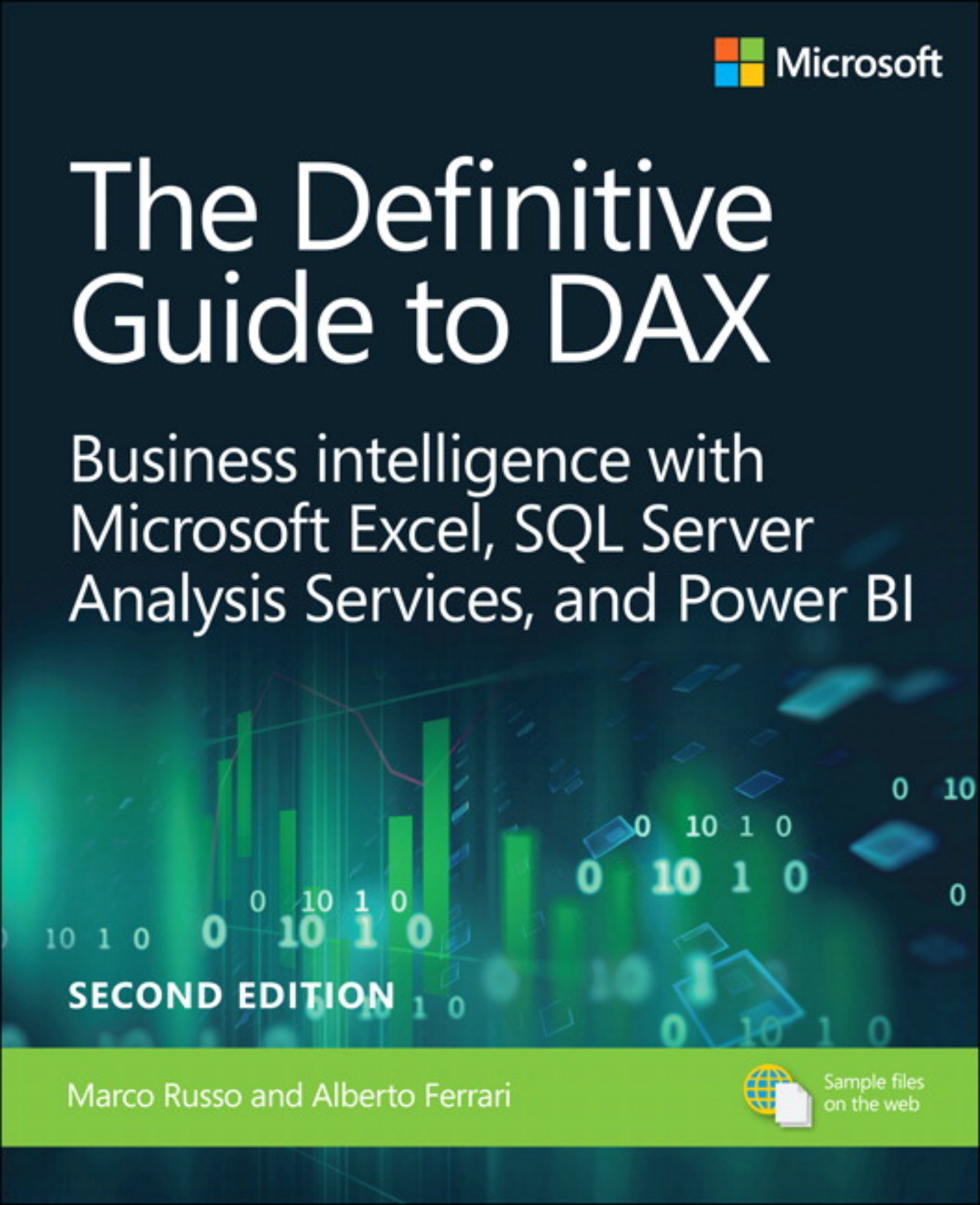
Microsoft

The Definitive Guide to DAX

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



SECOND EDITION

Marco Russo and Alberto Ferrari

Sample files on the web

Microsoft

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

Second Edition

Marco Russo and Alberto Ferrari

Published with the authorization of Microsoft Corporation by:

Pearson Education, Inc.

ISBN-13: 978-1-5093-0697-8

ISBN-10: 1-5093-0697-8

Library of Congress Control Number: 2019930884

ScoutAutomatedPrintCode

EDITOR-IN-CHIEF

Brett Bartow

EXECUTIVE EDITOR

Loretta Yates

DEVELOPMENT EDITOR

Mark Renfrow

MANAGING EDITOR

Sandra Schroeder

SENIOR PROJECT EDITOR

Tonya Simpson

COPY EDITOR

Chuck Hutchinson

INDEXER

Ken Johnson

PROOFREADER

Abigail Manheim

TECHNICAL EDITOR

Daniil Maslyuk

EDITORIAL ASSISTANT

Cindy Teeters

COVER DESIGNER

Twist Creative, Seattle

COMPOSITOR

codeMantra

Contents at a Glance

Contents

Foreword

You may not 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 creation. 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, so no more 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 appreciating 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 have become the source of learning for thousands of people. We write the code, but we do not spend much time teaching developers how to use it; Marco and Alberto are the ones who spread the knowledge about DAX.

Alberto and Marco's books are among a few bestsellers on this topic, and now with this new guide to DAX, they have 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 extract 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

Christian Wade, Senior Program Manager

Acknowledgments

Writing this second edition required an entire year's worth of work, three months more than the first edition. It has been a long and amazing journey, connecting people all around the world in any latitude and time zone to be able to produce the result you are going to read. 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 so. Blog comments, forum posts, email discussions, chats with attendees and speakers at technical conferences, analysis of customer scenarios, and so much more have been useful to us, and many people have contributed significant ideas to this book. Moreover, big thanks to all the students of our courses: By teaching you, we got better!

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

We want to start with Edward Melomed: He has inspired us, and we probably would not have started our journey with the DAX language without a passionate discussion 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: They all greatly helped us along the process of book writing.

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 a lot of their precious time teaching us important concepts about Power BI and DAX: They are Marius Dumitru, Jeffrey Wang, Akshai Mirchandani, Krystian Sakowski, and Cristian Petculescu. Your help has been priceless, guys!

We also want to thank Amir Netz, Christian Wade, 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 tremendously in strategic choices we made in this book and in our career.

We wanted to reserve a special mention to a woman who did an incredible job improving and cleaning up our English. Claire Costa proofread the entire manuscript and made it so much easier to read. Claire, your help is invaluable—Thanks!

The last special mention goes to our technical reviewer: Daniil Maslyuk carefully tested every single line of code, text, example, and reference we had written. He found any and all kinds of mistakes we would have missed. He rarely made comments that did not require a change in the book. The result is amazing for us. If the book contains fewer errors than our original manuscript, it is only because of Daniil's efforts. 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. You can access updates to this book—in the form of a list of submitted errata and their related corrections—at https://MicrosoftPressStore.com/DefinitiveGuideDAX/errata

For additional book support and information, please visit https://MicrosoftPressStore.com/Support.

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*.

Stay in touch

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

Introduction to the second edition

When we decided it was time to update this book, we thought it would be an easy job: After all, not many things have changed in the DAX language, and the theoretical core of the book was still very good. We believed the focus would mainly be on updating the screenshots from Excel to Power BI, adding a few touch-ups here and there, and we would be done. How wrong we were!

As soon as we started updating the first chapter, we quickly discovered that we wanted to rewrite nearly everything. We felt so not only in the first chapter, but at every page of the book. Therefore, this is not really a second edition; it is a brand new book.

The reason is not that the language or the tools have changed so drastically. The reason is that over these last few years we—as authors and teachers—have evolved a lot, hopefully for the better. We have taught DAX to thousands of users and developers all around the world; we worked hard with our students, always striving for the best way to explain complex topics. Eventually, we found different ways of describing the language we love.

We increased the number of examples for this edition, showing practical uses of the functionalities after teaching the theoretical foundation of DAX. We tried to use a simpler style, without compromising on precision. We fought with the editor to increase the page count, as this was needed to cover all the topics we wanted to share. Nevertheless, we did not change the leitmotif of the book: we assume no previous knowledge of DAX, even though this is not a book for the casual DAX developer. This is a book for people who really want to learn the language and gain a deep understanding of the power and complexity of DAX.

Yes, if you want to leverage the real power of DAX, you need to be prepared for a long journey with us, reading the book from cover to cover, and then reading it again, searching for the many details that—at first sight—are not obvious.

Introduction to the first edition

We have created considerable amounts of content on DAX: books about Power Pivot and SSAS Tabular, blog posts, articles, white papers, and finally 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 a definite 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. In the end, nearly everything in a book goes back to the number of pages. As authors, this is somewhat frustrating. In fact, whenever we write a book, we have to carefully allocate space to the description of the product (either Power Pivot for Microsoft Excel or SSAS Tabular) and of to the DAX language. This has always left us with the bitter feeling 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 size would be intimidating for anybody.

Thus, for 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 a 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 so.

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 have already learned other programming and/or query

languages, you are probably used to learning a new language by looking at 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 mighty language, used in a growing number of analytical tools. It is very powerful, but it includes 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 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. The goal of this book here is different: to enable you to 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, 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 the examples. For educational purposes we have often used code that was 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 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 up 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 the most complex parts again, once you have 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: Power BI users might need to author DAX formulas in their models, 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. In this book, we tried to provide information to all 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 Power BI and Excel users too should understand the range of possible performance of DAX expressions to achieve the best results for their models.

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 remain realistic. DAX is simple, but it is not easy. 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 this content in a few days, by reading casually. This book requires your attention at a very high level. In exchange for that, 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 basic knowledge of Power BI and some experience in the analysis of numbers. If you have already had prior exposure to the DAX language, then this is good for you—you will read the first part faster—but of course knowing DAX is not necessary.

There are references throughout the book to MDX and SQL code; however, you do not really need to know these languages because they just reflect comparisons between different ways of writing expressions. If you do not understand those lines of code, it is 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 not everybody might be familiar with. Any developer will feel at home there, whereas Power BI and Excel users might be a bit intimidated. Nevertheless, this information is required in order to discuss DAX optimization. Indeed, the most advanced part of the book is aimed more towards BI developers than towards Power BI and 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: For example, if you are in doubt about the behavior of *ALLSELECTED*, then you can jump straight 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 differences 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, and error-handling functions; we also 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 table functions, whereas we cover advanced table functions in Chapter 12 and 13.
- Chapter 4 describes evaluation contexts. Evaluation contexts are the foundation of the DAX language, so this chapter, along with the next one, is probably the most important in the entire book.
- Chapter 5 only covers 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 describes variables. We use variables in all the examples of the book, but Chapter 6 is where we introduce their syntax and explain how to use variables. This chapter will be useful as a reference when you see countless examples using variables in the following chapters.

- Chapter 7 covers iterators and CALCULATE: a marriage made in heaven. Learning how to use iterators, along with the power of context transition, leverages much of the power of DAX. In this chapter, we show several examples that are useful to understand how to take advantage of these tools.
- Chapter 8 describes 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 9 is dedicated to the latest feature introduced in DAX: calculation groups. Calculation groups are very powerful as a modeling tool. This chapter describes how to create and use calculation groups, introducing the basic concepts and showing a few examples.
- Chapter 10 covers more advanced uses of the filter context, data lineage, inspection of the filter context, and other useful tools to compute advanced formulas.
- Chapter 11 shows you how to perform calculations over hierarchies and how to handle parent/child structures using DAX.
- Chapters 12 and 13 cover advanced table functions that are useful both to author queries and/or to compute advanced calculations.
- Chapter 14 advances 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. This is an advanced chapter that uncovers most of the secrets of complex DAX expressions.
- Chapter 15 is about managing relationships in DAX. Indeed, thanks to DAX any type of relationship can be set within a data model. This chapter includes the description of many types of relationships that are common in an analytical data model.
- Chapter 16 contains several examples of complex calculations solved in DAX. This is the final chapter about the language, useful to discover solutions and new ideas.
- Chapter 17 includes a detailed description of the VertiPaq engine, which is the most common storage engine used by models running DAX. Understanding it is essential to learning how to get the best performance in DAX.
- Chapter 18 uses the knowledge from Chapter 17 to show possible optimizations that you can apply at the data model level. You learn how to reduce the cardinality of columns, how to choose columns to import, and how to improve performance by choosing the proper relationship types and by reducing memory usage in DAX.

- Chapter 19 teaches you how to read a query plan and how to measure the performance of a DAX query with the aid of tools such as DAX Studio and SQL Server Profiler.
- Chapter 20 shows several optimization techniques, based on the content of the previous chapters about optimization. We show many DAX expressions, measure their performance, and then display 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, tables, 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:

MicrosoftPressStore.com/DefinitiveGuideDAX/downloads

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 have enriched with several views, to make it easier to create a data model on top of it.
- A separate Power BI Desktop model for each figure of the book. Every figure has its own file. The data model is almost always the same, but you can use these files to closely follow the steps outlined in the book.

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 become a real DAX champion.

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

We want to give a few words of warning to our readers. The concept of evaluation contexts is simple, 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 on your DAX learning path. We have been teaching DAX to thousands of users in public and private classes, so we know that this is 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 easy: Come back to this chapter, read it again, and you will probably find something new that you missed during your first read.

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

In the rest of the book we will use these concepts. Then in Chapter 14, "Advanced DAX concepts," you will complete your learning of evaluation contexts with expanded tables. Beware that the content of this chapter is not the definitive description of evaluation contexts just yet. A more detailed description of evaluation contexts is the description based on expanded tables, but it would be too hard to learn about expanded tables before having a good understanding of the basics of evaluation contexts. Therefore, we introduce the whole theory in different steps.

Introducing evaluation contexts

There are two evaluation contexts: the filter context and the row context. In the next sections, you learn what they are and how to use them to write DAX code. Before learning what they are, it is important to state one point: They are different concepts, with different functionalities and a completely different usage.

The most common mistake of DAX newbies is that of confusing the two contexts as if the row context was a slight variation of a filter context. This is not the case. The filter context filters data, whereas the row context iterates tables. When DAX is iterating, it is not filtering; and when it is filtering, it is not iterating. Even though this is a simple concept, we know from experience that it is hard to imprint in the mind. Our brain seems to prefer a short path to learning—when it believes there are some similarities, it uses them by merging the two concepts into one. Do not be fooled. Whenever you have the feeling that the two evaluation contexts look the same, stop and repeat this sentence in your mind like a mantra: "The filter context filters, the row context iterates, they are not the same."

An evaluation context is the context under which a DAX expression is evaluated. In fact, any DAX expression can provide different values in different contexts. This behavior is intuitive, and this is the reason why one can write DAX code without learning about evaluation contexts in advance. You probably reached this point in the book having authored DAX code without learning about evaluation contexts. Because you want more, it is now time to be more precise, to set up the foundations of DAX the right way, and to prepare yourself to unleash the full power of DAX.

Understanding filter contexts

Let us begin by understanding what an evaluation context is. All DAX expressions are evaluated inside a context. The context is the "environment" within which the formula is evaluated. For example, consider a measure such as

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

This formula computes the sum of quantity multiplied by price in the *Sales* table. We can use this measure in a report and look at the results, as shown in Figure 4-1.

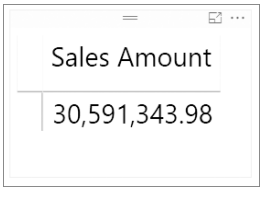


FIGURE 4-1 The measure *Sales Amount*, without a context, shows the grand total of sales.

This number alone does not look interesting. However, if you think carefully, the formula computes exactly what one would expect: the sum of all sales amounts. In a real report, one is likely to slice the value by a certain column. For example, we can select the product brand, use it on the rows, and the matrix report starts to reveal interesting business insights as shown in Figure 4-2.

Brand	Sales Amount
A. Datum	2,096,184.64
Adventure Works	4,011,112.28
Contoso	7,352,399.03
Fabrikam	5,554,015.73
Litware	3,255,704.03
Northwind Traders	1,040,552.13
Proseware	2,546,144.16
Southridge Video	1,384,413.85
Tailspin Toys	325,042.42
The Phone Company	1,123,819.07
Wide World Importers	1,901,956.66
Total	**30,591,343.98**

FIGURE 4-2 Sum of *Sales Amount*, sliced by brand, shows the sales of each brand in separate rows.

The grand total is still there, but now it is the sum of smaller values. Each value, together with all the others, provides more detailed insights. However, you should note that something weird is happening: The formula is not computing what we apparently asked. In fact, inside each cell of the report, the formula is no longer computing the sum of all sales. Instead, it computes the sales of a given brand. Finally, note that nowhere in the code does it say that it can (or should) work on subsets of data. This filtering happens outside of the formula.

Each cell computes a different value because of the *evaluation context* under which DAX executes the formula. You can think of the evaluation context of a formula as the surrounding area of the cell where DAX evaluates the formula.

> **DAX evaluates all formulas within a respective context. Even though the formula is the same, the result is different because DAX executes the same code against different subsets of data.**

This context is named *Filter Context* and, as the name suggests, it is a context that filters tables. Any formula ever authored will have a different value depending on the filter context used to perform its evaluation. This behavior, although intuitive, needs to be well understood because it hides many complexities.

Every cell of the report has a different filter context. You should consider that every cell has a different evaluation—as if it were a different query, independent from the other cells in the same report. The engine might perform some level of internal optimization to improve computation speed, but you should assume that every cell has an independent and autonomous evaluation of the underlying DAX expression. Therefore, the computation of the Total row in Figure 4-2 is not computed by summing the other rows of the report. It is computed by aggregating all the rows of the *Sales* table, although this means other iterations were already computed for the other rows in the same report. Consequently,

depending on the DAX expression, the result in the Total row might display a different result, unrelated to the other rows in the same report.

Note In these examples, we are using a matrix for the sake of simplicity. We 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 only think of reports, to have a simplified and visual understanding of the concepts.

When *Brand* is on the rows, the filter context filters one brand for each cell. If we increase the complexity of the matrix by adding the year on the columns, we obtain the report in Figure 4-3.

Brand	CY 2007	CY 2008	CY 2009	**Total**
A. Datum	1,181,110.71	463,721.61	451,352.33	**2,096,184.64**
Adventure Works	2,249,988.11	892,674.52	868,449.65	**4,011,112.28**
Contoso	2,729,818.54	2,369,167.68	2,253,412.80	**7,352,399.03**
Fabrikam	1,652,751.34	1,993,123.48	1,908,140.91	**5,554,015.73**
Litware	647,385.82	1,487,846.74	1,120,471.47	**3,255,704.03**
Northwind Traders	372,199.93	469,827.70	198,524.49	**1,040,552.13**
Proseware	880,095.80	763,586.23	902,462.12	**2,546,144.16**
Southridge Video	688,107.56	294,635.04	401,671.25	**1,384,413.85**
Tailspin Toys	74,603.14	97,193.87	153,245.41	**325,042.42**
The Phone Company	362,444.46	355,629.36	405,745.25	**1,123,819.07**
Wide World Importers	471,440.71	740,176.76	690,339.18	**1,901,956.66**
Total	**11,309,946.12**	**9,927,582.99**	**9,353,814.87**	**30,591,343.98**

FIGURE 4-3 *Sales amount* is sliced by brand and year.

