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

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.

Copyright © 2020 by Alberto Ferrari and Marco Russo

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 1

What is DAX?

DAX, which stands for Data Analysis eXpressions, is the programming language of Microsoft Power BI, Microsoft Analysis Services, and Microsoft Power Pivot for Excel. It was created in 2010, with the first release of PowerPivot for Microsoft Excel 2010. In 2010, PowerPivot was spelled without the space. The space was introduced in the Power Pivot name in 2013. Since then, DAX has gained popularity, both within the Excel community, which uses DAX to create Power Pivot data models in Excel, and within the Business Intelligence (BI) community, which uses DAX to build models with Power BI and Analysis Services. DAX is present in many different tools, all sharing the same internal engine named *Tabular*. For this reason, we often refer to *Tabular models*, including all these different tools in a single word.

DAX is a simple language. That said, DAX is different from most programming languages, so becoming acquainted with some of its new concepts might take some time. In our experience, having taught DAX to thousands of people, learning the basics of DAX is straightforward: you will be able to start using it in a matter of hours. When it comes to understanding advanced concepts such as evaluation contexts, iterations, and context transitions, everything will likely seem complex. Do not give up! Be patient. When your brain starts to digest these concepts, you will discover that DAX is, indeed, an easy language. It just takes some getting used to.

This first chapter begins with a recap of what a data model is in terms of tables and relationships. We recommend readers of all experience levels read this section to gain familiarity with the terms used throughout the book when referring to tables, models, and different kinds of relationships.

In the following sections, we offer advice to readers who have experience with programming languages such as Microsoft Excel, SQL, and MDX. Each section is focused on a certain language, for readers curious to briefly compare DAX to it. Focus on languages you know if a comparison is helpful to you; then read the final section, "DAX for Power BI users," and move on to the next chapter where our journey into the DAX language truly begins.

Understanding the data model

DAX is specifically designed to compute business formulas over a data model. The readers might already know what a data model is. If not, we start with a description of data models and relationships to create a foundation on which to build their DAX knowledge.

A data model is a set of tables, linked by relationships.

We all know what a table is: a set of rows containing data, with each row divided into columns. Each column has a data type and contains a single piece of information. We usually refer to a row in a table as a record. Tables are a convenient way to organize data. A table is a data model in itself although in its simplest form. Thus, when we write names and numbers in an Excel workbook, we are creating a data model.

If a data model contains many tables, it is likely that they are linked through relationships. A relationship is a link between two tables. When two tables are tied with a relationship, we say that they are related. Graphically, a relationship is represented by a line connecting the two tables. Figure 1-1 shows an example of a data model.

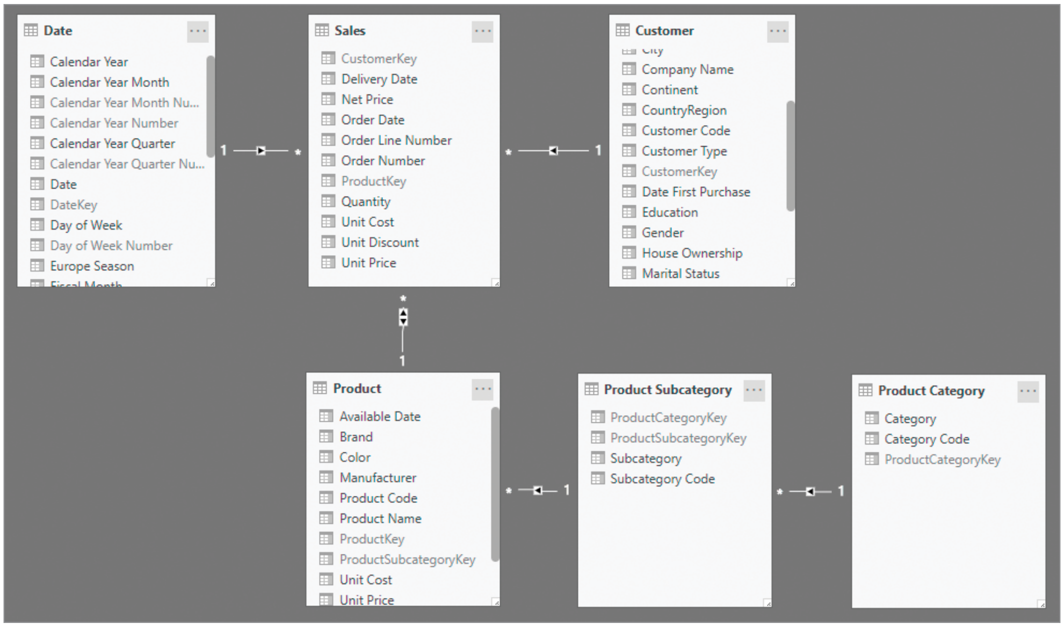


FIGURE 1-1 This data model is made up of six tables.

