies, the only useful information is the total size, because other attributes depend directly on the columns they use.

The most relevant columns are as follows:

- DIMENSION_NAME: Even if it is named "dimension," for Tabular models it is the table name.
- TABLE_ID: The internal ID of the table, which might be useful to create relationships, because it contains the GUID used by SSAS as a suffix on most table names. Hierarchies and relationships, reported by the same DMV, have an ID starting with H$ or R$.
- COLUMN_ID: For columns, it is the column name, while for hierarchies it indicates ID_TO_POS or POS_TO_ID, which are internal names for hierarchy structures.
- COLUMN_TYPE: Indicates the type of column. Standard columns contain BASIC_DATA, whereas hierarchies contain different internal names of no interest for this book.
- COLUMN_ENCODING: Indicates the encoding used for the column: 1 stands for hash (dictionary encoding), 2 is value encoding.
- DICTIONARY_SIZE: Is the size, in bytes, of the dictionary of the column.

For example, to retrieve table name, column name, and dictionary size of all columns in your model, you can run this query:

```
SELECT
    DIMENSION_NAME AS TABLE_NAME,
    COLUMN_ID AS COLUMN_NAME,
    DICTIONARY_SIZE AS DICTIONARY_SIZE_BYTES
FROM
    $SYSTEM.DISCOVER_STORAGE_TABLE_COLUMNS
WHERE COLUMN_TYPE = 'BASIC_DATA'
```

Using DISCOVER_STORAGE_TABLE_COLUMN_SEGMENTS

Among the various DMVs, this is the most detailed one, because it reports information about individual segments and partitions of columns. Its content is very detailed and might be overwhelming. Thus, it makes sense to use this information only when you are interested in squeezing the size of a large table, or in performing some kind of extreme optimization.

The most relevant columns are as follows:

- DIMENSION_NAME: Even if it is named "dimension," for Tabular models it is the table name.
- TABLE_ID: The internal ID of the table.
- COLUMN_ID: For columns, it is the column name, whereas for hierarchies it indicates ID_TO_POS or POS_TO_ID, which are internal names for hierarchy structures.
- SEGMENT_NUMBER: The number of the segment reported, zero-based.
- TABLE_PARTITION_NUMBER: The number of the partition to which the segment belongs.
- RECORDS_COUNT: The number of rows in the segment.
- ALLOCATED_SIZE: The size allocated for the segment.
- USED_SIZE: The size actually used for the segment.
- COMPRESSION_TYPE: Indicates the compression algorithm used for the column in the segment. Its content is private and not documented, because the algorithm is patented.
- BITS_COUNT: Number of bits used to represent the column in the segment.
- VERTIPAQ_STATE: can be SKIPPED, COMPLETED, or TIMEBOXED: And it indicates if the engine had the option to find the optimal sorting for the segment (COMPLETED), if it used the best found during the time it was allowed to use but stopped before finding the optimal one (TIMEBOXED), or if the sorting step was skipped (SKIPPED).

Understanding materialization

Now that you have a basic understanding of how VertiPaq stores data in memory, you need to learn what materialization is. Materialization is a step in query resolution that happens when using columnar databases. Understanding when and how it happens is of paramount importance.

In order to understand what materialization is, look at this simple query:

```
EVALUATE
ROW (
    "Result", COUNTROWS ( SUMMARIZE ( Sales, Sales[ProductKey] ) )
)
```

The result is the distinct count of product keys in the Sales table. Even if we have not yet covered the query engine (we will, in Chapter 15, "Analyzing DAX query plans" and Chapter 16, "Optimizing DAX"), you can already imagine how VertiPaq can execute this query. Because the only column queried is *ProductKey*, it can scan that column only, finding all the values in the compressed structure of the column. While scanning, it keeps track of values found in a bitmap index and, at the end, it only has to count the bits that are set. Thanks to parallelism at the segment level, this query can run extremely fast on very large tables and the only memory it has to allocate is the bitmap index to count the keys.

The previous query runs on the compressed version of the column. In other words, there is no need to decompress the columns and to rebuild the original table to resolve it. This optimizes the memory usage at query time and reduces the memory reads.

The same scenario happens for more complex queries, too. Look at the following one:

```
EVALUATE
ROW (
    "Result", CALCULATE (
        COUNTROWS ( Sales ),
        Product[Brand] = "Contoso"
    )
)
```

This time, we are using two different tables: Sales and Product. Solving this query requires a bit more effort. In fact, because the filter is on Product and the table to aggregate is Sales, you cannot scan a single column.