Now each cell shows a subset of data pertinent to one brand and one year. The reason for this is that the filter context of each cell now filters both the brand and the year. In the Total row, the filter is only on the brand, whereas in the Total column the filter is only on the year. The grand total is the only cell that computes the sum of all sales because—there—the filter context does not apply any filter to the model.

The rules of the game should be clear at this point: The more columns we use to slice and dice, the more columns are being filtered by the filter context in each cell of the matrix. If one adds the *Store[Continent]* column to the rows, the result is—again—different, as shown in Figure 4-4.

Brand	CY 2007	CY 2008	CY 2009	**Total**
A. Datum	**1,181,110.71**	**463,721.61**	**451,352.33**	**2,096,184.64**
Asia	281,936.73	125,055.80	145,386.55	**552,379.08**
Europe	395,159.31	165,924.22	146,867.73	**707,951.26**
North America	504,014.67	172,741.59	159,098.05	**835,854.31**
Adventure Works	**2,249,988.11**	**892,674.52**	**868,449.65**	**4,011,112.28**
Asia	620,545.52	347,150.65	414,507.89	**1,382,204.07**
Europe	662,553.70	275,126.51	264,973.65	**1,202,653.86**
North America	966,888.88	270,397.36	188,968.10	**1,426,254.35**
Contoso	**2,729,818.54**	**2,369,167.68**	**2,253,412.80**	**7,352,399.03**
Asia	838,967.94	998,113.24	753,146.22	**2,590,227.39**
Europe	905,295.91	529,596.05	694,250.12	**2,129,142.08**
North America	985,554.69	841,458.40	806,016.47	**2,633,029.56**
Fabrikam	**1,652,751.34**	**1,993,123.48**	**1,908,140.91**	**5,554,015.73**
Asia	640,664.16	727,025.63	783,871.11	**2,151,560.89**
Europe	503,428.83	383,827.59	454,944.80	**1,342,201.22**
Total	**11,309,946.12**	**9,927,582.99**	**9,353,814.87**	**30,591,343.98**

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

Now the filter context of each cell is filtering brand, country, and year. In other words, the filter context contains the complete set of fields that one uses on rows and columns of the report.

Note Whether a field is on the rows or on the columns of the visual, or on the slicer and/or page/report/visual filter, or in any other kind of filter we can create with a report—all this is irrelevant. All these filters contribute to define a single filter context, which DAX uses to evaluate the formula. Displaying a field on rows or columns is useful for aesthetic purposes, but nothing changes in the way DAX computes values.

Visual interactions in Power BI compose a filter context by combining different elements from a graphical interface. Indeed, the filter context of a cell is computed by merging together all the filters coming from rows, columns, slicers, and any other visual used for filtering. For example, look at Figure 4-5.

Brand	CY 2007	CY 2008	CY 2009	Total
A. Datum	57,276.00			**57,276.00**
Adventure Works	77,413.46		8,110.53	**85,523.99**
Contoso	125,596.01	2,638.18	14,156.95	**142,391.14**
Fabrikam	4,340.62	8,640.00	29,854.98	**42,835.60**
Litware	17,910.87		7,956.00	**25,866.87**
Northwind Traders	34,161.39	12,733.92	2,122.32	**49,017.63**
Proseware	13,183.70		10,647.00	**23,830.70**
Southridge Video	27,239.71	774.23	3,874.18	**31,888.12**
Tailspin Toys	4,581.53	3,976.38	5,886.67	**14,444.57**
The Phone Company	1,384.80	864.90		**2,249.70**
Wide World Importers	2,395.37			**2,395.37**
Total	**365,483.46**	**29,627.61**	**82,608.63**	**477,719.70**

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

The filter context of the top-left cell (A.Datum, CY 2007, 57,276.00) not only filters the row and the column of the visual, but it also filters the occupation (Professional) and the continent (Europe), which are coming from different visuals. All these filters contribute to the definition of a single filter context valid for one cell, which DAX applies to the whole data model prior to evaluating the formula.

A more formal definition of a filter context is to say that a filter context is a set of filters. A filter, in turn, is a list of tuples, and a tuple is a set of values for some defined columns. Figure 4-6 shows a visual representation of the filter context under which the highlighted cell is evaluated. Each element of the report contributes to creating the filter context, and every cell in the report has a different filter context.

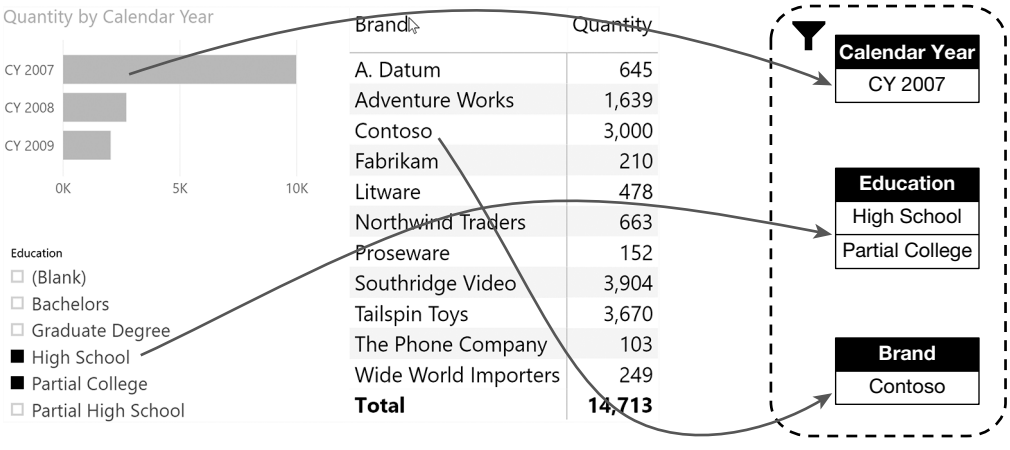


FIGURE 4-6 The figure shows a visual representation of a filter context in a Power BI report.

The filter context of Figure 4-6 contains three filters. The first filter contains a tuple for *Calendar Year* with the value CY 2007. The second filter contains two tuples for *Education* with the values High School and Partial College. The third filter contains a single tuple for *Brand*, with the value Contoso. You might

notice that each filter contains tuples for one column only. You will learn how to create tuples with multiple columns later. Multi-column tuples are both powerful and complex tools in the hand of a DAX developer.

Before leaving this introduction, let us recall the measure used at the beginning of this section:

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

Here is the correct way of reading the previous measure: *The measure computes the sum of Quantity multiplied by Net Price for all the rows in Sales which are visible in the current filter context*.

The same applies to simpler aggregations. For example, consider this measure:

```
Total Quantity := SUM ( Sales[Quantity] )
```

It sums the *Quantity* column of all the rows in *Sales* that are visible in the current filter context. You can better understand its working by considering the corresponding *SUMX* version:

```
Total Quantity := SUMX ( Sales, Sales[Quantity] )
```

Looking at the *SUMX* definition, we might consider that the filter context affects the evaluation of the *Sales* expression, which only returns the rows of the *Sales* table that are visible in the current filter context. This is true, but you should consider that the filter context also applies to the following measures, which do not have a corresponding iterator:

```
Customers := DISTINCTCOUNT ( Sales[CustomerKey] )  -- Count customers in filter context

Colors :=
VAR ListColors = DISTINCT ( 'Product'[Color] )     -- Unique colors in filter context
RETURN COUNTROWS ( ListColors )                    -- Count unique colors
```

It might look pedantic, at this point, to spend so much time stressing the concept that a filter context is always active, and that it affects the formula result. Nevertheless, keep in mind that DAX requires you to be extremely precise. Most of the complexity of DAX is not in learning new functions. Instead, the complexity comes from the presence of many subtle concepts. When these concepts are mixed together, what emerges is a complex scenario. Right now, the filter context is defined by the report. As soon as you learn how to create filter contexts by yourself (a critical skill described in the next chapter), being able to understand which filter context is active in each part of your formula will be of paramount importance.

Understanding the row context

In the previous section, you learned about the filter context. In this section, you now learn the second type of evaluation context: the *row context*. Remember, although both the row context and the filter context are evaluation contexts, *they are not the same concept*. As you learned in the previous section, the purpose of the filter context is, as its name implies, to filter tables. On the other hand, the row context is not a tool to filter tables. Instead, it is used to iterate over tables and evaluate column values.

This time we use a different formula for our considerations, defining a calculated column to compute the gross margin:

```
Sales[Gross Margin] = Sales[Quantity] * ( Sales[Net Price] - Sales[Unit Cost] )
```

There is a different value for each row in the resulting calculated column, as shown in Figure 4-7.

Quantity	Unit Cost	Net Price	Gross Margin
1	915.08	1,989.90	1,074.82
1	960.82	2,464.99	1,504.17
1	1,060.22	2,559.99	1,499.77
1	1,060.22	2,719.99	1,659.77
1	1,060.22	2,879.99	1,819.77
1	1,060.22	3,199.99	2,139.77
2	0.48	0.76	0.56
2	0.48	0.88	0.81
2	1.01	1.79	1.56
2	1.01	1.85	1.68

FIGURE 4-7 There is a different value in each row of *Gross Margin*, depending on the value of other columns.

As expected, for each row of the table there is a different value in the calculated column. Indeed, because there are given values in each row for the three columns used in the expression, it comes as a natural consequence that the final expression computes different values. As it happened with the filter context, the reason is the presence of an evaluation context. This time, the context does not filter a table. Instead, it identifies the row for which the calculation happens.

Note The row context references a row in the result of a DAX table expression. It should not be confused with a row in the report. DAX does not have a way to directly reference a row or a column in the report. The values displayed in a matrix in Power BI and in a PivotTable in Excel are the result of DAX measures computed in a filter context, or are values stored in the table as native or calculated columns.

In other words, we know that a calculated column is computed row by row, but how does DAX know which row it is currently iterating? It knows the row because there is another evaluation context providing the row—it is the *row context*. When we create a calculated column over a table with one million rows, DAX creates a row context that evaluates the expression iterating over the table row by row, using the row context as the cursor.

When we create a calculated column, DAX creates a row context by default. In that case, there is no need to manually create a row context: A calculated column is always executed in a row context. You have already learned how to create a row context manually—by starting an iteration. In fact, one can write the gross margin as a measure, like in the following code:

```
Gross Margin :=
SUMX (
    Sales,
    Sales[Quantity] * ( Sales[Net Price] - Sales[Unit Cost] )
)
```

In this case, because the code is for a measure, there is no automatic row context. *SUMX*, being an iterator, creates a row context that starts iterating over the *Sales* table, row by row. During the iteration, it executes the second expression of *SUMX* inside the row context. Thus, during each step of the iteration, DAX knows which value to use for the three column names used in the expression.

The row context exists when we create a calculated column or when we are computing an expression inside an iteration. There is no other way of creating a row context. Moreover, it helps to think that a row context is needed whenever we want to obtain the value of a column for a certain row. For example, the following measure definition is invalid. Indeed, it tries to compute the value of *Sales[Net Price]* and there is no row context providing the row for which the calculation needs to be executed:

```
Gross Margin := Sales[Quantity] * ( Sales[Net Price] - Sales[Unit Cost] )
```

This same expression is valid when executed for a calculated column, and it is invalid if used in a measure. The reason is not that measures and calculated columns have different ways of using DAX. The reason is that a calculated column has an automatic row context, whereas a measure does not. If one wants to evaluate an expression row by row inside a measure, one needs to start an iteration to create a row context.

Note A column reference requires a row context to return the value of the column from a table. A column reference can be also used as an argument for several DAX functions without a row context. For example, *DISTINCT* and *DISTINCTCOUNT* can have a column reference as a parameter, without defining a row context. Nonetheless, a column reference in a DAX expression requires a row context to be evaluated.

At this point, we need to repeat one important concept: A row context is not a special kind of filter context that filters one row. The row context is not filtering the model in any way; the row context only indicates to DAX which row to use out of a table. If one wants to apply a filter to the model, the tool to use is the filter context. On the other hand, if the user wants to evaluate an expression row by row, then the row context will do the job.

Testing your understanding of evaluation contexts

Before moving on to more complex descriptions about evaluation contexts, it is useful 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 of it. As a hint, try to remember, while thinking, "*The filter context filters*; *the row context iterates. This means that the row context does not filter, and the filter context does not iterate*."

Using *SUM* in a calculated column

The first test uses an aggregator inside a calculated column. What is the result of the following expression, used in a calculated column, in *Sales*?

```
Sales[SumOfSalesQuantity] = SUM ( Sales[Quantity] )
```

Remember, this internally corresponds to this equivalent syntax:

```
Sales[SumOfSalesQuantity] = SUMX ( Sales, Sales[Quantity] )
```

Because it is a calculated column, it is computed row by row in a row context. What number do you expect to see? Choose from these three answers:

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

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

Here is the correct reasoning. You have learned that the formula means, "*the sum of quantity for all the rows visible in the current filter context*." Moreover, because the code is executed for a calculated column, DAX evaluates the formula row by row, in a row context. Nevertheless, the row context is not filtering the table. The only context that can filter the table is the filter context. This turns the question into a different one: What is the filter context, when the formula is evaluated? The answer is straightforward: The filter context is empty. Indeed, the filter context is created by visuals or by queries, and a calculated column is computed at data refresh time when no filtering is happening. Thus, *SUM* works on the whole *Sales* table, aggregating the value of *Sales[Quantity]* for all the rows of *Sales*.

The correct answer is the second answer. This calculated column computes the same value for each row, that is, the grand total of *Sales[Quantity]* repeated for all the rows. Figure 4-8 shows the result of the *SumOfSalesQuantity* calculated column.

Quantity	Unit Cost	Net Price	SumOfSalesQuantity
1	0.48	0.76	140,180.00
1	0.48	0.86	140,180.00
1	0.48	0.88	140,180.00
1	0.48	0.95	140,180.00
1	1.01	1.79	140,180.00
1	1.01	1.85	140,180.00
1	1.01	1.99	140,180.00
1	1.50	2.35	140,180.00
1	1.50	2.50	140,180.00
1	1.50	2.65	140,180.00
1	1.50	2.79	140,180.00
1	1.50	2.94	140,180.00

FIGURE 4-8 *SUM (Sales[Quantity])*, in a calculated column, is computed against the entire database.

This example shows that the two evaluation contexts exist at the same time, but they do not interact. The evaluation contexts both work on the result of a formula, but they do so in different ways. Aggregators like *SUM*, *MIN,* and *MAX* only use the filter context, and they ignore the row context. If you have chosen the first answer, as many students typically do, it is perfectly normal. The thing is that you are still confusing the filter context and the row context. Remember, the filter context filters; the row context iterates. The first answer is the most common, when using intuitive logic, but it is wrong—now you know why. However, if you chose the correct answer ... then we are glad this section helped you in learning the important difference between the two contexts.

Using columns in a measure

The second test is slightly different. Imagine we define the formula for the gross margin in a measure instead of in a calculated column. We have a column with the net price, another column for the product cost, and we write the following expression:

```
GrossMargin% := ( Sales[Net Price] - Sales[Unit Cost] ) / Sales[Unit Cost]
```

What will the result be? As it happened earlier, choose among the three possible answers:

- The expression works correctly, time to test the result in a report.
- An error, we should not even write this formula.
- We can define the formula, but it will return an error when used in a report.

As in the previous test, stop reading, think about the answer, and then read the following explanation.

The code references *Sales[Net Price]* and *Sales[Unit Cost]* without any aggregator. As such, DAX needs to retrieve the value of the columns for a certain row. DAX has no way of detecting which row the formula needs to be computed for because there is no iteration happening and the code is not in a calculated column. In other words, DAX is missing a row context that would make it possible to retrieve a value for the columns that are part of the expression. Remember that a measure does not have an automatic row context; only calculated columns do. If we need a row context in a measure, we should start an iteration.

Thus, the second answer is the correct one. We cannot write the formula because it is syntactically wrong, and we get an error when trying to enter the code.

Using the row context with iterators

You learned that DAX creates a row context whenever we define a calculated column or when we start an iteration with an X-function. When we use a calculated column, the presence of the row context is simple to use and understand. In fact, we can create simple calculated columns without even knowing about the presence of the row context. The reason is that the row context is created automatically by the engine. Therefore, we do not need to worry about the presence of the row context. On the other hand, when using iterators we are responsible for the creation and the handling of the row context. Moreover, by using iterators we can create multiple nested row contexts; this increases the complexity of the code. Therefore, it is important to understand more precisely the behavior of row contexts with iterators.

For example, look at the following DAX measure:

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

Because *SUMX* is an iterator, *SUMX* creates a row context on the *Sales* table and uses it during the iteration. The row context iterates the *Sales* table (first parameter) and provides the current row to the second parameter during the iteration. In other words, DAX evaluates the inner expression (the second parameter of *SUMX*) in a row context containing the currently iterated row on the first parameter.

Please note that the two parameters of *SUMX* use different contexts. In fact, any piece of DAX code works in the context where it is called. Thus, when the expression is executed, there might already be a filter context and one or many row contexts active. Look at the same expression with comments:

```
SUMX (
    Sales,                      -- External filter and row contexts
    Sales[Net Price] * 1.1      -- External filter and row contexts + new row context
)
```

The first parameter, *Sales,* is evaluated using the contexts coming from the caller. The second parameter (the expression) is evaluated using both the external contexts plus the newly created row context.

All iterators behave the same way:

1. Evaluate the first parameter in the existing contexts to determine the rows to scan.
2. Create a new row context for each row of the table evaluated in the previous step.
3. Iterate the table and evaluate the second parameter in the existing evaluation context, including the newly created row context.
4. Aggregate the values computed during the previous step.

Be mindful that the original contexts are still valid inside the expression. Iterators add a new row context; they do not modify existing filter contexts. For example, if the outer filter context contains a filter for the color Red, that filter is still active during the whole iteration. Besides, remember that the row context iterates; it does not filter. Therefore, no matter what, we cannot override the outer filter context using an iterator.

This rule is always valid, but there is an important detail that is not trivial. If the previous contexts already contained a row context for the same table, then the newly created row context hides the previous existing row context on the same table. For DAX newbies, this is a possible source of mistakes. Therefore, we discuss row context hiding in more detail in the next two sections.

Nested row contexts on different tables

The expression evaluated by an iterator can be very complex. Moreover, the expression can, on its own, contain further iterations. At first sight, starting an iteration inside another iteration might look strange. Still, it is a common DAX practice because nesting iterators produce powerful expressions.

For example, the following code contains three nested iterators, and it scans three tables: *Categories, Products*, and *Sales*.

```
SUMX (
    'Product Category',                  -- Scans the Product Category table
    SUMX (                               -- For each category
        RELATEDTABLE ( 'Product' ),      -- Scans the category products
        SUMX (                           -- For each product
            RELATEDTABLE ( Sales )       -- Scans the sales of that product
            Sales[Quantity]              --
                * 'Product'[Unit Price]  -- Computes the sales amount of that sale
                * 'Product Category'[Discount]
        )
    )
)
```

The innermost expression—the multiplication of three factors—references three tables. In fact, three row contexts are opened during that expression evaluation: one for each of the three tables that are currently being iterated. It is also worth noting that the two *RELATEDTABLE* functions return the rows of a related table starting from the current row context. Thus, *RELATEDTABLE (Product)*, being

executed in a row context from the Categories table, returns the products of the given category. The same reasoning applies to *RELATEDTABLE (Sales)*, which returns the sales of the given product.