Following are a few important aspects of relationships:

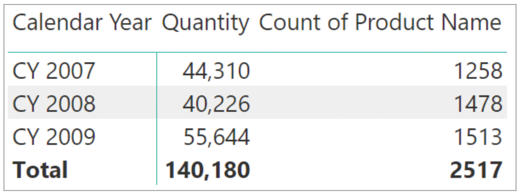
- Two tables in a relationship do not have the same role. They are called the *one-side* and the *many-side* of the relationship, represented respectively with a 1 and with a *. In Figure 1-1, focus on the relationship between Product and Product Subcategory. A single subcategory contains many products, whereas a single product has only one subcategory. Therefore, Product Subcategory is the one-side of the relationship, having one subcategory, while Product is the many-side having many products.
- Special kinds of relationships are 1:1 and weak relationships. In 1:1 relationships, both tables are the one-side, whereas in weak relationships, both tables can be the many-side. These special kinds of relationships are uncommon; we discuss them in detail in Chapter 15, "Advanced relationships."

- The columns used to create the relationship, which usually have the same name in both tables, are called the keys of the relationship. On the one-side of the relationship, the column needs to have a unique value for each row, and it cannot contain blanks. On the many-side, the same value can be repeated in many different rows, and it often is. When a column has a unique value for each row, it is called a key for the table.
- Relationships can form a chain. Each product has a subcategory, and each subcategory has a category. Thus, each product has a category. To retrieve the category of a product, one must traverse a chain of two relationships. Figure 1-1 includes an example of a chain made up of three relationships, starting with Sales and continuing on to Product Category.
- In each relationship, one or two small arrows can determine the *cross filter direction*. Figure 1-1 shows two arrows in the relationship between Sales and Product, whereas all other relationships have a single arrow. The arrow indicates the direction of the automatic filtering of the relationship (*cross filter*). Because determining the correct direction of filters is one of the most important skills to learn, we discuss this topic in more detail in later chapters. We usually discourage the use of bidirectional filters, as described in Chapter 15. They are present in this model for educational purposes only.

Understanding the direction of a relationship

Each relationship can have a unidirectional or bidirectional cross filter. Filtering always happens from the one-side of the relationship to the many-side. If the cross filter is bidirectional—that is, if it has two arrows on it—the filtering also happens from the many-side to the one-side.

An example might help in understanding this behavior. If a report is based on the data model shown in Figure 1-1, with the years on the rows and *Quantity* and *Count of Product Name* in the values area, it produces the result shown in Figure 1-2.



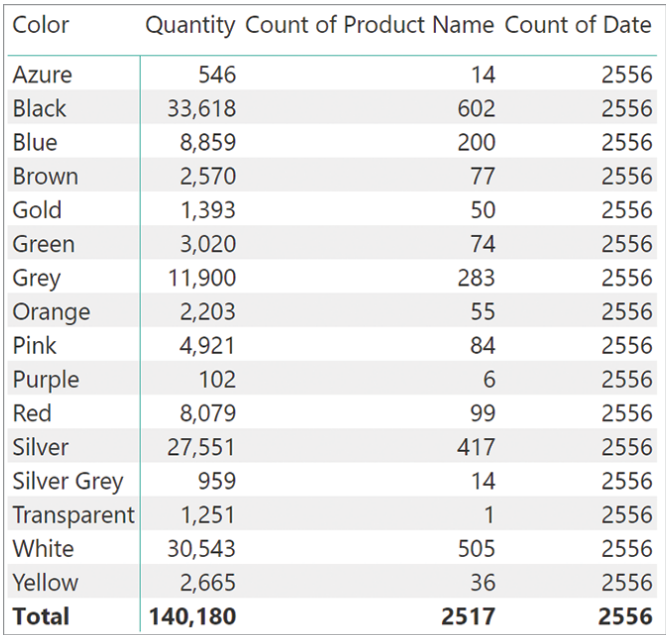
Calendar Year	Quantity	Count of Product Name
CY 2007	44,310	1258
CY 2008	40,226	1478
CY 2009	55,644	1513
Total	**140,180**	**2517**

FIGURE 1-2 This report shows the effect of filtering across multiple tables.