If you are not yet used to columnar databases, you probably think that, to solve the query, you have to iterate the Sales table, follow the relationship with Products, and sum 1 if the product brand is Contoso, 0 otherwise. Thus, you might think of an algorithm similar to this one:

```
EVALUATE
ROW (
    "Result", SUMX (
        Sales,
        IF ( RELATED ( Product[Brand] ) = "Contoso", 1, 0 )
    )
)
```

This is a simple algorithm, but it hides much more complexity than expected. In fact, if you carefully think of the columnar nature of VertiPaq, this query involves three different columns:

- *Product[Brand]* used to filter the Product table.
- *Product[ProductKey]* used to follow the relationship between Product and Sales.
- *Sales[ProductKey]* used on the Sales side to follow the relationship.

Iterating over *Sales[ProductKey]*, searching the row number in Products scanning *Product[ProductKey]*, and finally gathering the brand in *Product[Brand]* would be extremely expensive and require a lot of random reads to memory, negatively affecting performance. In fact, VertiPaq uses a completely different algorithm, optimized for columnar databases.

First, it scans *Product[Brand]* and retrieves the row numbers where *Product[Brand]* is Contoso. As you can see in Figure 13-8, it scans the Brand dictionary (1), retrieves the encoding of Contoso, and finally scans the segments (2) searching for the row numbers where ID equals to 0, returning the indexes to the rows found (3).

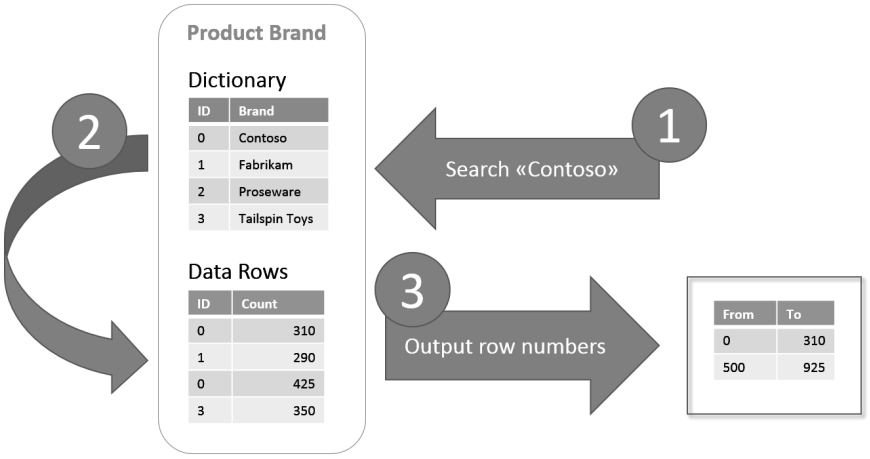


FIGURE 13-8 The output of a brand scan is the list of rows where Brand equals Contoso.

At this point, VertiPaq knows which rows in the Product table have the given brand. The relationship between Product and Sales is based on *Products[ProductKey]* and, at this point VertiPaq knows only the row numbers. Moreover, it is important to remember that the filter will be placed on Sales, not on Products. Thus, in reality, VertiPaq does not need the values of *Products[ProductKey]*, what it really needs is the set of values of *Sales[ProductKey]*, that is, the data IDs in the Sales table, not the ones in Product.

You might remember, at this point, that VertiPaq stores relationships as pairs of row numbers in Product and data IDs in *Sales[ProductKey]*. It turns out that this is the perfect data structure to move the filter from row numbers in Products to *ProductKeys* in Sales. In fact, VertiPaq performs a lookup of the selected row numbers to determine the values of *Sales[ProductKey]* valid for those rows, as you can see in Figure 13-9.