The previous code is suboptimal in terms of both performance and readability. As a rule, it is fine to nest iterators provided that the number of rows to scan is not too large: hundreds is good, thousands is fine, millions is bad. Otherwise, we may easily hit performance issues. We used the previous code to demonstrate that it is possible to create multiple nested row contexts; we will see more useful examples of nested iterators later in the book. One can express the same calculation in a much faster and readable way by using the following code, which relies on one individual row context and the *RELATED* function:

```
SUMX (
    Sales,
    Sales[Quantity]
        * RELATED ( 'Product'[Unit Price] )
        * RELATED ( 'Product Category'[Discount] )
)
```

Whenever there are multiple row contexts on different tables, one can use them to reference the iterated tables in a single DAX expression. There is one scenario, however, which proves to be challenging. This happens when we nest multiple row contexts on the same table, which is the topic covered in the following section.

Nested row contexts on the same table

The scenario of having nested row contexts on the same table might seem rare. However, it does happen quite often, and more frequently in calculated columns. Imagine we want to rank products based on the list price. The most expensive product should be ranked 1, the second most expensive product should be ranked 2, and so on. We could solve the scenario using the *RANKX* function. But for educational purposes, we show how to solve it using simpler DAX functions.

To compute the ranking, for each product we can count the number of products whose price is higher than the current product's. If there is no product with a higher price than the current product price, then the current product is the most expensive and its ranking is 1. If there is only one product with a higher price, then the ranking is 2. In fact, what we are doing is computing the ranking of a product by counting the number of products with a higher price and adding 1 to the result.

Therefore, one can author a calculated column using this code, where we used **PriceOfCurrentProduct** as a placeholder to indicate the price of the current product.

```
1.  'Product'[UnitPriceRank] =
2.  COUNTROWS (
3.      FILTER (
4.          'Product',
5.          'Product'[Unit Price] > PriceOfCurrentProduct
6.      )
7.  ) + 1
```

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

Focus your attention on line 5 of the previous code. There, the reference to *Product[Unit Price]* refers to the value of *Unit Price* in the current row context. What is the active row context when DAX executes row number 5? There are two row contexts. Because the code is written in a calculated column, there is a default row context automatically created by the engine that scans the *Product* table. Moreover, *FILTER* being an iterator, there is the row context generated by *FILTER* that scans the product table again. This is shown graphically in Figure 4-9.

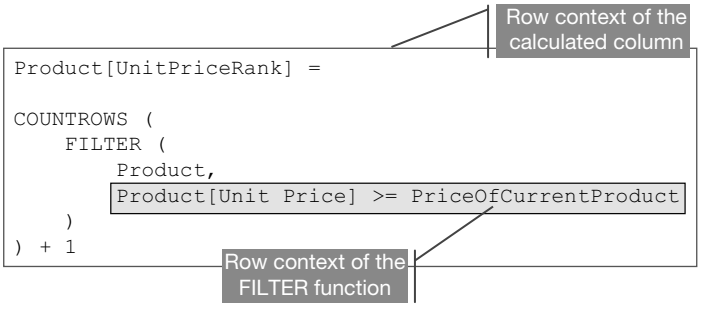


FIGURE 4-9 During the evaluation of the innermost expression, there are two row contexts on the same table.

The outer box includes the row context of the calculated column, which is iterating over *Product*. However, the inner box shows the row context of the *FILTER* function, which is iterating over *Product* too. The expression *Product[Unit Price]* depends on the context. Therefore, a reference to *Product[Unit Price]* in the inner box can only refer to the currently iterated row by *FILTER*. The problem is that, in that box, we need to evaluate the value of *Unit Price* that is referenced by the row context of the calculated column, which is now hidden.

Indeed, when one does not create a new row context using an iterator, the value of *Product[Unit Price]* is the desired value, which is the value in the current row context of the calculated column, as in this simple piece of code:

```
Product[Test] = Product[Unit Price]
```

To further demonstrate this, let us evaluate *Product[Unit Price]* in the two boxes, with some dummy code. What comes out are different results as shown in Figure 4-10, where we added the evaluation of *Product[Unit Price]* right before *COUNTROWS,* only for educational purposes.

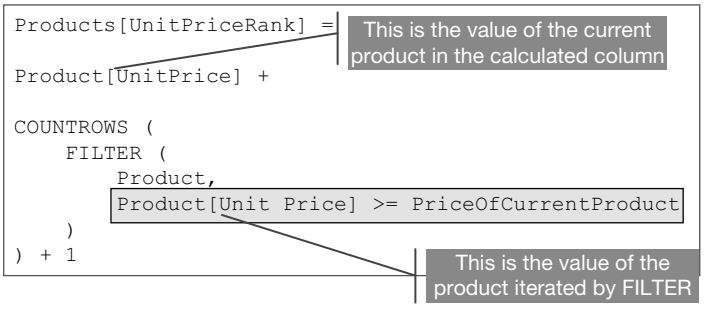


FIGURE 4-10 Outside of the iteration, *Product[Unit Price]* refers to the row context of the calculated column.

Here is a recap of the scenario so far:

- The inner row context, generated by *FILTER*, hides the outer row context.
- We need to compare the inner *Product[Unit Price]* with the value of the outer *Product[Unit Price]*.
- If we write the comparison in the inner expression, we are unable to access the outer *Product[Unit Price]*.

Because we can retrieve the current unit price, if we evaluate it outside of the row context of *FILTER*, the best approach to this problem is saving the value of the *Product[Unit Price]* inside a variable. Indeed, one can evaluate the variable in the row context of the calculated column using this code:

```
'Product'[UnitPriceRank] =
VAR
    PriceOfCurrentProduct = 'Product'[Unit Price]
RETURN
    COUNTROWS (
        FILTER (
           'Product',
           'Product'[Unit Price] > PriceOfCurrentProduct
        )
    ) + 1
```

Moreover, it is even better to write the code in a more descriptive way by using more variables to separate the different steps of the calculation. This way, the code is also easier to follow:

```
'Product'[UnitPriceRank] =
VAR PriceOfCurrentProduct = 'Product'[Unit Price]
VAR MoreExpensiveProducts =
    FILTER (
        'Product',
        'Product'[Unit Price] > PriceOfCurrentProduct
    )
RETURN
    COUNTROWS ( MoreExpensiveProducts ) + 1
```

Figure 4-11 shows a graphical representation of the row contexts of this latter formulation of the code, which makes it easier to understand which row context DAX computes each part of the formula in.

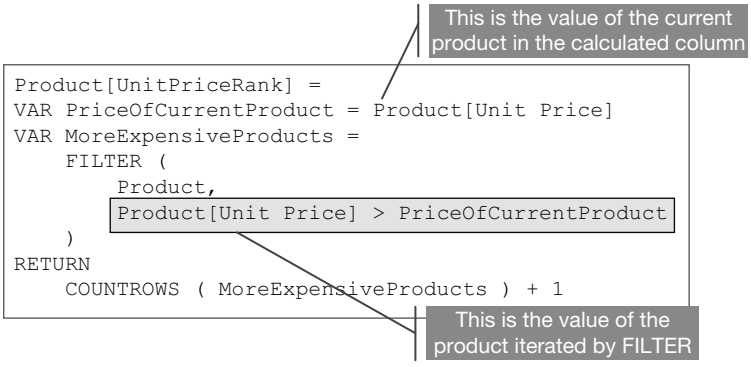


FIGURE 4-11 The value of *PriceOfCurrentProduct* is evaluated in the outer row context.

Figure 4-12 shows the result of this calculated column.

Product Name	Unit Price	UnitPriceRank
Fabrikam Refrigerator 24.7CuFt X9800 Blue	3,199.99	1
Fabrikam Refrigerator 24.7CuFt X9800 Brown	3,199.99	1
Fabrikam Refrigerator 24.7CuFt X9800 Green	3,199.99	1
Fabrikam Refrigerator 24.7CuFt X9800 Grey	3,199.99	1
Fabrikam Refrigerator 24.7CuFt X9800 Orange	3,199.99	1
Fabrikam Refrigerator 24.7CuFt X9800 Silver	3,199.99	1
Fabrikam Refrigerator 24.7CuFt X9800 White	3,199.99	1
Litware Refrigerator 24.7CuFt X980 Blue	3,199.99	1
Litware Refrigerator 24.7CuFt X980 Brown	3,199.99	1
Litware Refrigerator 24.7CuFt X980 Green	3,199.99	1
Litware Refrigerator 24.7CuFt X980 Grey	3,199.99	1
Litware Refrigerator 24.7CuFt X980 Silver	3,199.99	1
Litware Refrigerator 24.7CuFt X980 White	3,199.99	1
Litware Refrigerator L1200 Orange	3,199.99	1
Adventure Works 52" LCD HDTV X590 Black	2,899.99	15
Adventure Works 52" LCD HDTV X590 Brown	2,899.99	15
Adventure Works 52" LCD HDTV X590 Silver	2,899.99	15
Adventure Works 52" LCD HDTV X590 White	2,899.99	15
NT Washer & Dryer 27in L2700 Blue	2,652.90	19
NT Washer & Dryer 27in L2700 Green	2,652.90	19
NT Washer & Dryer 27in L2700 Silver	2,652.90	19

FIGURE 4-12 *UnitPriceRank* is a useful example of how to use variables to navigate within nested row contexts.

Because there are 14 products with the same unit price, their rank is always 1; the fifteenth product has a rank of 15, shared with other products with the same price. It would be great if we could rank 1, 2, 3 instead of 1, 15, 19 as is the case in the figure. We will fix this soon but, before that, it is important to make a small digression.

To solve a scenario like the one proposed, it is necessary to have a solid understanding of what a row context is, to be able to detect which row context is active in different parts of the formula and, most importantly, to conceive how the row context affects the value returned by a DAX expression. It is worth stressing that the same expression *Product[Unit Price]*, evaluated in two different parts of the formula, returns different values because of the different contexts under which it is evaluated. When one does not have a solid understanding of evaluation contexts, it is extremely hard to work on such complex code.

As you have seen, a simple ranking expression with two row contexts proves to be a challenge. Later in Chapter 5 you learn how to create multiple filter contexts. At that point, the complexity of the code increases a lot. However, if you understand evaluation contexts, these scenarios are simple. Before moving to the next level in DAX, you need to understand evaluation contexts well. This is the reason why we urge you to read this whole section again—and maybe the whole chapter so far—until these concepts are crystal clear. It will make reading the next chapters much easier and your learning experience much smoother.

Before leaving this example, we need to solve the last detail—that is, ranking using a sequence of 1, 2, 3 instead of the sequence obtained so far. The solution is easier than expected. In fact, in the previous code we focused on counting the products with a higher price. By doing that, the formula counted 14 products ranked 1 and assigned 15 to the second ranking level. However, counting products is not very useful. If the formula counted the prices higher than the current price, rather than the products, then all 14 products would be collapsed into a single price.

```
'Product'[UnitPriceRankDense] =
VAR PriceOfCurrentProduct = 'Product'[Unit Price]
VAR HigherPrices =
    FILTER (
        VALUES ( 'Product'[Unit Price] ),
        'Product'[Unit Price] > PriceOfCurrentProduct
    )
RETURN
    COUNTROWS ( HigherPrices ) + 1
```

Figure 4-13 shows the new calculated column, along with *UnitPriceRank*.

Product Name	Unit Price	UnitPriceRank	UnitPriceRankDense
Fabrikam Refrigerator 24.7CuFt X9800 Blue	3,199.99	1	1
Fabrikam Refrigerator 24.7CuFt X9800 Brown	3,199.99	1	1
Fabrikam Refrigerator 24.7CuFt X9800 Green	3,199.99	1	1
Fabrikam Refrigerator 24.7CuFt X9800 Grey	3,199.99	1	1
Fabrikam Refrigerator 24.7CuFt X9800 Orange	3,199.99	1	1
Fabrikam Refrigerator 24.7CuFt X9800 Silver	3,199.99	1	1
Fabrikam Refrigerator 24.7CuFt X9800 White	3,199.99	1	1
Litware Refrigerator 24.7CuFt X980 Blue	3,199.99	1	1
Litware Refrigerator 24.7CuFt X980 Brown	3,199.99	1	1
Litware Refrigerator 24.7CuFt X980 Green	3,199.99	1	1
Litware Refrigerator 24.7CuFt X980 Grey	3,199.99	1	1
Litware Refrigerator 24.7CuFt X980 Silver	3,199.99	1	1
Litware Refrigerator 24.7CuFt X980 White	3,199.99	1	1
Litware Refrigerator L1200 Orange	3,199.99	1	1
Adventure Works 52" LCD HDTV X590 Black	2,899.99	15	2
Adventure Works 52" LCD HDTV X590 Brown	2,899.99	15	2
Adventure Works 52" LCD HDTV X590 Silver	2,899.99	15	2
Adventure Works 52" LCD HDTV X590 White	2,899.99	15	2
NT Washer & Dryer 27in L2700 Blue	2,652.90	19	3
NT Washer & Dryer 27in L2700 Green	2,652.90	19	3
NT Washer & Dryer 27in L2700 Silver	2,652.90	19	3
NT Washer & Dryer 27in L2700 White	2,652.90	19	3

FIGURE 4-13 *UnitPriceRankDense* returns a more useful ranking because it counts prices, not products.

This final small step is counting prices instead of counting products, and it might seem harder than expected. The more you work with DAX, the easier it will become to start thinking in terms of ad hoc temporary tables created for the purpose of a calculation.

In this example you learned that the best technique to handle multiple row contexts on the same table is by using variables. Keep in mind that variables were introduced in the DAX language as late as 2015. You might find existing DAX code—written before the age of variables—that uses another technique to access outer row contexts: the *EARLIER* function, which we describe in the next section.

Using the *EARLIER* function

DAX provides a function that accesses the outer row contexts: *EARLIER*. *EARLIER* retrieves the value of a column by using the previous row context instead of the last one. Therefore, we can express the value of **PriceOfCurrentProduct** using *EARLIER (Product[UnitPrice])*.

Many DAX newbies feel intimidated by *EARLIER* because they do not understand row contexts well enough and they do not realize that they can nest row contexts by creating multiple iterations over the

same table. *EARLIER* is a simple function, once you understand the concept of row context and nesting. For example, the following code solves the previous scenario without using variables:

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

Note *EARLIER* accepts a second parameter, which is the number of steps to skip, so that one can skip two or more row contexts. Moreover, there is also a function named *EARLIEST* that lets a developer access the outermost row context defined for a table. In the real world, neither *EARLIEST* nor the second parameter of *EARLIER* is used often. Though having two nested row contexts is a common scenario in calculated columns, having three or more of them is something that rarely happens. Besides, since the advent of variables, *EARLIER* has virtually become useless because variable usage superseded *EARLIER*.

The only reason to learn *EARLIER* is to be able to read existing DAX code. There are no further reasons to use *EARLIER* in newer DAX code because variables are a better way to save the required value when the right row context is accessible. Using variables for this purpose is a best practice and results in more readable code.

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

In the preceding examples, we used *FILTER* as a convenient way of filtering a table. *FILTER* is a common function to use whenever one wants to apply a filter that further restricts the existing filter context.

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

```
NumOfRedProducts :=
VAR RedProducts =
    FILTER (
        'Product',
        'Product'[Color] = "Red"
    )
RETURN
    COUNTROWS ( RedProducts )
```

We can use this formula inside a report. For example, put the product brand on the rows to produce the report shown in Figure 4-14.

Brand	NumOfRedProducts
Adventure Works	6
Contoso	36
Fabrikam	12
Litware	12
Northwind Traders	3
Proseware	7
Southridge Video	13
Tailspin Toys	6
Wide World Importers	4
Total	**99**

FIGURE 4-14 We can count the number of red products using the *FILTER* function.

Before moving on with this example, stop for a moment and think carefully about how DAX computed these values. *Brand* is a column of the *Product* table. Inside each cell of the report, the filter context filters one given brand. Therefore, each cell shows the number of products of the given brand that are also red. The reason for this is that *FILTER* iterates the *Product* table as it is visible in the current filter context, which only contains products with that specific brand. It might seem trivial, but it is better to repeat this a few times than there being a chance of forgetting it.

This is more evident if we add a slicer to the report filtering the color. In Figure 4-15 there are two identical reports with two slicers filtering color, where each slicer only filters the report on its immediate right. The report on the left filters Red and the numbers are the same as in Figure 4-14, whereas the report on the right is empty because the slicer is filtering Azure.

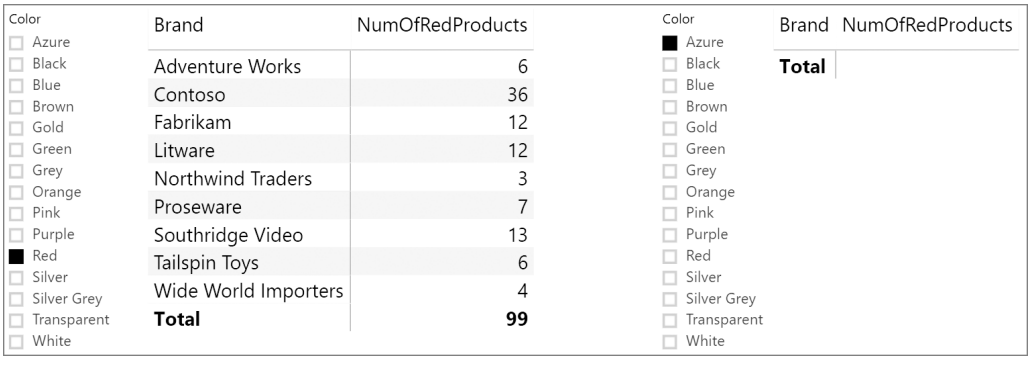


FIGURE 4-15 DAX evaluates *NumOfRedProducts* taking into account the outer context defined by the slicer.

In the report on the right, the *Product* table iterated by *FILTER* only contains Azure products, and, because *FILTER* can only return Red products, there are no products to return. As a result, the *NumOfRedProducts* measure always evaluates to blank.

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 cell in the report, which is affected by the slicer selection—and a row context introduced in the formula by the *FILTER* function. Both contexts work at the same time and modify the result. DAX uses the filter context to evaluate the *Product* table, and the row context to evaluate the filter condition row by row during the iteration made by *FILTER*.

We want to repeat this concept again: *FILTER* does not change the filter context. *FILTER* is an iterator that scans a table (already filtered by the filter context) and it returns a subset of that table, according to the filtering condition. In Figure 4-14, the filter context is filtering the brand and, after *FILTER* returned the result, it still only filtered the brand. Once we added the slicer on the color in Figure 4-15, the filter context contained both the brand and the color. For this reason, in the left-hand side report *FILTER* returned all the products iterated, and in the right-hand side report it did not return any product. In both reports, *FILTER* did not change the filter context. *FILTER* only scanned a table and returned a filtered result.

At this point, one might want to define another formula that returns the number of red products regardless of the selection done on the slicer. In other words, the code needs to ignore the selection made on the slicer and must always return the number of all the red products.

To accomplish this, the *ALL* function comes in handy. *ALL* returns the content of a table *ignoring the filter context*. We can define a new measure, named *NumOfAllRedProducts*, by using this expression:

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

This time, *FILTER* does not iterate *Product*. Instead, it iterates *ALL (Product)*.