Calendar Year is a column that belongs to the *Date* table. Because *Date* is on the one-side of the relationship with *Sales,* the engine filters *Sales* based on the year. This is why the quantity shown is filtered by year.

With *Products,* the scenario is slightly different. The filtering happens because the relationship between the *Sales* and *Product* tables is bidirectional. When we put the count of product names in the report, we obtain the number of products sold in each year because the filter on the year propagates to *Product* through *Sales*. If the relationship between *Sales* and *Product* were unidirectional, the result would be different, as we explain in the following sections.

If we modify the report by putting *Color* on the rows and adding *Count of Date* in the values area, the result is different, as shown in Figure 1-3.



Color	Quantity	Count of Product Name	Count of Date
Azure	546	14	2556
Black	33,618	602	2556
Blue	8,859	200	2556
Brown	2,570	77	2556
Gold	1,393	50	2556
Green	3,020	74	2556
Grey	11,900	283	2556
Orange	2,203	55	2556
Pink	4,921	84	2556
Purple	102	6	2556
Red	8,079	99	2556
Silver	27,551	417	2556
Silver Grey	959	14	2556
Transparent	1,251	1	2556
White	30,543	505	2556
Yellow	2,665	36	2556
Total	**140,180**	**2517**	**2556**

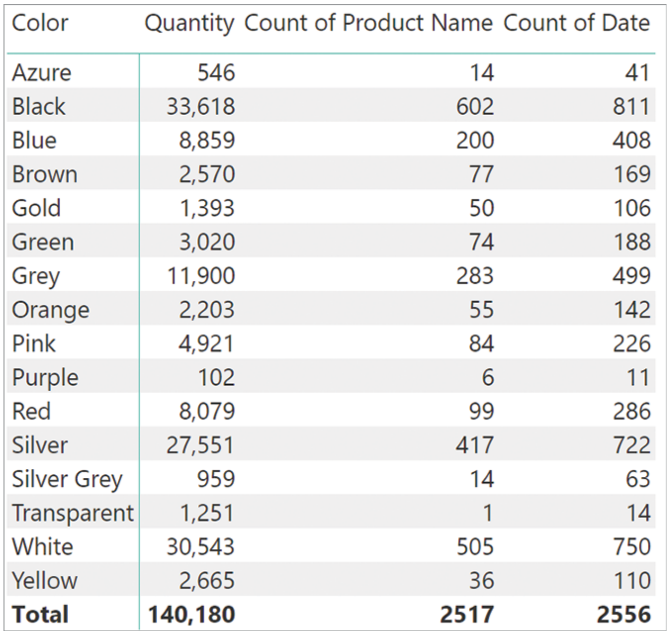
FIGURE 1-3 This report shows that if bidirectional filtering is not active, tables are not filtered.

The filter on the rows is the *Color* column in the *Product* table. Because *Product* is on the one-side of the relationship with *Sales, Quantity* is filtered correctly. *Count of Product Name* is filtered because it is computing values from the table that is on the rows, that is *Product*. The unexpected number is *Count of Date*. Indeed, it always shows the same value for all the rows—that is, the total number of rows in the *Date* table.

The filter coming from the *Color* column does not propagate to *Date* because the relationship between *Date* and *Sales* is unidirectional. Thus, although *Sales* has an active filter on it, the filter cannot propagate to *Date* because the type of relationship prevents it.

If we change the relationship between *Date* and *Sales* to enable bidirectional cross-filtering, the result is as shown in Figure 1-4.

The numbers now reflect the number of days when at least one product of the given color was sold. At first sight, it might look as if all the relationships should be defined as bidirectional, so as to let the filter propagate in any direction and always return results that make sense. As you will learn later in this book, designing a data model this way is almost never appropriate. In fact, depending on the scenario you are working with, you will choose the correct propagation of relationships. If you follow our suggestions, you will avoid bidirectional filtering as much as you can.