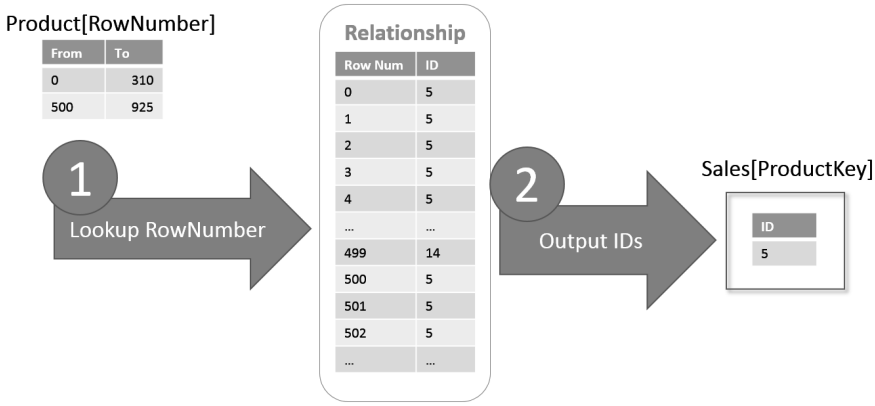


FIGURE 13-9 VertiPaq scans the product key to retrieve the IDs where brand equals Contoso.

The last step is to apply the filter on the Sales table. Since we already have the list of values of *Sales[ProductKey]*, it is enough to scan the *Sales[ProductKey]* column to transform this list of values into row numbers and finally count them. If, instead of computing a *COUNTROWS*, VertiPaq had to perform the *SUM* of a column, then it would perform another step transforming row numbers into column values to perform the last step.

As you can see, this process is made up of simple table scanning where, at each step, you access a single column. However, because data in a column is in the same memory area, VertiPaq sequentially reads blocks of memory and performs simple operations on it, producing every time as output a small data structure that is used in the following step.

The process of resolving a query in VertiPaq is very different from what common sense would suggest. At the beginning, it is very hard to think in terms of columns instead of tables. The algorithms of VertiPaq are optimized for column scanning; the concept of a table is a second-class citizen in a columnar database.

Yet there are scenarios where the engine cannot use these algorithms and reverts to table scanning. Look, for example, at the following query:

```
EVALUATE
ROW (
    "Result", COUNTROWS (
        SUMMARIZE ( Sales, Sales[ProductKey], Sales[CustomerKey] )
    )
)
```

This query looks very innocent, and in its simplicity it shows the limits of columnar databases (but also a row-oriented database faces the same challenge presented here). The query returns the count of unique pairs of product and customer. This query cannot be solved by scanning separately *ProductKey* and *CustomerKey*. The only option here is to build a table containing the unique pairs of *ProductKey* and *CustomerKey*, and finally count the rows in it. Putting it differently, this time VertiPaq has to build a table, even if with only a pair of columns, and it cannot execute the query directly on the original store.

This step, that is, building a table with partial results, which is scanned later to compute the final value, is known as *materialization*. Materialization happens for nearly every query and, by itself, it is neither good nor bad. It all depends on the size of the table materialized. In fact, temporary tables generated by materialization are not compressed (compressing them would take a lot of time, and materialization happens at query time, when latency is extremely important).

It is significant to note that materialization does not happen when you access multiple columns from a table. It all depends on what you have to do with those columns. For example, a query such as the following does not need any materialization, even if it accesses two different columns:

```
EVALUATE
ROW (
    "Result", SUMX (
        Sales, Sales[Quantity] * Sales[Net Price]
    )
)
```

VertiPaq computes the sum performing the multiplication while scanning the two columns, so there is no need to materialize a table with *Quantity* and *Net Price*. Nevertheless, if the expression becomes much more complex, or if you need the table for further processing (as it was the case in the previous example, which required a *COUNTROWS*), then materialization might be required.

In extreme scenarios, materialization might use huge amounts of RAM (sometimes more than the whole database) and generate very slow queries. When this happens, your only chance is to rewrite the calculation or modify the model in such a way that VertiPaq does not need to materialize tables to answer your queries. You will see some examples of these techniques in the following chapters of this book.

Choosing hardware for VertiPaq

Choosing the right hardware is critical for a solution based on SSAS. Spending more does not always mean having a better machine. This final section in the chapter describes how to choose the right server and, as you will see, the perfect Tabular server is not expensive.

Since the introduction of Analysis Services 2012, we helped several companies adopting the new Tabular model in their solutions. A very common issue was that, when going into production, performance was lower than expected. Worse, sometimes it was lower than in the development environments. Most of the times, the reason for that was incorrect hardware sizing, especially when the server was in a virtualized environment. As you will see, the problem is not the use of a virtual machine by itself, but the technical specs of the underlying hardware. A very complete and detailed hardware-sizing guide for Analysis Services Tabular is available in the whitepaper "Hardware Sizing a Tabular Solution (SQL Server Analysis Services)" (*http://msdn.microsoft.com/en-us/library/jj874401.aspx*). The goal of this section is to provide a shorter quick guide that will help you understand the issues affecting many data centers when they have to host a Tabular solution. If you use Power Pivot or Power BI Desktop on a personal computer, you might skip details about Non-Uniform Memory Access (NUMA) support, but all the other considerations are equally true for choosing the right hardware.