ALL ignores the filter context and always returns all the rows of the table, so that *FILTER* returns the red products even if products were previously filtered by another brand or color.

The result shown in Figure 4-16—although correct—might be surprising.

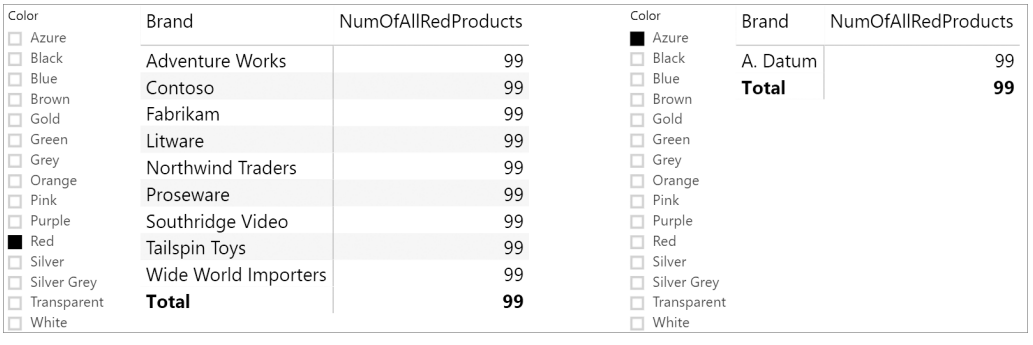


FIGURE 4-16 *NumOfAllRedProducts* returns strange results.

There are a couple of interesting things to note here, and we want to describe both in more detail:

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

First, 99 is the total number of red products, not the number of red products of any given brand. *ALL*—as expected—ignores the filters on the *Product* table. It not only ignores the filter on the color, but it also ignores the filter on the brand. This might be an undesired effect. Nonetheless, *ALL* is easy and powerful, but it is an all-or-nothing function. If used, *ALL* ignores all the filters applied to the table specified as its argument. With the knowledge you have gained so far, you cannot yet choose to only ignore part of the filter. In the example, it would have been better to only ignore the filter on the color. Only after the next chapter, with the introduction of *CALCULATE*, will you have better options to achieve the selective ignoring of filters.

Let us now describe the second point: The brands on the two reports are different. Because the slicer is filtering one color, the full matrix is computed with the filter on the color. On the left the color is Red, whereas on the right the color is Azure. This determines two different sets of products, and consequently, of brands. The list of brands used to populate the axis of the report is computed in the original filter context, which contains a filter on color. Once the axes have been computed, then DAX computes values for the measure, always returning 99 as a result regardless of the brand and color. Thus, the report on the left shows the brands of red products, whereas the report on the right shows the brands of azure products, although in both reports the measure shows the total of all the red products, regardless of their brand.

Note The behavior of the report is not specific to DAX, but rather to the *SUMMARIZECOLUMNS* function used by Power BI. We cover *SUMMARIZECOLUMNS* in Chapter 13, "Authoring queries."

We do not want to further explore this scenario right now. The solution comes later when you learn *CALCULATE*, which offers a lot more power (and complexity) for the handling of filter contexts. As of now, we used this example to show that you might find unexpected results from relatively simple formulas because of context interactions and the coexistence, in the same expression, of filter and row contexts.

Working with several tables

Now that you have learned the basics of evaluation contexts, we can describe how the context behaves when it comes to relationships. In fact, few data models contain just one single table. There would most likely be several tables, linked by relationships. If there is a relationship between *Sales* and *Product*, does a filter context on *Product* filter *Sales*, too? And what about a filter on *Sales*, is it filtering *Product*? Because there are two types of evaluation contexts (the row context and the filter context) and relationships have two sides (a one-side and a many-side), there are four different scenarios to analyze.

The answer to these questions is already found in the mantra you are learning in this chapter, "*The filter context filters; the row context iterates*" and in its consequence, "*The filter context does not iterate; the row context does not filter.*"

To examine the scenario, we use a data model containing six tables, as shown in Figure 4-17.

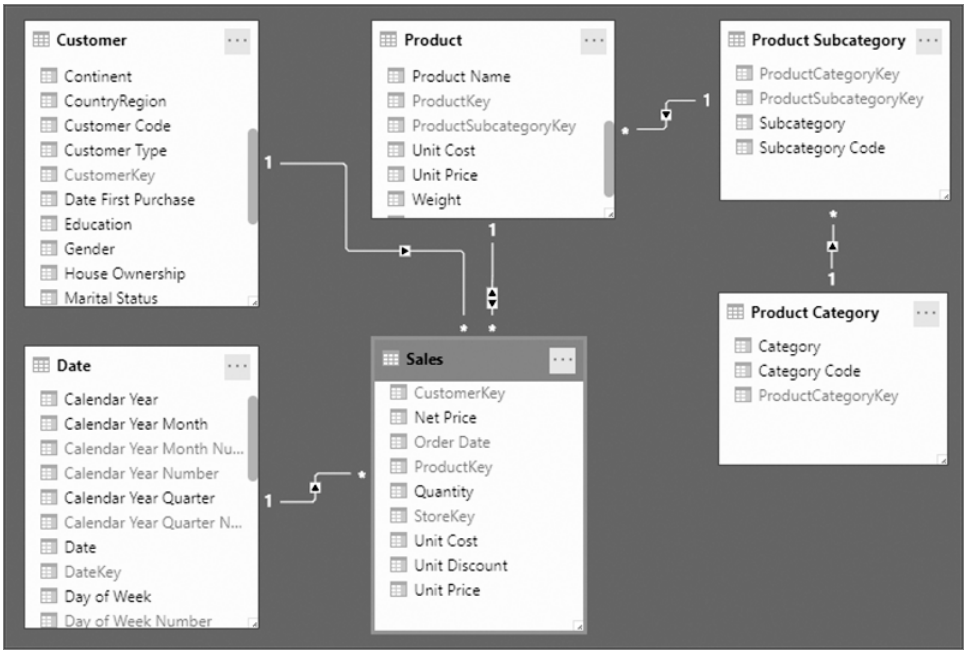


FIGURE 4-17 Data model used to learn the interaction between contexts and relationships.

The model presents a couple of noteworthy details:

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

This model is going to be useful when looking at the details of evaluation contexts and relationships in the next sections.

Row contexts and relationships

The row context iterates; it does not filter. Iteration is the process of scanning a table row by row and of performing an operation in the meantime. Usually, one wants some kind of aggregation like sum or average. During an iteration, the row context is iterating an individual table, and it provides a value to

all the columns of the table, and only that table. Other tables, although related to the iterated table, do not have a row context on them. In other words, the row context does not interact automatically with relationships.

Consider as an example a calculated column in the *Sales* table containing the difference between the unit price stored in the fact table and the unit price stored in the *Product* table. The following DAX code does not work because it uses the *Product[UnitPrice]* column and there is no row context on *Product*:

```
Sales[UnitPriceVariance] = Sales[Unit Price] – 'Product'[Unit Price]
```

This being a calculated column, DAX automatically generates a row context on the table containing the column, which is the *Sales* table. The row context on *Sales* provides a row-by-row evaluation of expressions using the columns in *Sales*. Even though *Product* is on the one-side of a one-to-many relationship with *Sales*, the iteration is happening on the *Sales* table only.

When we are iterating on the many-side of a relationship, we can access columns on the one-side of the relationship, but we must use the *RELATED* function. *RELATED* accepts a column reference as the parameter and retrieves the value of the column in the corresponding row in the target table. *RELATED* can only reference one column and multiple *RELATED* functions are required to access more than one column on the one-side of the relationship. The correct version of the previous code is the following:

```
Sales[UnitPriceVariance] = Sales[Unit Price] - RELATED ( 'Product'[Unit Price] )
```

RELATED requires a row context (that is, an iteration) on the table on the many-side of a relationship. If the row context were active on the one-side of a relationship, then *RELATED* would no longer be useful because *RELATED* would find multiple rows by following the relationship. In this case, that is, when iterating the one-side of a relationship, the function to use is *RELATEDTABLE*. *RELATEDTABLE* returns all the rows of the table on the many-side that are related with the currently iterated table. For example, if one wants to compute the number of sales of each product, the following formula defined as a calculated column on Product solves the problem:

```
Product[NumberOfSales] =
VAR SalesOfCurrentProduct = RELATEDTABLE ( Sales )
RETURN
    COUNTROWS ( SalesOfCurrentProduct )
```

This expression counts the number of rows in the *Sales* table that corresponds to the current product. The result is visible in Figure 4-18.

Product Name	NumberOfSales
A. Datum Advanced Digital Camera M300 Azure	13
A. Datum Advanced Digital Camera M300 Black	23
A. Datum Advanced Digital Camera M300 Green	32
A. Datum Advanced Digital Camera M300 Grey	32
A. Datum Advanced Digital Camera M300 Orange	3
A. Datum Advanced Digital Camera M300 Pink	41
A. Datum Advanced Digital Camera M300 Silver	18
A. Datum All in One Digital Camera M200 Azure	29
A. Datum All in One Digital Camera M200 Black	16
A. Datum All in One Digital Camera M200 Green	19
A. Datum All in One Digital Camera M200 Grey	51

FIGURE 4-18 *RELATEDTABLE* is useful in a row context on the one-side of the relationship.

Both *RELATED* and *RELATEDTABLE* can traverse a chain of relationships; they are not limited to a single hop. For example, one can create a column with the same code as before but, this time, in the *Product Category* table:

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

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.

In a similar way, one can create a calculated column in the *Product* table that copies the category name from the *Product Category* table.

```
'Product'[Category] = RELATED ( 'Product Category'[Category] )
```

In this case, a single *RELATED* function traverses the chain of relationships from *Product* to *Product Subcategory* to *Product Category*.

Note The only exception to the general rule of *RELATED* and *RELATEDTABLE* is for one-to-one relationships. If two tables share a one-to-one relationship, then both *RELATED* and *RELATEDTABLE* work in both tables and they result either in a column value or in a table with a single row, depending on the function used.

Regarding chains of relationships, all the relationships need to be of the same type—that is, one-to-many or many-to-one. If the chain links two tables through a one-to-many relationship to a bridge table, followed by a many-to-one relationship to the second table, then neither *RELATED* nor *RELATEDTABLE* works with single-direction filter propagation. Only *RELATEDTABLE* can work using bidirectional

filter propagation, as explained later. On the other hand, a one-to-one relationship behaves as a one-to-many and as a many-to-one relationship at the same time. Thus, there can be a one-to-one relationship in a chain of one-to-many (or many-to-one) without interrupting the chain.

For example, in the model we chose as a reference, *Customer* is related to *Sales* and *Sales* is related to *Produc*t. 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 *Customer* to *Product*. However, the two relationships are not in the same direction. This scenario is known as a many-to-many relationship. A customer is related to many products bought and a product is in turn related to many customers who bought that product. We cover many-to-many relationships later in Chapter 15, "Advanced relationships"; let us focus on row context, for the moment. If one uses *RELATEDTABLE* through a many-to-many relationship, the result would be wrong. Consider a calculated column in *Product* with this formula:

```
Product[NumOfBuyingCustomers] =
VAR CustomersOfCurrentProduct = RELATEDTABLE ( Customer )
RETURN
    COUNTROWS ( CustomersOfCurrentProduct )
```

The result of the previous code is not the number of customers who bought that product. Instead, the result is the total number of customers, as shown in Figure 4-19.

Product Name	NumOfBuyingCustomers
A. Datum Advanced Digital Camera M300 Azure	18869
A. Datum Advanced Digital Camera M300 Black	18869
A. Datum Advanced Digital Camera M300 Green	18869
A. Datum Advanced Digital Camera M300 Grey	18869
A. Datum Advanced Digital Camera M300 Orange	18869
A. Datum Advanced Digital Camera M300 Pink	18869
A. Datum Advanced Digital Camera M300 Silver	18869
A. Datum All in One Digital Camera M200 Azure	18869
A. Datum All in One Digital Camera M200 Black	18869
A. Datum All in One Digital Camera M200 Green	18869

FIGURE 4-19 *RELATEDTABLE* does not work over a many-to-many relationship.

RELATEDTABLE cannot follow the chain of relationships because they are not going in the same direction. The row context from *Product* does not reach *Customers*. It is worth noting that if we try the formula in the opposite direction, that is, if we count the number of products bought for each customer, the result is 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 row context but, rather, the context transition generated by *RELATEDTABLE*. We added this final note for full disclosure. It is not time to elaborate on this just yet. You will have a better understanding of this after reading Chapter 5.

Filter context and relationships

In the previous section, you learned that the row context iterates and, as such, that it does not use relationships. The filter context, on the other hand, filters. A filter context is not applied to an individual table. Instead, it always works on the whole model. At this point, you can update the evaluation context mantra to its complete formulation:

The filter context filters the model; the row context iterates one table.

Because a filter context filters the model, it uses relationships. The filter context interacts with relationships automatically, and it behaves differently depending on how the cross-filter direction of the relationship is set. The cross-filter direction is represented with a small arrow in the middle of a relationship, as shown in Figure 4-20.

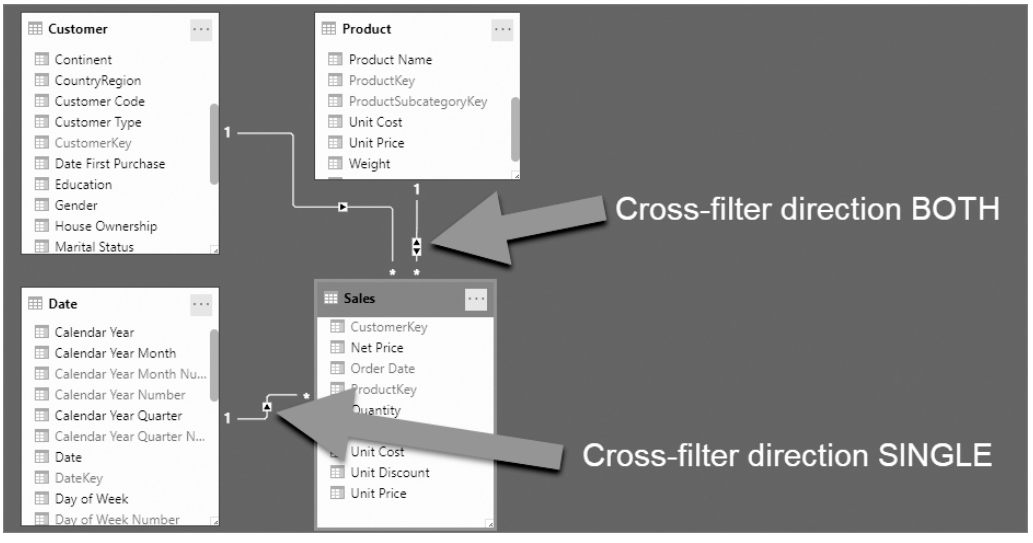


FIGURE 4-20 Behavior of filter context and relationships.

The filter context uses a relationship by going in the direction allowed by the arrow. In all relationships the arrow allows propagation from the one-side to the many-side, whereas when the cross-filter direction is *BOTH*, propagation is allowed from the many-side to the one-side too.

A relationship with a single cross-filter is a *unidirectional relationship*, whereas a relationship with *BOTH* cross-filter directions is a *bidirectional relationship*.

This behavior is intuitive. Although we have not explained this sooner, all the reports we have used so far relied on this behavior. Indeed, in a typical report filtering by *Product[Color]* and aggregating the *Sales[Quantity]*, one would expect the filter from *Product* to propagate to *Sales*. This is exactly what happens: *Product* is on the one-side of a relationship; thus a filter on *Product* propagates to *Sales*, regardless of the cross-filter direction.

Because our sample data model contains both a bidirectional relationship and many unidirectional relationships, we can demonstrate the filtering behavior by using three different measures that count the number of rows in the three tables: *Sales*, *Product*, and *Customer*.

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

The report contains the *Product[Color]* on the rows. Therefore, each cell is evaluated in a filter context that filters the product color. Figure 4-21 shows the result.

Color	NumOfSales	NumOfProducts	NumOfCustomers
Azure	398	14	18,869
Black	24,048	602	18,869
Blue	6,277	200	18,869
Brown	1,840	77	18,869
Gold	988	50	18,869
Green	2,150	74	18,869
Grey	8,525	283	18,869
Orange	1,577	55	18,869
Pink	3,518	84	18,869
Purple	75	6	18,869
Red	5,802	99	18,869
Silver	19,735	417	18,869
Silver Grey	675	14	18,869
Transparent	896	1	18,869
White	21,854	505	18,869
Yellow	1,873	36	18,869
Total	**100,231**	**2,517**	**18,869**

FIGURE 4-21 This shows the behavior of filter context and relationships.

In this first example, the filter is always propagating from the one-side to the many-side of relationships. The filter starts from *Product[Color]*. From there, it reaches *Sales*, which is on the many-side of the relationship with *Product*, and *Product*, because it is the very same table. On the other hand, *NumOfCustomers* always shows the same value—the total number of customers. This is because the relationship between *Customer* and *Sales* does not allow propagation from *Sales* to *Customer*. The filter is moved from *Product* to *Sales*, but from there it does not reach *Customer*.

You might have noticed that the relationship between *Sales* and *Product* is a bidirectional relationship. Thus, a filter context on *Customer* also filters *Sales* and *Product*. We can prove it by changing the report, slicing by *Customer[Education]* instead of *Product[Color]*. The result is visible in Figure 4-22.

Education	NumOfSales	NumOfProducts	NumOfCustomers
	78,059	2,097	385
Bachelors	5,963	415	5,356
Graduate Degree	3,351	290	3,189
High School	4,721	392	3,294
Partial College	5,747	423	5,064
Partial High School	2,390	263	1,581
Total	**100,231**	**2,517**	**18,869**

FIGURE 4-22 Filtering by customer education, the *Product* table is filtered too.

This time the filter starts from *Customer*. It can reach the *Sales* table because *Sales* is on the many-side of the relationship. Furthermore, it propagates from *Sales* to *Product* because the relationship between *Sales* and *Product* is bidirectional—its cross-filter direction is *BOTH*.

Beware that a single bidirectional relationship in a chain does not make the whole chain bidirectional. In fact, a similar measure that counts the number of subcategories, such as the following one, demonstrates that the filter context starting from *Customer* does not reach *Product Subcategory*:

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

Adding the measure to the previous report produces the results shown in Figure 4-23, where the number of subcategories is the same for all the rows.

Education	NumOfSales	NumOfProducts	NumOfCustomers	NumOfSubcategories
	78,059	2,097	385	44
Bachelors	5,963	415	5,356	44
Graduate Degree	3,351	290	3,189	44
High School	4,721	392	3,294	44
Partial College	5,747	423	5,064	44
Partial High School	2,390	263	1,581	44
Total	**100,231**	**2,517**	**18,869**	**44**

FIGURE 4-23 If the relationship is unidirectional, customers cannot filter subcategories.

Because the relationship between *Product* and *Product Subcategory* is unidirectional, the filter does not propagate to *Product Subcategory*. If we update the relationship, setting the cross-filter direction to *BOTH*, the result is different as shown in Figure 4-24.

Education	NumOfSales	NumOfProducts	NumOfCustomers	NumOfSubcategories
	78,059	2,097	385	32
Bachelors	5,963	415	5,356	32
Graduate Degree	3,351	290	3,189	32
High School	4,721	392	3,294	32
Partial College	5,747	423	5,064	32
Partial High School	2,390	263	1,581	31
Total	**100,231**	**2,517**	**18,869**	**44**

FIGURE 4-24 If the relationship is bidirectional, customers can filter subcategories too.