Color	Quantity	Count of Product Name	Count of Date
Azure	546	14	41
Black	33,618	602	811
Blue	8,859	200	408
Brown	2,570	77	169
Gold	1,393	50	106
Green	3,020	74	188
Grey	11,900	283	499
Orange	2,203	55	142
Pink	4,921	84	226
Purple	102	6	11
Red	8,079	99	286
Silver	27,551	417	722
Silver Grey	959	14	63
Transparent	1,251	1	14
White	30,543	505	750
Yellow	2,665	36	110
Total	**140,180**	**2517**	**2556**

FIGURE 1-4 If we enable bidirectional filtering, the *Date* table is filtered using the *Color* column.

DAX for Excel users

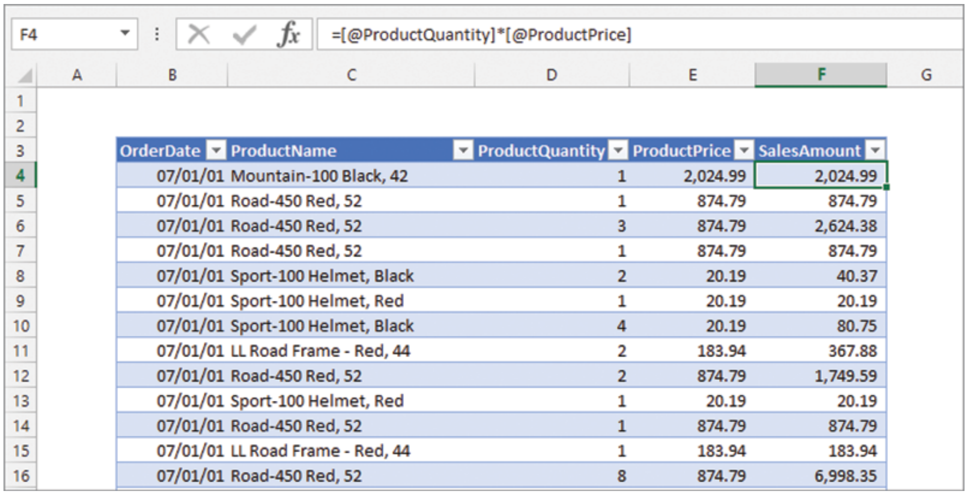
Chances are you already know the Excel formula language that DAX somewhat resembles. After all, the roots of DAX are in Power Pivot for Excel, and the development team tried to keep the two languages similar. This similarity makes the transition to this new language easier. However, there are some important differences.

Cells versus tables

Excel performs calculations over cells. A cell is referenced using its coordinates. Thus, we can write formulas as follows:

```
= (A1 * 1.25) - B2
```

In DAX, the concept of a cell and its coordinates does not exist. DAX works on tables and columns, not cells. As a consequence, DAX expressions refer to tables and columns, and this means writing code differently. The concepts of tables and columns are not new in Excel. In fact, if we define an Excel range as a table by using the *Format as Table* function, we can write formulas in Excel that reference tables and columns. In Figure 1-5, the *SalesAmount* column evaluates an expression that references columns in the same table instead of cells in the workbook.



F4 =[@ProductQuantity]*[@ProductPrice]

OrderDate	ProductName	ProductQuantity	ProductPrice	SalesAmount
07/01/01	Mountain-100 Black, 42	1	2,024.99	2,024.99
07/01/01	Road-450 Red, 52	1	874.79	874.79
07/01/01	Road-450 Red, 52	3	874.79	2,624.38
07/01/01	Road-450 Red, 52	1	874.79	874.79
07/01/01	Sport-100 Helmet, Black	2	20.19	40.37
07/01/01	Sport-100 Helmet, Red	1	20.19	20.19
07/01/01	Sport-100 Helmet, Black	4	20.19	80.75
07/01/01	LL Road Frame - Red, 44	2	183.94	367.88
07/01/01	Road-450 Red, 52	2	874.79	1,749.59
07/01/01	Sport-100 Helmet, Red	1	20.19	20.19
07/01/01	Road-450 Red, 52	1	874.79	874.79
07/01/01	LL Road Frame - Red, 44	1	183.94	183.94
07/01/01	Road-450 Red, 52	8	874.79	6,998.35

FIGURE 1-5 Excel can reference column names in tables.

Using Excel, we refer to columns in a table using the *[@ColumnName]* format. *ColumnName* is the name of the column to use, and the @ symbol means "take the value for the current row." Although the syntax is not intuitive, normally we do not write these expressions. They appear when we click a cell, and Excel takes care of inserting the right code for us.