Can you choose hardware?

The first question is whether you can choose the hardware or not. The problem of using a virtual machine for a Tabular solution is that often the hardware has already been selected and installed, and you can only influence the number of cores and the amount of RAM that are assigned to your server. Unfortunately, these parameters are not so relevant for the performance. If you will have these limited choices, you should collect information about the CPU model and clock of your host server as soon as possible. If you do not have access to this information, ask a small virtual machine running on the same host server and run the Task Manager: in the Performance tab, you will see the CPU model and the clock rate. With this information, you can predict whether the performance will be worse than an average modern laptop. Unfortunately, chances are that you will be in that position, so you have to sharpen your political skills to convince the right people that running Tabular on that server is a bad idea. If you find that your host server is okay, you will only have to avoid the pitfall of running a Virtual Machine on different NUMA nodes (more on that later).

Set hardware priorities

Assuming that you can influence the hardware selection, keep in mind that you have to set priorities in this order:

1. CPU Clock and Model
2. Memory Speed
3. Number of Cores
4. Memory Size

As you see, disk I/O performance is not in the list, because it is not important at all. There is a condition (paging) where disk I/O affects performance, and we discuss it later in this section. However, you should size the RAM of the system so that you will not have paging at all. Allocate your budget on CPU and memory speed, memory size, and do not waste money on disk I/O bandwidth.

CPU model

The most important factors that affect the speed of code running in the VertiPaq are CPU clock and model. Different CPU models might have a different performance at the same clock rate, so considering the clock alone is not enough. The best practice is to run your own benchmark, measuring the different performance in queries that stress the formula engine. An example of such a query, on a model derived by Adventure Works, is the following:

```
EVALUATE
ROW (
    "Test", COUNTROWS (
        GENERATE (
            TOPN (
                8000,
                CROSSJOIN (
                    ALL ( Reseller[ResellerKey] ),
                    ALL ( Reseller[GeographyKey] )
                ),
                Reseller[ResellerKey]
            ),
            ADDCOLUMNS (
                SUMMARIZE (
                    Sales,
                    OrderDate[FullDate],
                    Products[ProductKey]
                ),
                "Sales", CALCULATE ( SUM ( Sales[SalesAmount] ) )
            )
        )
    )
)
```

You can download the sample workbook to test this query on your hardware here: *http://www.sqlbi.com/articles/choose-the-right-hardware-for-analysis-services-tabular/*. Just open the Excel workbook and run the previous query in DAX Studio, measuring the performance (more on this in Chapter 15).

You can try this query (which is intentionally slow and does not produce any meaningful result) or similar ones. Using a query of a typical workload for your data model is certainly better, because performance might vary on different hardware depending on the memory allocated to materialize intermediate results (the query in the preceding code block has a minimal use of memory).

For example, this query runs in 8.1 seconds on an Intel i7-4770K 3.5 GHz, and in 12.4 seconds on an Intel i7-2860QM 2.5 GHz. These CPUs run a desktop workstation and a notebook, respectively.

Do not presume that a server might run faster. Do your test and look at the results, because they are often surprising. If you do not have Excel on the server, you can restore the Power Pivot model on Analysis Services Tabular and run the query on SQL Server Management Studio if you do not have DAX Studio.

In general, Intel Xeon processors used on a server are E5 and E7 series, and it is very common to find clock speed around 2 GHz, even with a very high number of cores available. You should look for a clock speed of 3 GHz or more, whenever possible. Another important factor is the L2 and L3 cache size: the larger, the better. This is especially important for large tables and relationships between tables based on columns that have more than 1 million unique values.

Memory speed

The memory speed is an important factor for VertiPaq. Every operation made by the engine accesses memory at a very high speed. When the RAM bandwidth is the bottleneck, you see CPU usage instead of I/O waits. Unfortunately, we do not have a performance counter that monitors the time spent a waiting the RAM access. In Tabular, this amount of time can be relevant and it is hard to measure.