With the row context, we use *RELATED* and *RELATEDTABLE* to propagate the row context through relationships. On the other hand, with the filter context, no functions are needed to propagate the filter. The filter context filters the model, not a table. As such, once one applies a filter context, the entire model is subject to the filter according to the relationships.

Important From the examples, it may look like enabling bidirectional filtering on all the relationships is a good option to let the filter context propagate to the whole model. **This is definitely not the case.** We will cover advanced relationships in depth later, in Chapter 15. Bidirectional filters come with a lot more complexity than what we can share with this introductory chapter, and you should not use them unless you have a clear idea of the consequences. As a rule, you should enable bidirectional filters in specific measures by using the *CROSSFILTER* function, and only when strictly required.

Using *DISTINCT* and *SUMMARIZE* in filter contexts

Now that you have a solid understanding of evaluation contexts, we can use this knowledge to solve a scenario step-by-step. In the meantime, we provide the analysis of a few details that—hopefully—will shed more light on the fundamental concepts of row context and filter context. Besides, in this example we also further describe the *SUMMARIZE* function, briefly introduced in Chapter 3, "Using basic table functions."

Before going into more details, please note that this example shows several inaccurate calculations before reaching the correct solution. The purpose is educational because we want to teach the process of writing DAX code rather than give a solution. In the process of authoring a measure, it is likely you will make several initial errors. In this guided example, we describe the correct way of reasoning, which helps you solve similar errors by yourself.

The requirement is to compute the average age of customers of Contoso. Even though this looks like a legitimate requirement, it is not complete. Are we speaking about their current age or their age at the time of the sale? If a customer buys three times, should it count as one event or as three events in the average? What if they buy three times at different ages? We need to be more precise. Here is the more complete requirement: "*Compute the average age of customers at the time of sale, counting each customer only once if they made multiple purchases at the same age.*"

The solution can be split into two steps:

- Computing the age of the customer when the sale happened
- Averaging it

The age of the customer changes for every sale. Thus, the age needs to be stored in the *Sales* table. For each row in *Sales*, one can compute the age of the customer at the time when the sale happened. A calculated column perfectly fits this need:

```
Sales[Customer Age] =
DATEDIFF (                          -- Compute the difference between
    RELATED ( Customer[Birth Date] ), -- the customer's birth date
    Sales[Order Date],              -- and the date of the sale
    YEAR                            -- in years
)
```

Because *Customer Age* is a calculated column, it is evaluated in a row context that iterates *Sales*. The formula needs to access *Customer[Birth Date]*, which is a column in *Customer*, on the one-side of a relationship with *Sales*. In this case, *RELATED* is needed to let DAX access the target table. In the sample database Contoso, there are many customers for whom the birth date is blank. *DATEDIFF* returns blank if the first parameter is blank.

Because the requirement is to provide the average, a first—and inaccurate—solution might be a measure that averages this column:

```
Avg Customer Age Wrong := AVERAGE ( Sales[Customer Age] )
```

The result is incorrect because *Sales[Customer Age]* contains multiple rows with the same age if a customer made multiple purchases at a certain age. The requirement is to compute each customer only once, and this formula is not following such a requirement. Figure 4-25 shows the result of this last measure side-by-side with the expected result.

Color	Avg Customer Age Wrong	Correct Average
Azure	46.44	46.44
Black	46.59	46.67
Blue	45.87	45.91
Brown	45.48	45.48
Gold	45.26	45.26
Green	47.26	47.26
Grey	46.44	46.44
Orange	37.27	37.27
Pink	46.18	46.17
Purple	50.09	50.09
Red	45.42	45.45
Silver	45.87	45.82
Silver Grey	49.93	49.93
White	46.00	46.25
Yellow	47.76	47.76
Total	**46.18**	**46.20**

FIGURE 4-25 A simple average computes the wrong result for the customer's age.

Here is the problem: The age of each customer must be counted only once. A possible solution—still inaccurate—would be to perform a *DISTINCT* of the customer ages and then average it, with the following measure:

```
Avg Customer Age Wrong Distinct :=
AVERAGEX (                              -- Iterate on the distinct values of
    DISTINCT ( Sales[Customer Age] ),   -- Sales[Customer Age] and compute the
    Sales[Customer Age]                 -- average of the customer's age
)
```

This solution is not the correct one yet. In fact, *DISTINCT* returns the distinct values of the customer age. Two customers with the same age would be counted only once by this formula. The requirement is to count each customer once, whereas this formula is counting each age once. In fact, Figure 4-26 shows the report with the new formulation of *Avg Customer Age*. You see that this solution is still inaccurate.

Color	Avg Customer Age Wrong Distinct	Correct Average
Azure	50.92	46.44
Black	58.38	46.67
Blue	55.33	45.91
Brown	50.15	45.48
Gold	45.14	45.26
Green	50.92	47.26
Grey	54.33	46.44
Orange	38.33	37.27
Pink	53.45	46.17
Purple	53.74	50.09
Red	56.10	45.45
Silver	61.67	45.82
Silver Grey	47.93	49.93
White	58.57	46.25
Yellow	55.83	47.76
Total	**62.00**	**46.20**

FIGURE 4-26 The average of the distinct customer ages still provides a wrong result.

In the last formula, one might try to replace *Customer Age* with *CustomerKey* as the parameter of *DISTINCT*, as in the following code:

```
Avg Customer Age Invalid Syntax :=
AVERAGEX (                              -- Iterate on the distinct values of
    DISTINCT ( Sales[CustomerKey] ),    -- Sales[CustomerKey] and compute the
    Sales[Customer Age]                 -- average of the customer's age
)
```

This code contains an error and DAX will not accept it. Can you spot the reason, without reading the solution we provide in the next paragraph?

AVERAGEX generates a row context that iterates a table. The table provided as the first parameter to *AVERAGEX* is *DISTINCT (Sales[CustomerKey])*. *DISTINCT* returns a table with one column only, and all the unique values of the customer key. Therefore, the row context generated by *AVERAGEX* only contains one column, namely *Sales[CustomerKey]*. DAX cannot evaluate *Sales[Customer Age]* in a row context that only contains *Sales[CustomerKey]*.

What is needed is a row context that has the granularity of *Sales[CustomerKey]* but that also contains *Sales[Customer Age]*. *SUMMARIZE*, introduced in Chapter 3, can generate the existing unique combinations of two columns. Now we can finally show a version of this code that implements all the requirements:

```
Correct Average :=
AVERAGEX (                      -- Iterate on
    SUMMARIZE (                 -- all the existing combinations
        Sales,                  -- that exist in Sales
        Sales[CustomerKey],     -- of the customer key and
        Sales[Customer Age]     -- the customer age
    ),                          --
    Sales[Customer Age]         -- and average the customer's age
)
```

As usual, it is possible to use a variable to split the calculation in multiple steps. Note that the access to the *Customer Age* column still requires a reference to the *Sales* table name in the second argument of the *AVERAGEX* function. A variable can contain a table, but it cannot be used as a table reference.

```
Correct Average :=
VAR CustomersAge =
    SUMMARIZE (                 -- Existing combinations
        Sales,                  -- that exist in Sales
        Sales[CustomerKey],     -- of the customer key and
        Sales[Customer Age]     -- the customer age
    )
RETURN
AVERAGEX (                      -- Iterate on list of
    CustomersAge,               -- Customers/age in Sales
    Sales[Customer Age]         -- and average the customer's age
)
```

SUMMARIZE generates all the combinations of customer and age available in the current filter context. Thus, multiple customers with the same age will duplicate the age, once per customer. *AVERAGEX* ignores the presence of *CustomerKey* in the table; it only uses the customer age. *CustomerKey* is only needed to count the correct number of occurrences of each age.

It is worth stressing that the full measure is executed in the filter context generated by the report. Thus, only the customers who bought something are evaluated and returned by *SUMMARIZE*. Every cell of the report has a different filter context, only considering the customers who purchased at least one product of the color displayed in the report.

Conclusions

It is time to recap the most relevant topics you learned in this chapter about evaluation contexts.

- There are two evaluation contexts: the filter context and the row context. The two evaluation contexts are not variations of the same concept: *the filter context filters the model*; *the row context iterates one table*.
- To understand a formula's behavior, you always need to consider both evaluation contexts because they operate at the same time.
- DAX creates a row context automatically for a calculated column. One can also create a row context programmatically by using an iterator. Every iterator defines a row context.
- You can nest row contexts and, in case they are on the same table, the innermost row context hides the previous row contexts on the same table. Variables are useful to store values retrieved when the required row context is accessible. In earlier versions of DAX where variables were not available, the *EARLIER* function was used to get access to the previous row context. As of today, using *EARLIER* is discouraged.
- When iterating over a table that is the result of a table expression, the row context only contains the columns returned by the table expression.
- Client tools like Power BI create a filter context when you use fields on rows, columns, slicers, and filters. A filter context can also be created programmatically by using *CALCULATE*, which we introduce in the next chapter.
- The row context does not propagate through relationships automatically. One needs to force the propagation by using *RELATED* and *RELATEDTABLE*. You need to use these functions in a row context on the correct side of a one-to-many relationship: *RELATED* on the many-side, *RELATEDTABLE* on the one-side.
- The filter context filters the model, and it uses relationships according to their cross-filter direction. It always propagates from the one-side to the many-side. In addition, if you use the cross-filtering direction *BOTH*, then the propagation also happens from the many-side to the one-side.

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 is a huge chance that was because you have not fully understood these rules.

As we said in the introduction, at first glance all these topics look simple. In fact, they are. What makes them complex is the fact that in a DAX expression you might have several evaluation contexts active in different parts of the formula. Mastering evaluation contexts 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 writing some DAX formulas of your own, you will intuitively know which contexts are used and which functions they require, and you will finally master the DAX language.

CHAPTER 17

The DAX engines

The goal of the book up to this point has been to provide a solid understanding of the DAX language. On top of gaining further experience through practice, the next goal for you is to write efficient DAX and not just DAX that works. Writing efficient DAX requires understanding the internals of the engine. The next chapters aim to provide the essential knowledge to measure and improve DAX code performance.

More specifically, this chapter is dedicated to the internal architecture of the engines running DAX queries. Indeed, a DAX query can run on a model that is stored entirely in memory, or entirely on the original data source, or on a mix of these two options.

Starting from this chapter, we somewhat deviate from DAX and begin to discuss low-level technical details about the implementation of products that use DAX. This is an important topic, but you need to be aware that implementation details change often. We did our best to show information at a level that is not likely to change soon, carefully balancing detail level and usefulness with consistency over time. Nevertheless, given the pace at which technology runs these days, the information might be outdated within a few years. The most up-to-date information is always available online, in blog posts and articles.

New versions of the engines come out every month, and the query optimizer can change and improve the query execution. Therefore, we aim to teach how the engines work, rather than just provide a few rules about writing DAX code that would quickly become obsolete. We sometimes provide best practices, but remember to always double-check how our suggestions apply to your specific scenario.

Understanding the architecture of the DAX engines

The DAX language is used in several Microsoft products based on the Tabular technology. Yet, specific features might only be available in a few editions or license conditions. A Tabular model uses both DAX and MDX as query languages. This section describes the broader architecture of a Tabular model, regardless of the query language and of the limitations of specific products.

Every report sends queries to Tabular using either DAX or MDX. Despite the query language used, the Tabular model uses two engines to process a query:

- The **formula engine** (FE), which processes the request, generating and executing a query plan.
- The **storage engine** (SE), which retrieves data out of the Tabular model to answer the requests made by the Formula Engine. The Storage Engine has two implementations:
  - **VertiPaq** hosts a copy of the data in-memory that is refreshed periodically from the data source.
  - **DirectQuery** forwards queries directly to the original data source for every request. DirectQuery does not create an additional copy of data.

Figure 17-1 represents the architecture that executes a DAX or MDX query.

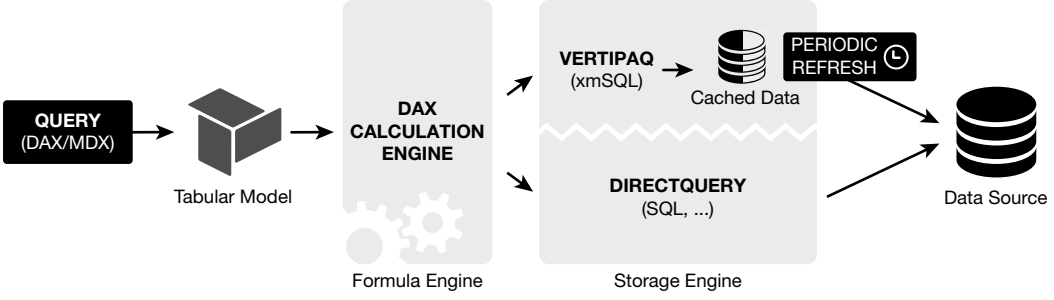


FIGURE 17-1 A query is processed by an architecture using a formula engine and a storage engine.

The formula engine is the higher-level execution unit of the query engine in a Tabular model. It can handle all the operations requested by DAX and MDX functions and can solve complex DAX and MDX expressions. However, when the formula engine must retrieve data from the underlying tables, it forwards part of the requests to the storage engine.

The queries sent to the storage engine might vary from a simple retrieval of the raw table data to more complex queries aggregating data and joining tables. The storage engine only communicates with the formula engine. The storage engine returns data in an uncompressed format, regardless of the original format of the data.

A Tabular model usually stores data using either the VertiPaq or the DirectQuery storage engine. However, composite models can use both technologies within the same data model and for the same tables. The choice of which engine to use is made by the engine on a by-query basis.

This book is exclusively focused on DAX. Be mindful that MDX uses the same architecture when it queries a Tabular model. This chapter describes the different types of storage engines available in a Tabular model, focusing more on the details of the VertiPaq engine because it is the native and faster engine for DAX.

Introducing the formula engine

The formula engine is the absolute core of the DAX execution. Indeed, the formula engine alone is able to understand the DAX language, though it understands MDX as well. The formula engine converts a DAX or MDX query into a query plan describing a list of physical steps to execute. The storage engine part of Tabular is not aware that its queries originated from a model supporting DAX.

Each step in the query plan corresponds to a specific operation executed by the formula engine. Typical operators of the formula engine include joins between tables, filtering with complex conditions, aggregations, and lookups. These operators typically require data from columns in the data model. In these cases, the formula engine sends a request to the storage engine, which answers by returning a datacache. A datacache is a temporary storage area created by the storage engine and read by the formula engine.

Note Datacaches are not compressed; datacaches are plain in-memory tables stored in an uncompressed format, regardless of the storage engine they come from.

The formula engine always works with datacaches returned by the storage engine or with data structures computed by other formula engine operators. The result of a formula engine operation is not persisted in memory across different executions, even within the same session. On the other hand, datacaches are kept in memory and can be reused in following queries. The formula engine does not have a cache system to reuse results between different queries. DAX relies entirely on the cache features of the storage engine.

Finally, the formula engine is single-threaded. This means that any operation executed in the formula engine uses just one thread and one core, no matter how many cores are available. The formula engine sends requests to the storage engine sequentially, one query at a time. A certain degree of parallelism is available only within each request to the storage engine, which has a different architecture and can take advantage of multiple cores available. This is described in the next sections.

Introducing the storage engine

The goal of the storage engine is to scan the Tabular database and produce the datacaches needed by the formula engine. The storage engine is independent from DAX. For example, DirectQuery on top of SQL Server uses SQL as the storage engine. SQL was born much earlier than DAX. Although it might seem strange, the internal storage engine of Tabular (known as VertiPaq) is independent from DAX too. The overall architecture is very clean and sound. The storage engine executes exclusively queries allowed by its own set of operators. Depending on the kind of storage engine used, the set of operators might range from very limited (VertiPaq) to very rich (SQL). This affects the performance and the kind of optimizations that a developer should consider when analyzing query plans.

A developer can define the storage engine used for each table, using one of these three options:

- **Import:** Also called in-memory, or VertiPaq. The content of the table is stored by the VertiPaq engine, copying and restructuring the data from the data source during data refresh.
- **DirectQuery:** The content of the table is read from the data source at query time, and it is not stored in memory during data refresh.
- **Dual:** The table can be queried in both VertiPaq and DirectQuery. During data refresh the table is loaded in memory, but at query time the table may also be read in DirectQuery mode, with the most up-to-date information.

Moreover, a table in a Tabular model could be used as an aggregation for another table. Aggregations are useful to optimize storage engine requests, but not to optimize a bottleneck in the formula engine. Aggregations can be defined in both VertiPaq and DirectQuery, though they are commonly defined in VertiPaq to achieve the best query performance.

The storage engine features a parallel implementation. However, it receives requests from the formula engine, which sends them synchronously. Thus, the formula engine waits for one storage engine query to finish before sending the next one. Therefore, parallelism in the storage engine might be reduced by the lack of parallelism of the formula engine.

Introducing the VertiPaq (in-memory) storage engine

The VertiPaq storage engine is the native lower-level execution unit of the DAX query engine. In certain products it was officially named xVelocity In-Memory Analytical Engine. Nevertheless, it is widely known as VertiPaq, which is the original code name used during development. VertiPaq stores a copy of the data read from the data source in a compressed in-memory format based on a columnar database structure.

VertiPaq queries are expressed using an internal pseudo-SQL language called xmSQL. xmSQL is not a real query language, but rather a textual representation of a storage engine query. The intent of xmSQL is to give visibility to humans as to how the formula engine is querying VertiPaq. VertiPaq offers a very limited set of operators: In case the calculation requires a more complex evaluation within an internal data scan, VertiPaq can perform a callback to the formula engine.

The VertiPaq storage engine is multithreaded. The operations performed by the VertiPaq storage engine are very efficient and can scale up on multiple cores. A single storage engine query can increase its parallelism up to one thread for each segment of a table. We will describe segments later in this chapter. Considering that the storage engine can use up to one thread per column segment, one can benefit from the parallelism of the storage engine only when there are many segments involved in the query. In other words, if there are eight storage engine queries, running on a small table (one segment), they will run sequentially one after the other, instead of all in parallel, because of the synchronous nature of communication between the formula engine and the storage engine.

A cache system stores the results produced by the VertiPaq storage engine, holding a limited number of results—typically the last 512 internal queries per database, but different versions of the engine might use a different number. When the storage engine receives an xmSQL query identical to one already in cache, it returns the corresponding datacache without doing any scan of data in memory. The cache is not involved in security considerations because the row-level security system only influences the formula engine behavior, producing different xmSQL queries in case the user is restricted to seeing specific rows in a table.

A scan operation made by the storage engine is usually faster than the equivalent scan performed by the formula engine, even with a single thread available. This is because the storage engine is better optimized for these operations and because it iterates over compressed data; the formula engine, on the other hand, can only iterate over datacaches, which are uncompressed.

Introducing the DirectQuery storage engine

The DirectQuery storage engine is a generic definition, describing the scenario where the data is kept in the original data source instead of being copied in the VertiPaq storage. When the formula engine sends a request to the storage engine in DirectQuery mode, it sends a query to the data source in its specific query language. This is SQL most of the time, but it could be different.