You might think of Excel as having two different ways of performing calculations. We can use standard cell references, in which case the formula for F4 would be E4*D4, or we can use column references inside a table. Using column references offers the advantage that we can use the same expression in all the cells of a column and Excel will compute the formula with a different value for each row.

Unlike Excel, DAX works only on tables. All the formulas must reference columns inside tables. For example, in DAX we write the previous multiplication this way:

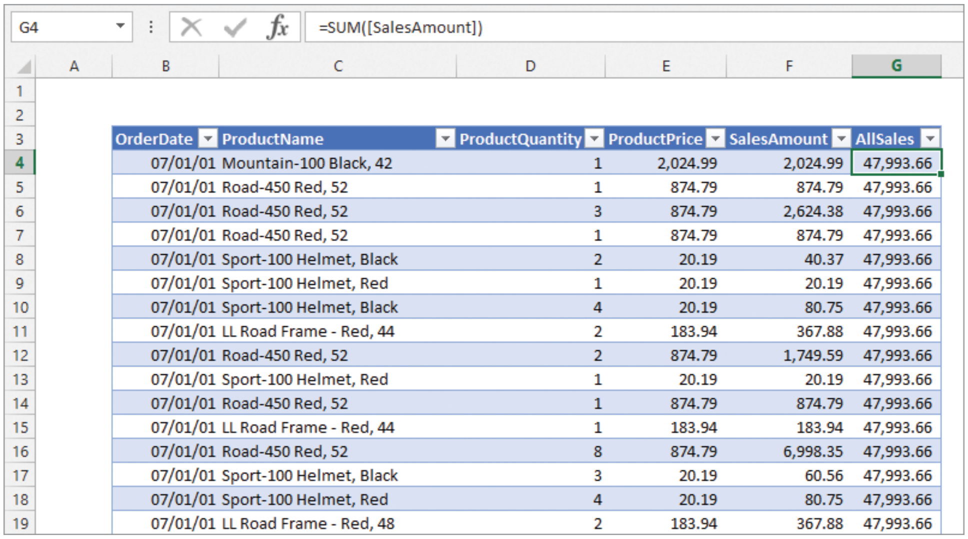
```
Sales[SalesAmount] = Sales[ProductPrice] * Sales[ProductQuantity]
```

As you can see, each column is prefixed with the name of its table. In Excel, we do not provide the table name because Excel formulas work inside a single table. However, DAX works on a data model containing many tables. As a consequence, we must specify the table name because two columns in different tables might have the same name.

Many functions in DAX work the same way as the equivalent Excel function. For example, the *IF* function reads the same way in DAX and in Excel:

```
Excel IF ( [@SalesAmount] > 10, 1, 0)
DAX IF ( Sales[SalesAmount] > 10, 1, 0)
```

One important aspect where the syntax of Excel and DAX is different is in the way to reference the entire column. In fact, in *[@ProductQuantity]*, the @ means "the value in the current row." In DAX, there is no need to specify that a value must be from the current row, because this is the default behavior of the language. In Excel, we can reference the entire column—that is, all the rows in that column—by removing the @ symbol. You can see this in Figure 1-6.


G4

=SUM([SalesAmount])

OrderDate	ProductName	ProductQuantity	ProductPrice	SalesAmount	AllSales
07/01/01	Mountain-100 Black, 42	1	2,024.99	2,024.99	47,993.66
07/01/01	Road-450 Red, 52	1	874.79	874.79	47,993.66
07/01/01	Road-450 Red, 52	3	874.79	2,624.38	47,993.66
07/01/01	Road-450 Red, 52	1	874.79	874.79	47,993.66
07/01/01	Sport-100 Helmet, Black	2	20.19	40.37	47,993.66
07/01/01	Sport-100 Helmet, Red	1	20.19	20.19	47,993.66
07/01/01	Sport-100 Helmet, Black	4	20.19	80.75	47,993.66
07/01/01	LL Road Frame - Red, 44	2	183.94	367.88	47,993.66
07/01/01	Road-450 Red, 52	2	874.79	1,749.59	47,993.66
07/01/01	Sport-100 Helmet, Red	1	20.19	20.19	47,993.66
07/01/01	Road-450 Red, 52	1	874.79	874.79	47,993.66
07/01/01	LL Road Frame - Red, 44	1	183.94	183.94	47,993.66
07/01/01	Road-450 Red, 52	8	874.79	6,998.35	47,993.66
07/01/01	Sport-100 Helmet, Black	3	20.19	60.56	47,993.66
07/01/01	Sport-100 Helmet, Red	4	20.19	80.75	47,993.66
07/01/01	LL Road Frame - Red, 48	2	183.94	367.88	47,993.66

FIGURE 1-6 In Excel, you can reference an entire column by omitting the @ symbol before the column name.