In general, you should get RAM that has at least 1,600 MHz, but if the hardware platform permits you should select faster RAM (1,833, 2,133, or 2,400 MHz). At the time of this writing (June 2015), 1,833 MHz is a fast standard on a server, whereas it is hard to find 2,133 MHz, and impossible to find 2,400 MHz unless you buy a desktop optimized to play videogames (by the way, did we mention that gaming machines are the top performers for VertiPaq?).

Number of cores

VertiPaq splits execution on multiple threads only when the table involved has multiple segments. Each segment contains 8 million rows by default (1 million on Power Pivot). If you have eight cores, you will not see all of them involved in a query unless you have at least 64 million rows.

For these reasons, scalability over multiple cores is effective only for very large tables. Raising the number of cores will improve performance for a single query only when it hits a large table (200 million of rows or more). In terms of scalability (number of concurrent users), a higher number of cores might not improve performance if users access the same tables (they would contend access to shared RAM). A better way to increase the number of concurrent users is to use more servers in a load balancing configuration.

The best practice is to get the maximum number of cores you can have on a single socket, getting the highest clock rate available. It is not good having two or more sockets on the same server. Analysis Services Tabular does not recognize the NUMA architecture, which splits memory between different sockets. NUMA requires a more expensive intersocket communication whenever a thread running on a socket accesses memory allocated by another socket—you can find more details about NUMA architecture in Hardware Sizing a Tabular Solution (SQL Server Analysis Services) at *http://msdn.microsoft.com/en-us/library/jj874401.aspx*.

Memory size

You have to store the entire database in memory. You also need RAM to execute process operations (unless you use a separate process server) and to execute queries. Usually optimized queries do not have a high request of RAM, but a single query can materialize temporary tables that could be very large (database tables have a high compression rate, whereas materialization of intermediate tables during a single query generates uncompressed data).

Having enough memory only guarantees that your queries will end returning a result, but increasing available RAM does not produce any performance improvement. Cache used by Tabular does not increase just because of more RAM available. However, a condition of low available memory might affect query performance in a negative way if the server starts paging data. You should simply have enough memory to store all the data of your database and to avoid materialization during query execution.

Disk I/O and paging

You should not allocate budget on storage I/O for Analysis Services Tabular. This is very different from Multidimensional, where random I/O operation on disk occurs very frequently, especially in certain measures. In Tabular, there are no direct storage I/O operations during a query. The only event when this might happen is when you have a low memory condition. However, it is less expensive and more effective to provide more RAM to a server than trying to improve performance by increasing storage I/O throughput when you have a systematic paging caused by low memory available.

Conclusions

You should measure performance before choosing the hardware for SSAS Tabular. It is common to observe a server running twice as slow as a development workstation, even if the server is a very new one. This is because a server designed to be scalable (especially for virtual machines) does not usually perform very well for activities made by a single thread. However, this type of workload is very common in VertiPaq. You will need time and numbers (do your benchmark) to convince your company that a "standard server" could be the weak point of the entire BI solution. Nevertheless, before convincing anybody else, keep in mind that you need to convince yourself. In this chapter, we gave you some insights about the engine. In Chapter 15, you will learn how to measure performance of queries. Take your time and do your tests. We bet they will surprise you.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

About the authors

Marco Russo and Alberto Ferrari

MARCO RUSSO and **ALBERTO FERRARI** are the founders of sqlbi.com, where they regularly publish articles about Microsoft Power Pivot, Power BI, DAX, and SQL Server Analysis Services. They have worked with DAX since the first beta version of Power Pivot in 2009 and, during these years, sqlbi.com became one of the major sources for DAX articles and tutorials.

They both provide consultancy and mentoring on business intelligence (BI), with a particular specialization in the Microsoft technologies related to BI. They have written several books and papers about these topics, with a particular mention of "SQLBI methodology," which is a complete methodology for designing and implementing the back end of a BI solution. They also wrote popular white papers such as "The Many-to-Many Revolution" (about modeling patterns using many-to-many relationships) and "Using Tabular Models in a Large-scale Commercial Solution" (a case study of Analysis Services adoption published by Microsoft).

Marco and Alberto are also regular speakers at major international conferences, including Microsoft Ignite, PASS Summit, and SQLBits. Contact Marco at marco.russo@sqlbi.com and contact Alberto at alberto.ferrari@sqlbi.com.