The formula engine is aware of the presence of DirectQuery. Therefore, the formula engine generates a different query plan compared to VertiPaq because it can take advantage of more advanced functions available in the query language used by the data source. For example, SQL can manage string transformations such as *UPPER* and *LOWER*, whereas the VertiPaq engine does not have any string manipulation functions available.

Any optimization of the storage engine using DirectQuery requires an optimization of the data source—for example, using indexes in a relational database. More details about DirectQuery and the possible optimizations are available in the following white paper: https://www.sqlbi.com/whitepapers/directquery-in-analysis-services-2016/. The considerations are valid for both Power BI and Analysis Services because they share the same underlying engine.

Understanding data refresh

DAX runs on SQL Server Analysis Services (SSAS) Tabular, Azure Analysis Services (same as SSAS in this book), Power BI service (both on server and on the local Power BI Desktop), and in the Power Pivot for Microsoft Excel add-in. Technically, both Power Pivot for Excel and Power BI use a customized version of SSAS Tabular. Speaking about different engines is thus somewhat artificial: Power Pivot and Power BI are like SSAS although SSAS runs in a hidden mode. In this book, we do not discriminate between these engines; when we mention SSAS, the reader should always mentally replace SSAS with Power Pivot or Power BI. If there are differences worth highlighting, then we will note them in that specific section.

When SSAS loads the content of a source table in memory, we say that it processes the table. This takes place during the process operation of SSAS or during the data refresh in Power Pivot for Excel and Power BI. The table process for DirectQuery simply clears the internal cache without executing any access to the data source. On the other hand, when processing occurs in VertiPaq mode, the engine reads the content of the data sources and transforms it into the internal VertiPaq data structure.

VertiPaq processes a table following these few steps:

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

The last two steps are not necessarily sequential. Indeed, a relationship can be based on a calculated column, or calculated columns can depend on a relationship because they use *RELATED* or *CALCULATE*. Therefore, SSAS creates a complex graph of dependencies to execute the steps in the correct order.

In the next sections, we describe these steps in more detail. We also cover the format of the internal structures created by SSAS during the transformation of the data source into the VertiPaq model.

Understanding the VertiPaq storage engine

The VertiPaq engine is the most common storage engine used in Tabular models. VertiPaq is used whenever a table is in Import storage mode. This is the common choice in many data models, and it is the only choice in Power Pivot for Excel. In composite models, the presence of tables or aggregations in dual storage mode also implies the use of the VertiPaq storage engine combined with DirectQuery.

For these reasons, a solid knowledge of the VertiPaq storage engine is a basic skill required to understand how to optimize both the memory consumption of the model and the execution time of the queries. In this section, we describe how the VertiPaq storage works.

Introducing columnar databases

VertiPaq is an in-memory columnar database. Being in-memory means that all the data handled by a model reside in RAM. But VertiPaq is not only in-memory; it is also a columnar database. Therefore, it is relevant to have a good understanding of what a columnar database is in order to correctly understand VertiPaq.

We think of a table as a list of rows, where each row is divided into columns. For example, consider the *Product* table in Figure 17-2.

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 17-2 The figure shows the *Product* table, with four columns and nine rows.

Thinking of a table as a set of rows, we are using the most natural visualization of a table structure. Technically, this is known as a *row store*. In a row store, data is organized in rows. When the table is stored in memory, we might think that the value of the *Name* column in the first row is adjacent to the values of the *ID* and *Color* columns in the same row. On the other hand, the value in the second row of the *Name* column is slightly farther from the *Name* value in the first row because in between we find *Color* and *Unit Price* in the first row, and the value of the *ID* column in the second row. As an example, the following code is a schematic representation of the physical memory layout of a row store:

```
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
```

Imagine a developer needs to compute the sum of *Unit Price*: The engine must scan the entire memory area, reading many irrelevant values in the process. Imagine scanning the memory of the database sequentially: To read the first value of *Unit Price*, the engine needs to read (and skip) the first row of *ID*, *Name,* and *Color*. Only then does it find an interesting value. The same process is repeated for all the rows. Following this technique, the engine needs to read and ignore many columns to find the relevant values to sum.

Reading and ignoring values take time. In fact, if we asked someone to compute the sum of *Unit Price*, they would not follow that algorithm. Instead, as human beings, they would probably scan the first row in Figure 17-2 searching for the position of *Unit Price*, and then move their eyes down, reading the values one at a time and mentally accumulating them to produce the sum. The reason for this very natural behavior is that we save time by reading vertically instead of row-by-row.

A columnar database organizes data to optimize vertical scanning. To obtain this result, it needs a way to make the different values of a column adjacent to one another. In Figure 17-3 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 17-3 The *Product* table organized column-by-column.

When stored in a columnar database, each column has its own data structure; it is physically separated from the others. Thus, the different values of *Unit Price* are adjacent to one another and distant from *Color*, *Name*, and *ID*. The following code is a schematic representation of the physical memory layout of a column store:

```
ID,1,2,3,4,5,6,7,8,9
Name,Camcorder,Camera,Smartphone,Console,TV,CD,Touch screen,PDA,Keyboard
Color,Red,Red,White,Black,Blue,Red,Blue,Black,Black
Unit Price,112.25,97.50,100.00,112.25,1240.85,39.99,45.12,120.25,120.50
```

With this data structure, computing the sum of *Unit Price* is much easier because the engine immediately goes to the structure containing *Unit Price*. There, it finds all the values needed to perform the computation next to each other. In other words, it does not have to read and ignore other column values: In a single scan, it obtains exclusively the useful numbers, and it can quickly aggregate them.

In our next scenario, instead of summing *Unit Price,* we compute the sum of *Unit Price* just for the Red products. You are encouraged to give this a try before reading on, in order to better understand the algorithm.

This is not so easy anymore; indeed, it is no longer possible to obtain the desired number by simply scanning the *Unit Price* column. What developers would typically do is scan the *Color* column, and whenever it is Red, retrieve the corresponding value in *Unit Price*. At the end, all the values would be summed up to compute the result.

Though very intuitive, this algorithm requires a constant move of the eyes from one column to the other in Figure 17-3, possibly using a finger as a guide to save the last scanned position of *Color*. It is not an optimized way of computing the value. The reason is that the engine needs to constantly jump from one memory area to another, resulting in poor performance. A better way—which only computers use—is to first scan the *Color* column, find the positions where the color is Red, and then scan the *Unit Price* column, summing only the values in the positions identified in the previous step.

This last algorithm is much better because it performs one scan of the first column and one scan of the second column, always accessing memory locations that are adjacent to one another—other than the jump between the scan of the first and second column. Sequential reading of memory is much faster than random access.

For a more complex expression, such as the sum of all products that are either Blue or Black with a price higher than US$50, things are even worse. This time, there is no possibility of scanning the column one at a time because the condition depends on way too many columns. As usual, trying on paper helps better understand the problem.

The simplest algorithm producing the desired result is to scan the table not on a column basis, but on a row basis instead. We naturally tend to scan the table row-by-row, though 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; indeed, it requires a lot of random reads of memory, leading to poorer performance than if computed doing a sequential scan.

As discussed, a columnar storage presents both pros and cons. Columnar databases provide very quick access to a single column; but as soon as one needs a calculation involving many columns, they need to spend some time—after having read the column content—to reorganize the information so that the final expression can be computed. Even though this example was very simple, it helps highlight the most important characteristics of column stores:

- Single-column access is very fast: It sequentially reads a single block of memory and then computes whatever aggregation is needed 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 a temporary area.
- The more columns are needed to compute an expression, the harder it becomes to produce a result. At a certain point it becomes 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 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 we will see in the next sections, columnar databases have more options to reduce the amount of time spent scanning data. The most relevant technique used by VertiPaq 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 described in this section.

Note The actual details of the compression algorithm of VertiPaq are proprietary. Thus, we cannot publish them in a book. Yet what we explain in this chapter is already a good approximation of what takes place in the engine, and we can use it, for all intents and purposes, to describe how the VertiPaq engine stores data.

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

- A smaller model makes better use of the 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 cost of a column. Consider a column containing the price of products, stored as integer values. The column contains many different values and a defined number of bits is required to represent all of them.

In the Figure 17-4 example, the maximum value of *Unit Price* is 216. At least 8 bits are required to store each integer value up to that number. Nevertheless, by using a simple mathematical operation, we can reduce the storage to 5 bits.

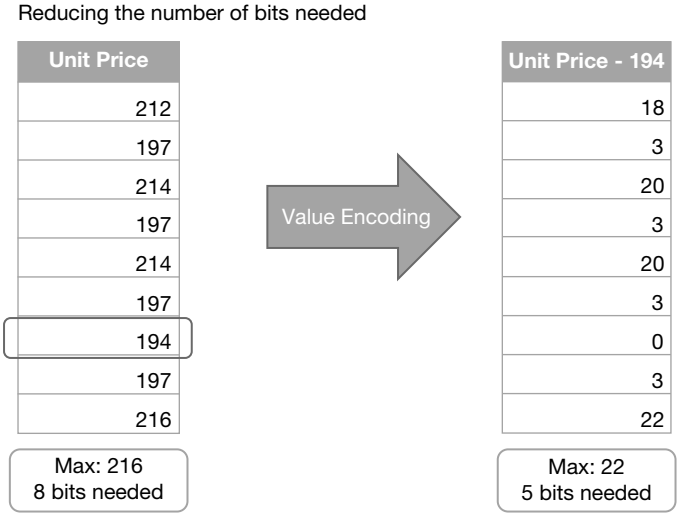


FIGURE 17-4 By using simple mathematical operations, VertiPaq reduces the number of bits needed for a column.

In the example, VertiPaq found out that by subtracting the minimum value (194) from all the values of the column, it could modify the range of the values in the column, reducing it to a range from 0 to 22. Storing numbers up to 22 requires fewer bits than storing numbers up to 216. While 3 bits might seem like an insignificant savings, when we multiply this by a few billion rows, it is easy to see that the difference can be important.

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. This reduces its memory footprint. Obviously, when using the column, it must reapply the transformation in the opposite direction to obtain the original value. Depending on the transformation, this can happen before or after aggregating the values. Again, this increases the CPU usage and reduces the number of reads, which is a very good option.

Value encoding only takes place for integer columns because it cannot be applied on strings or floating-point values. Be mindful that VertiPaq stores the *Currency* data type of DAX (also called Fixed Decimal Number) as an integer value. Therefore, currencies can be value-encoded too, whereas floating point numbers cannot.

Understanding hash encoding

Hash encoding (also known as dictionary encoding) is another technique used by VertiPaq to reduce the number of bits required to store a column. Hash encoding builds a dictionary of the distinct values of a column and then replaces the column values with indexes to the dictionary. In Figure 17-5 you can see the storage of the *Color* column, which uses strings and cannot be value-encoded.

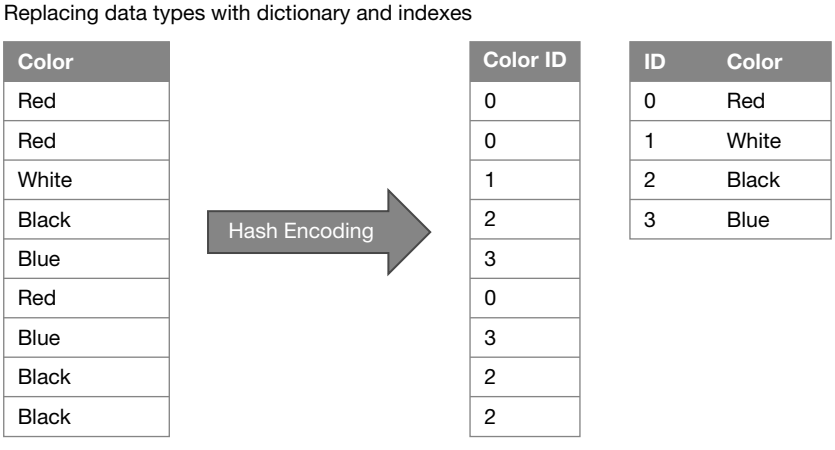


FIGURE 17-5 Hash encoding consists of building a dictionary and replacing values with indexes.

When VertiPaq encodes a column with hash encoding, it

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

There are some advantages in using hash encoding:

- All columns only contain integer values; this makes it simpler to optimize the internal code of the engine. Moreover, it also means that VertiPaq is data type 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, 2 bits are enough because there are only four different values.

These two aspects are of paramount importance for VertiPaq. It does not matter whether a column uses a string, a 64-bit integer, or a floating point to represent a value. All these data types can be hash encoded, providing the same performance 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 original column itself.

The primary factor to determine the column size is not the data type. Instead, 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 this important is always a good thing: Of all the various aspects 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. Consequently, 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 the engine needs to aggregate its values in a DAX expression.

Understanding Run Length Encoding (RLE)

Hash encoding and value encoding are two very good 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 storing in which quarter the sales took place, stored in the *Sales* table. This column might contain the string "Q1" repeated many times in contiguous rows, for all the sales in the same quarter. In such a case, VertiPaq avoids storing values that are repeated. It replaces them with a slightly more complex structure that contains the value only once, with the number of contiguous rows having the same value. This is shown in Figure 17-6.

RLE's efficiency strongly depends on the repetition pattern of the column. Some columns have the same value repeated for many rows, resulting in a great compression ratio. Other columns with quickly changing values produce a lower compression ratio. Data sorting is extremely important to improve the compression ratio of RLE. Therefore, finding an optimal sort order is an important step of the data refresh performed by VertiPaq.

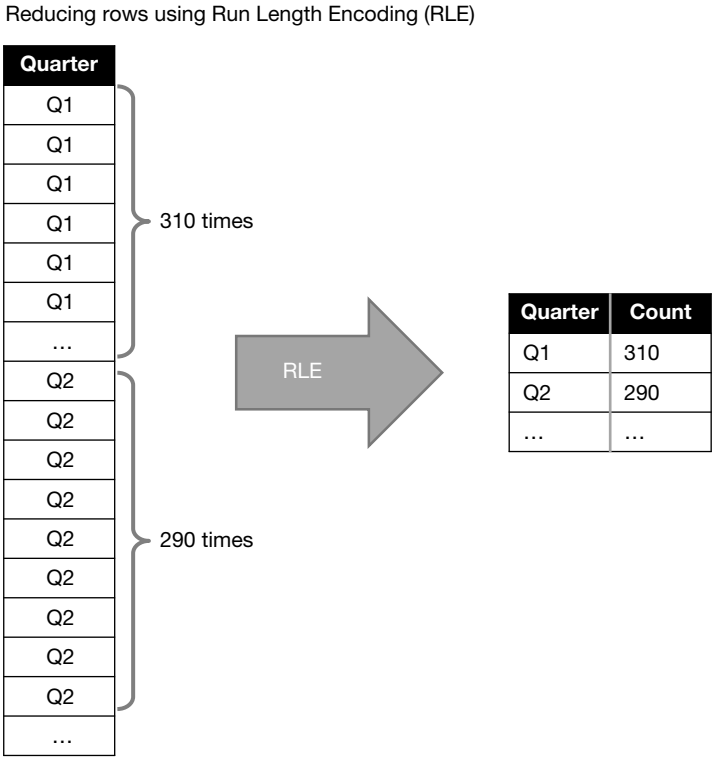


FIGURE 17-6 RLE replaces values that are repeated with the number of contiguous rows with the same value.

Finally, there could be columns in which the content changes so often that if VertiPaq tried to compress them using RLE, the compressed columns would end up using more space than the original columns. A great example of this is 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 cases like this, VertiPaq skips the RLE compression and stores the column as-is. Thus, the VertiPaq storage of a column never exceeds the original column size. Worst-case scenario, both would be the same size.

In the example, we have shown RLE working on a *Quarter* column containing strings. RLE can also process the already hash-encoded version of a column. Each column can have both RLE and either hash or value encoding. Therefore, the VertiPaq storage for a column compressed with hash encoding consists of two distinct entities: the dictionary and the data rows. The latter is the RLE-encoded result of the hash-encoded version of the original column, as shown in Figure 17-7.

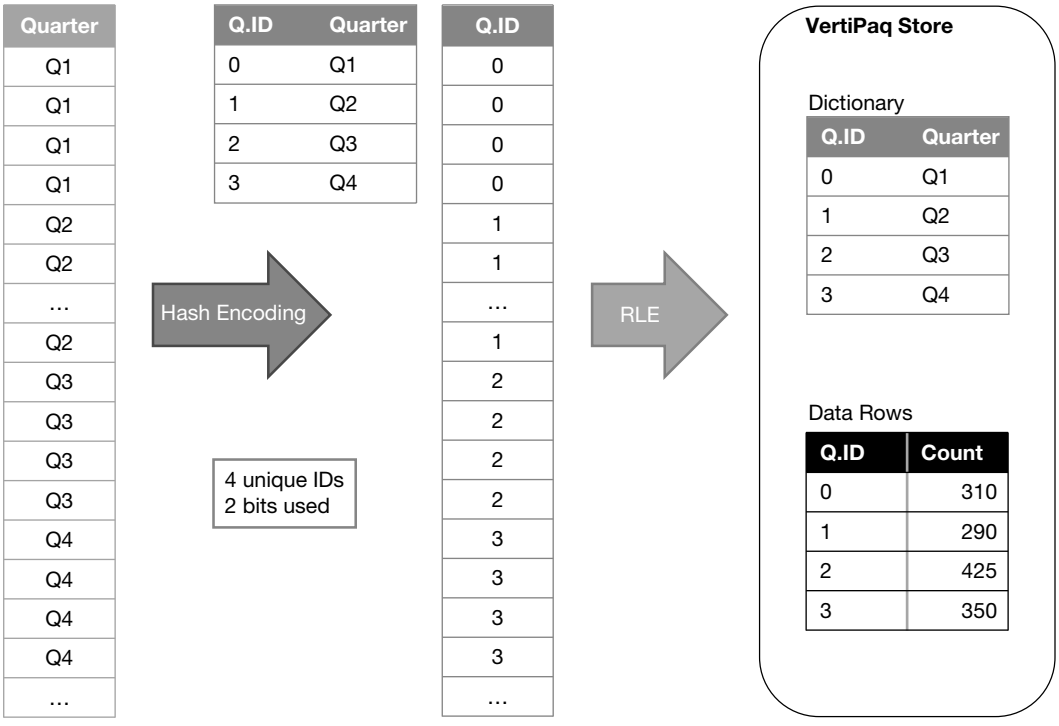


FIGURE 17-7 RLE is applied to the dictionary-encoded version of a column.

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

The factors influencing the compression ratio of a Tabular model are, in order of importance:

1. The cardinality of the column, which defines the number of bits used to store a value.
2. The number of repetitions, that is, the distribution of data in a column. A column with many repeated values is compressed more than a column with very frequently changing values.
3. The number of rows in the table.
4. The data type of the column, which only affects the dictionary size.

Given all these considerations, it is nearly impossible to predict the compression ratio of a table. Moreover, while a developer has full control over certain aspects of a table—they can limit the number of rows and change the data types—these are the least important aspects. Yet as you learn in the next chapter, one can work on cardinality and repetitions too. This improves the compression and performance of a model.

Finally, it is worth noting that reducing the cardinality of a column also increases the chances of repetitions. For example, if a time column is stored at the second granularity, then the column contains up

to 86,400 distinct values. If, on the other hand, the developer stores the same time column at the hour granularity, then not only have they reduced the cardinality, but they also introduced repeating values. Indeed, 3,600 seconds convert to one same hour. All this results in a much better compression ratio. On the other hand, changing the data type from *DateTime* to *Integer* or even *String* offers a negligible impact on column size.