The value of the *AllSales* column is the same in all the rows because it is the grand total of the *SalesAmount* column. In other words, there is a syntactical difference between the value of a column in the current row and the value of the column as a whole.

DAX is different. In DAX, this is how you write the *AllSales* expression of Figure 1-6:

```
AllSales := SUM ( Sales[SalesAmount] )
```

There is no syntactical difference between retrieving the value of a column for a specific row and using the column as a whole. DAX understands that we want to sum all the values of the column because we use the column name inside an aggregator (in this case the *SUM* function), which requires a column name to be passed as a parameter. Thus, although Excel requires an explicit syntax to differentiate between the two types of data to retrieve, DAX does the disambiguation automatically. This distinction might be confusing—at least in the beginning.

Excel and DAX: Two functional languages

One aspect where the two languages are similar is that both Excel and DAX are functional languages. A functional language is made up of expressions that are basically function calls. In Excel and DAX, the concepts of statements, loops, and jumps do not exist although they are common to many programming languages. In DAX, everything is an expression. This aspect of the language is often a challenge for programmers coming from different languages, but it should be no surprise at all for Excel users.

Iterators in DAX

A concept that might be new to you is the concept of iterators. When working in Excel, you perform calculations one step at a time. The previous example showed that to compute the total of sales, we create one column containing the price multiplied by the quantity. Then as a second step, we sum it to compute the total sales. This number is then useful as a denominator to compute the percentage of sales of each product, for example.

Using DAX, you can perform the same operation in a single step by using iterators. An iterator does exactly what its name suggests: it iterates over a table and performs a calculation on each row of the table, aggregating the result to produce the single value requested.

Using the previous example, we can now compute the sum of all sales using the *SUMX* iterator:

```
AllSales :=
SUMX (
    Sales,
    Sales[ProductQuantity] * Sales[ProductPrice]
)
```

This approach brings to light both an advantage and a disadvantage. The advantage is that we can perform many complex calculations in a single step without worrying about adding columns that would end up being useful only for specific formulas. The disadvantage is that programming with DAX is less visual than programming with Excel. Indeed, you do not see the column computing the price multiplied by the quantity; it exists only for the lifetime of the calculation.

As we will explain later, we can create a calculated column that computes the multiplication of price by quantity. Nevertheless, doing so is seldom a good practice because it uses memory and might slow down the calculations, unless you use DirectQuery and Aggregations, as we explain in Chapter 18, "Optimizing VertiPaq."

DAX requires theory

Let us be clear: The fact that DAX requires one to study theory first is not a difference between programming languages. This is a difference in mindset. You are probably used to searching the web for complex formulas and solution patterns for the scenarios you are trying to solve. When you are using Excel, chances are you will find a formula that almost does what you need. You can copy the formula, customize it to fit your needs, and then use it without worrying too much about how it works.

This approach, which works in Excel, does not work with DAX, however. You need to study DAX theory and thoroughly understand how evaluation contexts work before you can write good DAX code. If you do not have a proper theoretical foundation, you will find that DAX either computes values like magic or it computes strange numbers that make no sense. The problem is not DAX but the fact that you do not yet understood exactly how DAX works.

Luckily, the theory behind DAX is limited to a couple of important concepts, which we explain in Chapter 4, "Understanding evaluation contexts." When you reach that chapter, be prepared for some intense learning. After you master that content, DAX will have no secrets for you, and learning DAX will mainly be a matter of gaining experience. Remember: knowing is half the battle. So do not try to go further until you are somewhat proficient with evaluation contexts.

DAX for SQL developers

If you are accustomed to the SQL language, you have already worked with many tables and created joins between columns to set relationships. From this point of view, you will likely feel at home in the DAX world. Indeed, computing in DAX is a matter of querying a set of tables joined by relationships and aggregating values.