Understanding re-encoding

SSAS must decide which algorithm to use to encode each column. More specifically, it needs to decide whether to use value or dictionary encoding. In order to make an educated decision, it reads a row sample during the first scan of the source, and it chooses a compression algorithm depending on the values found.

If the data type of the column is not *Integer*, then the choice is straightforward: SSAS 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 fall within a defined range of values, then value encoding is the way to go.
- If the numbers fall within 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 sometimes makes the wrong decision and finds this out only very late during processing. For example, SSAS might read a few million rows where the values are in the 100–201 range, so 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 initial choice was wrong because the number of bits needed to store such a large number is huge. What should SSAS do then? Instead of continuing with the wrong choice, SSAS can decide to re-encode the column. This means that the entire column is re-encoded using dictionary encoding. This process might take a long time because SSAS needs to reprocess the whole column.

For very large datasets where processing time is important, a best practice is the following: the data distribution in the first set of rows read by SSAS should be of such quality that all types of values are represented. This in turn reduces re-encoding to a minimum. Developers do so by providing a quality sample in the first partition processed or by providing an encoding hint parameter to the column.

Note The *Encoding Hint* property was introduced in Analysis Services 2017, and it is not available in all products.

Finding the best sort order

As we said earlier, RLE's efficiency strongly depends on the sort order of the table. All the columns of the same table are sorted the same way to keep integrity of the data at the table level. In large tables it is important to determine the best sorting of data to improve the efficiency of RLE and to reduce the memory footprint of the model.

When SSAS reads a 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. At printing time, the default is currently 10 seconds per million rows. One can modify its value in the *ProcessingTimeboxSecPerMRow* entry in the configuration file of the SSAS service. Power BI and Power Pivot do not provide access to this value.

Note SSAS searches for the best sort order in the data, using a heuristic algorithm that certainly also considers the physical order of the rows it receives. For this reason, although one cannot force the sort order used by VertiPaq for RLE, it is possible to provide the engine with data sorted arbitrarily. The VertiPaq engine includes this sort order in the options to consider.

To attain maximum compression, one can set the value of *ProcessingTimeboxSecPerMRow* 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 vary. On the other hand, processing will take much longer because the engine is being instructed to try all the possible sort orders before making a choice.

Generally, developers should put the columns with the least number of unique values first in the sort order because these columns are likely to generate many repeating values. Still, keep in mind that finding the best sort order is a very complex task. It only makes sense to spend time on this when the data model is really large (in the order of a few billion rows). Otherwise, the benefit obtained from these extreme optimizations is limited.

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

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

Consider creating a calculated column that results in a *Boolean* value. There being only two values, the calculated column 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. Indeed, doing this, the table shows all the *True*

values first and only later the *False* values. Being a calculated column, the sort order is already defined by other columns; it might be the case that with the defined sort order, the calculated column frequently changes its value. In that case, the column ends up with less-than-optimal compression.

Whenever there is a chance to compute a column in DAX or in the data source (including Power Query), keep in mind that computing it in the data source results in slightly better compression. Many other factors may drive the choice of DAX instead of Power Query or 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 said 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 were in Power Query or SQL. This is something to consider when looking for the optimal compression.

Note A calculated table has the same compression as a regular table, without the side effects described for calculated columns. However, creating a calculated table can be quite expensive. Indeed, a calculated table requires enough memory to keep a copy of the entire uncompressed table in memory before it is compressed. Carefully think before creating a large calculated table because of the memory pressure generated at refresh time.

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.

There are two types of hierarchies: attribute hierarchies and user hierarchies. Hierarchies are data structures used primarily to improve performance of MDX queries and also to improve certain search operations in DAX. Because the concept of hierarchy is not present in the DAX language, hierarchies are not relevant to the topics of this book.

Relationships, on the other hand, play an important role in the VertiPaq engine; it is important to understand how they work for extreme optimizations. We will describe the role of relationships in a query in following chapters. Here, we are only interested in defining what relationships are, in terms of VertiPaq storage and behavior.

A relationship is a data structure that maps IDs from one table to row numbers in another table. For example, consider the columns *ProductKey* in *Sales* and *ProductKey* in *Product*. These two columns are used to build the relationship between the two tables. *Product[ProductKey]* is a primary key. Because it is a primary key, the engine used value encoding and no compression at all. Indeed, RLE could not reduce the size of a column in the absence of duplicated values. On the other hand, *Sales[ProductKey]* is likely to have been dictionary-encoded and compressed. This is because it probably contains many repetitions. Therefore, despite the columns having the same name and data type, their internal data structures are completely different.

Moreover, because they are part of a relationship, VertiPaq knows that queries are likely to use the columns very often placing a filter on *Product* and also expecting to filter *Sales*. VertiPaq would be very slow if—every time it needs to move a filter from *Product* to *Sales*—it had to perform the following: retrieve values from *Product[ProductKey]*, search them in the dictionary of *Sales[ProductKey]*, and finally retrieve the IDs of *Sales[ProductKey]* to place the filter.

Therefore, 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. Figure 17-8 shows how the relationship between *Sales* and *Product* is stored in VertiPaq.

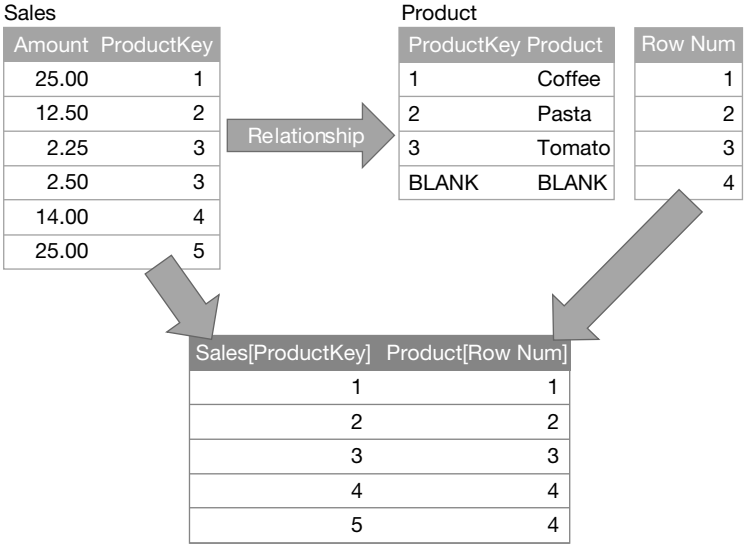


FIGURE 17-8 The figure shows the relationship between *Sales* and *Product*.

Even though the structure does not seem to be very intuitive, later in this chapter we describe how VertiPaq uses relationships and why relationships have this very specific structure. It would come naturally that it is a complex structure optimized for performance.

Understanding segmentation and partitioning

Compressing a table of several billion rows in one single step would be extremely memory-intensive and time-consuming. Therefore, the table is not processed as a single unit. Instead, during processing, SSAS splits the table into segments that contain 8 million rows each by default. When a segment is completely read, the engine starts to compress the segment while reading the next segment in the meantime.

It is possible to configure the segment size in SSAS using the *DefaultSegmentRowCount* entry in the configuration file of the service (or in the server properties in Management Studio). In Power BI Desktop and Power Pivot, the segment size has a set value of 1 million rows, and it cannot be changed.

Segmentation is important for several reasons, including query parallelisms and compression efficiency. 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. We 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 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 takes place at the segment level. Thus, if a column has 1,000 distinct values but only two distinct values are used in a specific segment, then that column will be compressed to a single bit for that segment.

If segments are small, then the parallelism at query time is increased. This is not always a good thing. While it is true that scanning the column is faster because more cores can do that in parallel, 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 on the overall query performance.

During processing, the treatment of the first segment is particular if the table has only one partition. Indeed, the first segment can be larger than *DefaultSegmentRowCount*. VertiPaq reads twice the size of *DefaultSegmentRowCount* and starts to segment a table only if the table contains more rows. This does not apply to a partitioned table. If a table is partitioned, then all the segments are smaller than the default segment row count. Consequently, in SSAS a nonpartitioned table with 10 million rows is stored as a single segment. On the other hand, a table with 20 million rows uses three segments: two containing 8 million rows and one containing 4 million rows. In Power BI Desktop and Power Pivot, VertiPaq uses multiple segments for tables with more than 2 million rows.

Segments cannot exceed the partition size. If the partitioning schema of a model creates partitions of only 1 million rows, then all the segments will be smaller than 8 million rows; namely, they will be same as the partition size. Overpartitioning a table is a common mistake made by novices to optimize performance. What they obtain is the opposite effect: Creating too many small partitions typically lowers performance.

Using Dynamic Management Views

SSAS enables the discovery of all the information about the data model using Dynamic Management Views (DMV). DMVs are extremely useful to explore how a 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.

DMVs can run from inside SQL Server Management Studio. Regardless, we suggest you use DAX Studio; it offers a list of all DMVs in a simpler way without the need to remember them or to reopen this

book looking for the DMV name. However, a more efficient way to use DMVs is with the free VertiPaq Analyzer tool (http://www.sqlbi.com/tools/vertipaq-analyzer/), which displays data from DMVs and organizes them in useful reports, as shown in Figure 17-9.

Row Labels	Cardinality	Table Size	Columns Total Size	Data Size	Dictionary Size	Columns Hierar	Encoding
⊞ExchangeRate	773	63,144	63,064	6,224	45,520	11,320	Many
⊞Geography	674	155,624	141,736	2,640	127,736	11,360	Many
⊟Inventory	8,013,099	108,978,244	108,973,588	76,679,640	188,556	32,105,392	Many
Aging	7		15,780	14,312	1,372	96	HASH
CurrencyKey	1		1,476	64	1,348	64	HASH
Datekey	156		4,240,320	4,229,328	9,696	1,296	HASH
DaysInStock	115		7,126,300	7,122,512	2,828	960	HASH
ETLLoadID	1		1,476	64	1,348	64	HASH
InventoryKey	8,013,099		53,420,840	21,368,304	120	32,052,416	VALUE
LoadDate	1		1,416	64	1,288	64	HASH
MaxDayInStock	60		6,412,616	6,410,504	1,584	528	HASH
MinDayInStock	55		6,412,484	6,410,440	1,564	480	HASH

FIGURE 17-9 VertiPaq Analyzer shows statistics about a data model in an efficient manner.

Although DMVs use an SQL-like syntax, the full SQL syntax is not available. DMVs do not run inside SQL Server. They are only a convenient way to discover the status of SSAS and to gather information about data models.

There are different DMVs, divided into two main categories:

- **SCHEMA views:** These return information about SSAS metadata, such as database names, tables, and individual columns. They are used to gather information about data types, names, and similar data, including statistical information about numbers of rows and unique values stored in columns.
- **DISCOVER views:** They are intended to gather information about the SSAS engine and/or discover statistics information about objects in a database. For example, one 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 describe the details of all the views because doing so would be going off topic. More information is available in Microsoft documentation on the web. Instead, we want to provide a few hints and point out the most useful DMVs related to databases used by DAX. Moreover, while many DMVs report useful information in many columns, in this book we describe the most interesting ones related to the internal structure.

A first useful DMV to discover the memory usage of all the objects in the SSAS instance is *DISCOVER_OBJECT_MEMORY_USAGE*. This DMV returns information about all the objects in all the databases in the SSAS instance. *DISCOVER_OBJECT_MEMORY_USAGE* is not limited to the current database. For example, the following query can be run in DAX Studio or SQL Server Management Studio:

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

Figure 17-10 shows a small excerpt of the result of the previous query. There are many more columns and rows, so analyzing this detailed information can be very time-consuming.

OBJECT_PARENT_PATH	OBJECT_ID	OBJECT_MEMORY_SHRINKABLE	OBJECT_MEMORY_NONSHRINKABLE	OBJECT_VER
GAP\AnalysisServicesWor...	H$DaxBook Sales...	0	0	
MessageManager	French (France)	0	37084	13796
Global	TMPersistenceSQ...	0	368	10477
	Global	0	6357634	
GAP\AnalysisServicesWor...	ID_TO_POS	0	0	

FIGURE 17-10 Partial result of the *DISCOVER_OBJECT_MEMORY_USAGE* DMV.

The output of the DMV is a table containing many rows that are very hard to read. The output structure is a parent/child hierarchy that starts with the instance name and ends with individual column information. Although the raw dataset is nearly impossible to read, one can build a Power Pivot data model on top of this query, implementing the parent/child hierarchy structure and browsing the full memory map of the instance. Kasper De Jonge published a workbook on his blog that does exactly this. It is available at http://www.powerpivotblog.nl/what-is-using-all-that-memory-on-my-analysis-server-instance/.

Other useful DMVs to check the current state of the Tabular engine are *DISCOVER_SESSIONS*, *DISCOVER_CONNECTIONS*, and *DISCOVER_COMMANDS*. These DMVs provide information about active sessions, connections, and executed commands. These views are used by an open source tool called SSAS Activity Monitor, available at https://github.com/RichieBzzzt/SSASActivityMonitor/tree/master/Download, that provides the same information (plus much more) in a more convenient way.

There are also DMVs that analyze the distribution of data in columns and tables, and the memory required for compressed data. These are *TMSCHEMA_COLUMN_STORAGES* and *DISCOVER_STORAGE_TABLE_COLUMNS*. The former is the more recent one; the latter is there for compatibility with older versions of the engine (compatibility level 1103 or lower).

Finally, a very useful DMV to analyze calculation dependency is *DISCOVER_CALC_DEPENDENCY*. This DMV can be used to create a graph of dependencies between calculations in the data model, including calculated columns, calculated tables, and measures. Figure 17-11 shows an excerpt of the result of this DMV.

OBJECT_TYPE	TABLE	OBJECT	EXPRESSION	REFERENCED_OBJECT_TYPE	REFERENCED_TABLE	REFERENCED_OBJECT
MEASURE	Sales	Sales Amo...	SUMX (Sales, Sales[Quantity] * Sales[Net Price])	COLUMN	Sales	Quantity
MEASURE	Sales	Sales Amo...	SUMX (Sales, Sales[Quantity] * Sales[Net Price])	COLUMN	Sales	Net Price
MEASURE	Sales	Total Cost	SUMX (Sales, Sales[Quantity] * Sales[Unit Cost])	TABLE	Sales	Sales
MEASURE	Sales	Total Cost	SUMX (Sales, Sales[Quantity] * Sales[Unit Cost])	COLUMN	Sales	Quantity
MEASURE	Sales	Total Cost	SUMX (Sales, Sales[Quantity] * Sales[Unit Cost])	COLUMN	Sales	Unit Cost

FIGURE 17-11 Partial result of the *DISCOVER_CALC_DEPENDENCY* DMV.

Understanding the use of relationships in VertiPaq

When a DAX query generates requests to the VertiPaq storage engine, the presence of relationships in the data model allows a quicker transfer of the filter context from one table to another. The internal implementation of a relationship in VertiPaq is worth knowing because relationships might affect the performance of a query even though most of the calculation happens in the storage engine.

To understand how relationships work, we start from the analysis of a query that only involves one table, *Sales*:

```
EVALUATE
ROW (
    "Result", CALCULATE (
        COUNTROWS ( Sales ),
        Sales[Quantity] > 1
    )
)

-- Result
-- 20016
```

A developer used to working with tables in relational databases might suppose that the engine iterates the *Sales* table, tests the value of the *Quantity* column for each row of *Sales*, and increments the returned value if the *Quantity* value is greater than 1. In fact, VertiPaq does it better: VertiPaq only scans the *Quantity* column because it already provides the number of rows for the entire table. Therefore, a single column scan is enough to solve the entire query.

If we write a similar query using the column of another table as a filter, then scanning a single column is no longer enough to produce the result. For example, consider the following query that counts the number of rows in *Sales* related to products of the Contoso brand:

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

-- Result
-- 37984
```

This time, we are using two different tables: *Sales* and *Product*. Solving this query requires a bit more effort. Indeed, because the filter is on *Product* and the table to aggregate is *Sales*, it is not possible to scan a single column.

If you are not used to columnar databases, you probably think that, to solve the query, the engine should iterate the *Sales* table, follow the relationship with *Product*, and sum 1 if the product brand is Contoso, 0 otherwise. This would be an algorithm like the following DAX code:

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

```
)

-- Result
-- 37984
```

Although this is a simple algorithm, it contains much more complexity than expected. Indeed, if we carefully think about the columnar nature of VertiPaq, we realize that this query involves three different columns:

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

Iterating over *Sales[ProductKey]*, searching the row number in *Product* scanning *Product[ProductKey]*, and finally gathering the brand in *Product[Brand]* would be extremely expensive. The process requires a lot of random reads to memory, with negative consequences on performance. Therefore, VertiPaq uses a completely different algorithm, optimized for columnar databases.

First, VertiPaq scans the *Product[Brand]* column and retrieves the row numbers of the *Product* table where *Product[Brand]* is Contoso. As shown in Figure 17-12, VertiPaq scans the *Brand* dictionary (1), retrieves the encoding of Contoso, and finally scans the segments (2) searching for the row numbers in the product table where the dictionary ID equals 0 (corresponding to Contoso), returning the indexes to the rows found (3).

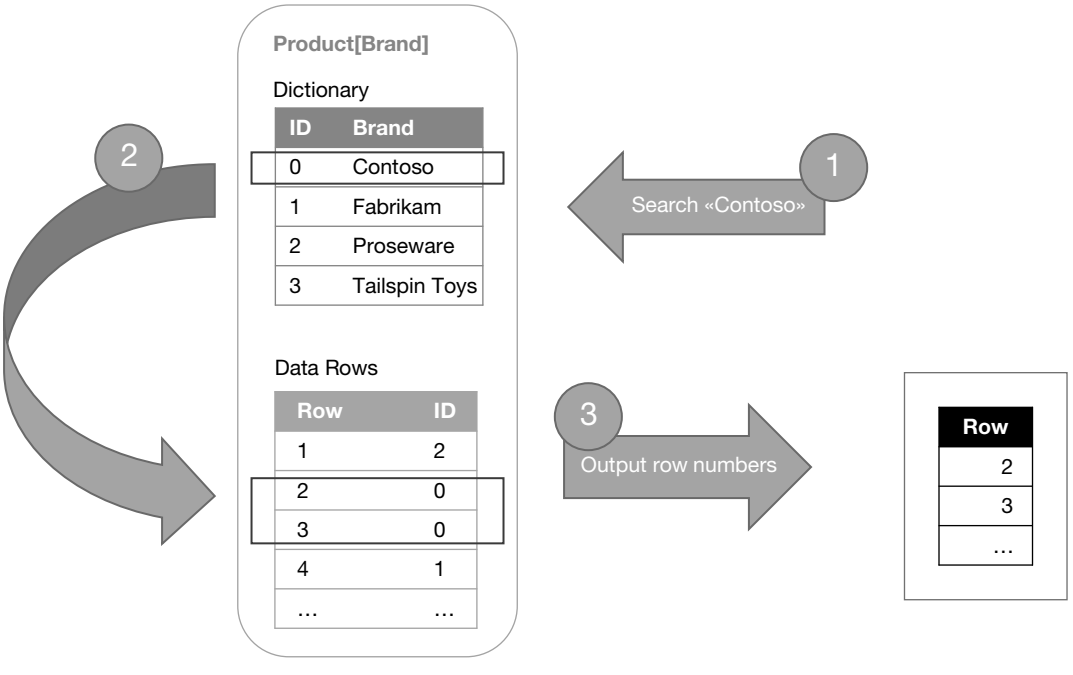


FIGURE 17-12 The output of a brand scan is the list of rows where *Brand* equals Contoso.

At this point, VertiPaq knows which rows in the *Product* table contain the given brand. The relationship between *Product* and *Sales* enables VertiPaq to translate the row numbers of *Product* in internal data IDs for *Sales[ProductKey]*. VertiPaq performs a lookup of the selected row numbers to determine the values of *Sales[ProductKey]* valid for those rows, as shown in Figure 17-13.

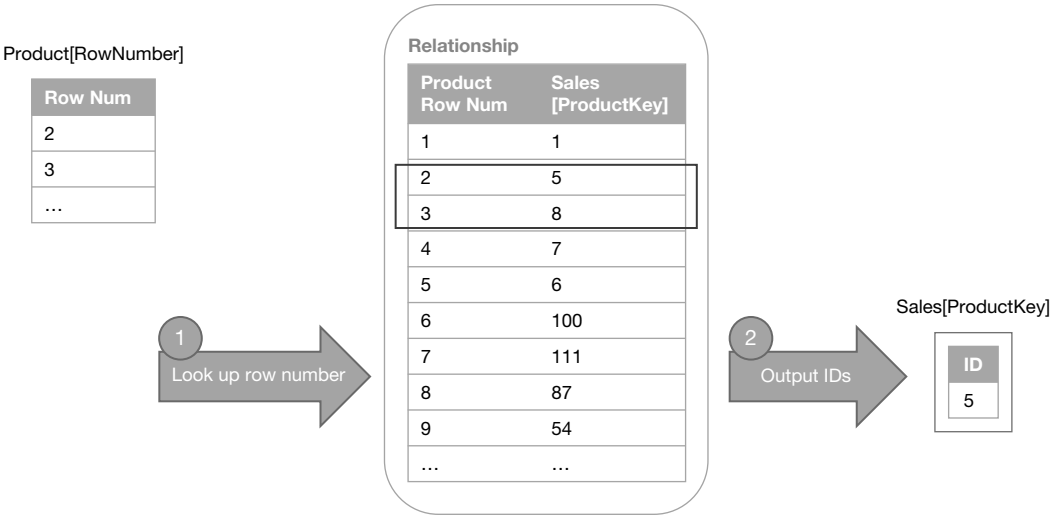


FIGURE 17-13 VertiPaq scans the product keys in the relationship to retrieve the IDs where brand equals Contoso.

The last step is to apply the filter on the *Sales* table. Since VertiPaq already has 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 an additional step transforming row numbers into column values to perform the last step.

The important takeaway is that the cost of a relationship depends on the cardinality of the column that defines the relationship. Even though the previous query filtered only one brand, the cost of the relationship was the number of products for that brand. The lower the cardinality of a relationship, the better. When the cardinality of a relationship is above one million unique values, the end user can experience slower performance. A performance degradation is already measurable when the relationship has 100,000 unique values. VertiPaq aggregations can mitigate the impact of high-cardinality relationships by pre-aggregating data at a different granularity, removing the cost of traversing expensive relationships at query time. We briefly discuss aggregations later in this chapter.

Introducing materialization

Now that we have provided a basic explanation of how VertiPaq stores data in memory, we can describe what *materialization* is. Materialization is a step of the query execution that occurs when using columnar databases. Understanding when and how it happens is of paramount importance.

The basic principle about materialization is that every time the formula engine sends a request to the storage engine, the formula engine receives an uncompressed table that is generated dynamically by the storage engine. This special temporary table is called a *datacache*. A datacache is always the materialization of data that will be consumed by the formula engine, regardless of the storage engine used. Both VertiPaq and DirectQuery generate datacaches.

A large materialization happens when a single storage engine query produces a large datacache. The conditions for a DAX query to produce a large materialization depend on many factors; basically, whenever the storage engine is not able to execute all the operations required by the DAX query, the formula engine will do the work using a copy of the data owned by the storage engine. Be mindful that the formula engine cannot access the raw data directly, whether VertiPaq or DirectQuery. To access the raw data, the formula engine needs to ask the storage engine to retrieve the data and save it in a datacache. The amount and kind of materialization can be very different depending on the storage engine used. In this book, we only describe how to reduce the materialization in VertiPaq. For DirectQuery there could be differences between different data source drivers. Even so, the tools used to measure the materialization produced by the storage engine are the same used for VertiPaq.

The next chapters describe how to measure the materialization produced by a DAX query using specific tools and metrics. In this section, we just introduce the concept of materialization and how it relates to the result of a query. The cardinality of the result of every DAX query defines the optimal materialization. For example, the following query returns a single row, counting the number of rows in a table:

```
EVALUATE
ROW (
   "Result", COUNTROWS ( Sales )
)

-- Result
-- 100231
```

The optimal materialization for the previous query is a datacache with only one row. This means that the entire calculation is performed within the storage engine. The next query returns one row for each year; therefore, the optimal materialization is three rows, one for each year with sales:

```
EVALUATE
SUMMARIZECOLUMNS (
    'Date'[Calendar Year],
    "Sales Amount", [Sales Amount]
)

-- Calendar Year |  Sales Amount
-----------------|---------------
-- CY 2007       | 11,309,946.12
-- CY 2008       |  9,927,582.99
-- CY 2009       |  9,353,814.87
```

Whenever the storage engine produces a single datacache with the same cardinality as the result of the DAX query, that is called a *late materialization*. If the storage engine produces more datacaches and/or the datacache produced has more rows than those displayed in the result, we have an *early*

materialization. With a late materialization the formula engine does not have to aggregate data, whereas with an early materialization the formula engine must perform operations like joining and grouping, which result in slower queries for the end users.

Predicting materialization is not easy without a deep knowledge of the VertiPaq engine. For example, the materialization of the following query is optimal because the entire calculation is executed within the storage engine:

```
EVALUATE
VAR LargeOrders =
    CALCULATETABLE (
        DISTINCT ( Sales[Order Number] ),
        Sales[Quantity] > 1
    )
VAR Result =
    ROW (
        "Orders", COUNTROWS ( LargeOrders )
    )
RETURN
    Result

-- Orders
-- 8388
```

On the other hand, the next query creates a temporary table that corresponds to the number of unique combinations between customers and dates related to sales with a quantity greater than one (for a total of 6,290 combinations):

```
EVALUATE
VAR LargeSalesCustomerDates =
    CALCULATETABLE (
        SUMMARIZE ( Sales, Sales[CustomerKey], Sales[Order Date] ),
        Sales[Quantity] > 1
    )
VAR Result =
    ROW (
        "CustomerDates", COUNTROWS ( LargeSalesCustomerDates )
    )
RETURN
    Result

-- CustomerDates
-- 6290
```

The latter query has a materialization of 6,290 rows, even though there is only one row in the result. The two queries are similar: a table is evaluated and then its rows are counted. The reason why the former has an earlier materialization is because it involves a single column, whereas the calculation requiring the combinations of two columns cannot be solved by the storage engine by just scanning the two columns. In general, any operation involving a single column has higher chances of being solved in the storage engine, but it would be a mistake to believe that involving multiple columns is

always an issue. For example, the following query has an optimal late materialization even though it multiplies two columns from two tables, *Sales* and *Product*:

```
DEFINE
    MEASURE Sales[Sales Amount] =
        SUMX (
            Sales,
            Sales[Quantity] * RELATED ( 'Product'[Unit Price] )
        )
EVALUATE
ROW ( "Sales Amount", [Sales Amount] )

-- Sales Amount
-- 33,690,148.51
```

In complex queries it is nearly impossible to obtain an optimal late materialization. Therefore, the effort for optimizing a query is reducing the materialization, pushing most of the workload to the storage engine, if possible.

Introducing aggregations

A data model can have multiple tables related to the same original raw data. The purpose of this redundancy is to offer alternative ways to the storage engine to retrieve the data faster. The tables used to this purpose are called *aggregations*.

An aggregation is nothing but a pregrouped version of the original table. By pre-aggregating data, one reduces the number of columns (hence, the number of rows) and replaces values with their aggregate.

As an example, consider the *Sales* table in Figure 17-14, which has one row for each date, product, and customer.

Sales

Date	Product	Customer	Quantity	Amount
2018-09-01	AV010	C092	3	29.97
2018-09-01	AV022	C092	1	16.40
2018-09-01	AV010	C054	2	19.98
2018-09-01	FL892	C248	1	190.00
2018-09-01	GT400	C127	1	999.00
2018-09-02	AV010	C115	3	29.97
2018-09-02	FL580	C127	1	790.00
2018-09-02	AV022	C772	2	32.80
2018-09-02	KB723	C614	2	59.98
2018-09-02	FL580	C614	1	790.00
…	…	…	…	…

FIGURE 17-14 The original *Sales* table has a high number of rows.

If a query requires the sum of *Quantity* or *Amount* by *Date*, the storage engine must evaluate and aggregate all the rows with the same *Date*. In VertiPaq this operation is relatively quick, thanks to the compression and the optimized algorithms that scan the memory. DirectQuery is usually much slower than VertiPaq to perform the same operation. Anyway, VertiPaq also requires time to scan billions of rows rather than millions of rows. Therefore, there could be an advantage in creating an alternate—smaller—table to use in place of the original one.

Figure 17-15 shows the content of a *Sales* table aggregated by *Date*. In this case, there is only one row for every date, and the *Quantity* and *Amount* columns store the sum of the values included in the original rows, pre-aggregated by *Date*.

Sales Agg Date

Date	Quantity	Amount
2018-09-01	8	1,255.35
2018-09-02	9	1,702.75
…	…	…

FIGURE 17-15 The *Sales Agg Date* table has one row for every date.

In an aggregated table, every column is either a "group by" or an aggregation of the original table. If a request to the storage engine only needs columns that are present in an aggregation table, then the engine uses the aggregation rather than the original source. The *Sales Agg Date* table shown in Figure 17-15 can be mapped as an aggregation of *Sales* by specifying the role of each column:

- *Date*: GroupBy *Sales[Date]*
- *Quantity*: Sum *Sales[Quantity]*
- *Amount*: Sum *Sales[Amount]*

The aggregation type must be specified for every column that is not a "group by." The aggregation types available are Count, Min, Max, Sum, and count rows of the table. A column in an aggregation table can only map native columns in the original table; it is not possible to specify an aggregation over a calculated column.

Important Aggregations cannot be used to optimize the execution of complex calculations in DAX. The only purpose of aggregations is to reduce the execution time of storage engine queries. Aggregations can be useful for relatively small tables in DirectQuery, whereas aggregations for VertiPaq should be considered only for tables with billions of rows.

A table in a Tabular model can have multiple aggregations with different priorities in case there are multiple aggregations compatible with a specific storage engine request. Moreover, aggregations and original tables can be stored with different storage engines. A common scenario is storing aggregations in VertiPaq to improve the performance of large tables accessed through DirectQuery. Nevertheless, it is also possible to create aggregations in the same storage engine used for the original table.

Note There could be limitations in storage engines available for aggregations and original tables, depending on the version and the license of the product used. This section provides general guidance on the concept of aggregations, which are one of the tools to optimize performance of a DAX query as described in the following chapters.

Aggregations are powerful, but they require a lot of attention to detail. An incorrect definition of aggregations produces incorrect or inconsistent results. It is a responsibility of the data modeler to guarantee that a query executed in an aggregation produces the same result as an equivalent query executed on the original table. Aggregations are an optimization tool and should be used only whenever strictly necessary. The presence of aggregations requires additional work to define and maintain the aggregation tables in the data model. One should therefore use them only after having checked that a performance benefit exists.

Choosing hardware for VertiPaq

Choosing the right hardware is critical for a solution based on a Tabular model using the VertiPaq storage engine. Spending more does not always mean having a better machine. This section describes how to choose the right hardware for a Tabular model.

Since the introduction of Analysis Services 2012, we helped several companies adopt the new Tabular model in their solutions. A very common issue was that when going into production, performance was slower than expected. Worse, sometimes it was slower 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 we will explain, the problem is not the use of a virtual machine in itself. Instead, the problem is more likely the technical specs of the underlying hardware. A very complete and detailed hardware-sizing guide for Analysis Services Tabular is available in the whitepaper titled "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 quick guide to understand the issues affecting many data centers when they host a Tabular solution. Users of Power Pivot or Power BI Desktop on a personal computer can skip the details about Non-Uniform Memory Access (NUMA) support, but all the other considerations are equally true for choosing the right hardware.

Hardware choice as an option

The first question is whether one can choose their 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. One can only influence the number of cores and the amount of RAM that are assigned to the server. Unfortunately, these parameters are not so relevant for performance. If there are limited choices available, one should collect information about the CPU model and clock of the host server as soon as possible. If this information is not accessible, ask for a small virtual machine running on the same host server and run the Task Manager: The Performance tab shows the CPU model and the clock rate. With this

information, one can predict whether the performance will be worse than an average modern laptop. Unfortunately, chances are that many developers will be in that position. If so, then they must sharpen their political skills to convince the right people that running Tabular on that server is a bad idea. If the host server is a good machine, then one still needs to avoid the pitfall of running a virtual machine on different NUMA nodes (more on this later).

Set hardware priorities

If it is possible to influence the hardware selection, this is the order of priorities:

1. **CPU Clock and Model**: the faster, the better.
2. **Memory Speed:** the faster, the better.
3. **Number of Cores:** the higher, the better. Still, a few fast cores are way better than many slow cores.
4. **Memory Size.**

Disk I/O performance is not on the list. Indeed, it is not important at query time although it could have a role in improving the speed of a disaster recovery. There is only one condition (paging) where disk I/O affects performance, and we discuss it later in this section. However, the RAM of the system should be sized so that there will be no paging at all. Our reader should allocate the budget on CPU and memory speed, memory size, and not waste money on disk I/O bandwidth. The following sections include information to consider for such allocation.

CPU model

The most important factors that affect the speed of code running in 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 a benchmark measuring the different performance in queries that stress the formula engine. An example of such a query is the following:

```
DEFINE
VAR t1 =
    SELECTCOLUMNS ( CALENDAR ( 1, 10000 ), "x", [Date] )
VAR t2 =
    SELECTCOLUMNS ( CALENDAR ( 1, 10000 ), "y", [Date] )
VAR c =
    CROSSJOIN ( t1, t2 )
VAR result =
    COUNTROWS ( c )
EVALUATE
    ROW ( "x", result )
```

This query can run in DAX Studio or SQL Server Management Studio connected to any Tabular model; the execution is intentionally slow and does not produce any meaningful result. Using a query of a typical workload for a specific 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 9.5 seconds on an Intel i7-4770K 3.5 GHz, and in 14.4 seconds on an Intel i7-6500U 2.5 GHz. These CPUs run a desktop workstation and a notebook, respectively. Do not assume that a server will be faster. You should always evaluate hardware performance by running the same test with the same version of the engine and looking at the results because they are often surprising.

In general, Intel Xeon processors used on a server are E5 and E7 series, and it is common to find clock speed around 2–2.4 GHz even with a very high number of cores available. You should look for a clock speed of 3 GHz or more. 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.

The reason why CPU and cache are so important for VertiPaq is clarified in Table 17-1, which compares the typical access time of data stored at different distances from the CPU. The column with human metrics represents the same difference using metrics that are easier for humans to understand.

TABLE 17-1 Expanded versions of the tables

Access	Access Time	Human Metrics
1 CPU cycle	0.3 ns	1 s
L1 cache	0.9 ns	3 s
L2 cache	2.8 ns	9 s
L3 cache	12.9 ns	43 s
RAM access	120 ns	6 min
Solid-state disk I/O	50–150 µs s	2–6 days
Rotational disk I/O	1–10 ms	1–12 months

As shown here, the fastest storage in a PC is not the RAM; it is the core cache. It should be clear that a large L2 cache is important, and the CPU speed plays a primary role in determining performance. The same table also clarifies why keeping data in RAM is so much better than accessing data in other, slower storage devices.

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, performance counters report CPU usage instead of I/O waits. Unfortunately, there are no performance counters that monitor the time spent waiting for the RAM access. In Tabular, this amount of time can be relevant, and it is hard to measure.

In general, you should use RAM that has at least 1,833 MHz; however, if the hardware platform permits, you should select faster RAM—2,133 MHz or more.

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 BI and Power Pivot). A CPU with eight cores will not use all of them in a single query unless a table has at least 64 million rows, or 8 million rows in Power BI and Power Pivot.

For these reasons, scalability over multiple cores is effective only for very large tables. Raising the number of cores improves performance for a single query only when it hits a large table, 200 million 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 as 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 available on a single socket, getting the highest clock rate possible. Having two or more sockets on the same server is not good, even though Analysis Services Tabular recognizes the NUMA architecture. NUMA requires a more expensive inter-socket 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

The entire volume of data managed by VertiPaq must be stored in memory. Additional RAM is required to execute process operations—unless there is a separate process server—and to execute queries. Optimized queries usually do not have a high request for 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 a query will end by returning a result, but increasing available RAM does not produce any performance improvement. Cache used by Tabular does not increase just because there is more RAM available. However, a condition of low available memory might negatively affect query performance if the server starts paging data. Developers should have enough memory to store all the data of their database and to avoid materialization during query execution. More memory than this is a waste of resources.

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 under low memory conditions. 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 there is systematic paging caused by low memory availability.

Best practices in hardware selection

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 very new. 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. One will need time and numbers, doing a proper benchmark, to convince a company that a "standard server" could be the weak point of their entire BI solution.

Conclusions

In this first chapter about optimization we described the internal architecture of a Tabular engine, and we provided the basic information about how data is stored in VertiPaq. As you will see in the following chapters, this knowledge is of paramount importance to optimize your code.

These are the main topics you learned in the chapter:

- There are two engines inside a Tabular server: the formula engine and storage engine.
- The formula engine is the top-level query engine. It is very powerful but rather limited in terms of speed because it is single-threaded.
- There are two storage engines: VertiPaq and DirectQuery.
- VertiPaq is an in-memory columnar database. It stores information on a column-by-column basis, providing very quick access to single columns. Using multiple columns in a single DAX formula might require materialization.
- VertiPaq compresses columns to reduce the memory scan time. Optimizing a model means optimizing the compression by reducing the cardinality of a column as much as possible.
- Both VertiPaq and DirectQuery storage engines can coexist in the same model; this is called a composite model. A single query can use only VertiPaq, only DirectQuery, or both, depending on the storage model of the tables involved in the query.

Now that we have provided the basic knowledge about the internals of the engine, in the next chapter we start learning a few techniques to optimize VertiPaq storage to reduce both the size of a data model and its execution time.

Index

Numbers

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

Marco Russo and **Alberto Ferrari** are the founders of sqlbi.com, where they regularly publish articles about Microsoft Power BI, Power Pivot, 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. Their courses, both in-person and online, are the major source of learning for many DAX enthusiasts.

They both provide consultancy and mentoring on business intelligence (BI) using Microsoft technologies. They have written several books and papers about Power BI, DAX, and Analysis Services. They constantly help the community of DAX users providing content for the websites daxpatterns.com, daxformatter.com, and dax.guide.

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