Relationship handling

The first difference between SQL and DAX is in the way relationships work in the model. In SQL, we can set foreign keys between tables to declare relationships, but the engine never uses these foreign keys in queries unless we are explicit about them. For example, if we have a *Customers* table and a *Sales* table, where *CustomerKey* is a primary key in *Customers* and a foreign key in *Sales,* we can write the following query:

```
SELECT
   Customers.CustomerName,
   SUM ( Sales.SalesAmount ) AS SumOfSales
FROM
   Sales
   INNER JOIN Customers
    ON Sales.CustomerKey = Customers.CustomerKey
GROUP BY
   Customers.CustomerName
```

Though we declare the relationship in the model using foreign keys, we still need to be explicit and state the join condition in the query. Although this approach makes queries more verbose, it is useful because you can use different join conditions in different queries, giving you a lot of freedom in the way you express queries.

In DAX, relationships are part of the model, and they are all *LEFT OUTER JOINs*. When they are defined in the model, you no longer need to specify the join type in the query: DAX uses an automatic *LEFT OUTER JOIN* in the query whenever you use columns related to the primary table. Thus, in DAX you would write the previous SQL query as follows:

```
EVALUATE
SUMMARIZECOLUMNS (
    Customers[CustomerName],
    "SumOfSales", SUM ( Sales[SalesAmount] )
)
```

Because DAX knows the existing relationship between *Sales* and *Customers,* it does the join automatically following the model. Finally, the *SUMMARIZECOLUMNS* function needs to perform a group by *Customers[CustomerName],* but we do not have a keyword for that: *SUMMARIZECOLUMNS* automatically groups data by selected columns.

DAX is a functional language

SQL is a declarative language. You define what you need by declaring the set of data you want to retrieve using *SELECT* statements, without worrying about how the engine actually retrieves the information.

DAX, on the other hand, is a functional language. In DAX, every expression is a function call. Function parameters can, in turn, be other function calls. The evaluation of parameters might lead to complex query plans that DAX executes to compute the result.

For example, if we want to retrieve only customers who live in Europe, we can write this query in SQL:

```
SELECT
   Customers.CustomerName,
   SUM ( Sales.SalesAmount ) AS SumOfSales
FROM
   Sales
   INNER JOIN Customers
    ON Sales.CustomerKey = Customers.CustomerKey
WHERE
   Customers.Continent = 'Europe'
GROUP BY
   Customers.CustomerName
```

Using DAX, we do not declare the *WHERE* condition in the query. Instead, we use a specific function (*FILTER*) to filter the result:

```
EVALUATE
SUMMARIZECOLUMNS (
    Customers[CustomerName],
    FILTER (
        Customers,
        Customers[Continent] = "Europe"
    ),
    "SumOfSales", SUM ( Sales[SalesAmount] )
)
```

You can see that *FILTER* is a function: it returns only the customers living in Europe, producing the expected result. The order in which we nest the functions and the kinds of functions we use have a strong impact on both the result and the performance of the engine. This happens in SQL too, although in SQL we trust the query optimizer to find the optimal query plan. In DAX, although the query optimizer does a great job, you, as programmer, bear more responsibility in writing good code.

DAX as a programming and querying language

In SQL, a clear distinction exists between the query language and the programming language—that is, the set of instructions used to create stored procedures, views, and other pieces of code in the database. Each SQL dialect has its own statements to let programmers enrich the data model with code. However, DAX virtually makes no distinction between querying and programming. A rich set of functions manipulates tables and can, in turn, return tables. The *FILTER* function in the previous query is a good example of this.

In that respect, it appears that DAX is simpler than SQL. When you learn it as a programming language—its original use—you will know everything needed to also use it as a query language.

Subqueries and conditions in DAX and SQL

One of the most powerful features of SQL as a query language is the option of using subqueries. DAX features similar concepts. In the case of DAX subqueries, however, they stem from the functional nature of the language.

For example, to retrieve customers and total sales specifically for the customers who bought more than US$100 worth, we can write this query in SQL:

```
SELECT
   CustomerName,
   SumOfSales
FROM (
   SELECT
    Customers.CustomerName,
    SUM ( Sales.SalesAmount ) AS SumOfSales
   FROM
    Sales
    INNER JOIN Customers
     ON Sales.CustomerKey = Customers.CustomerKey
   GROUP BY
    Customers.CustomerName
   ) AS SubQuery
WHERE
   SubQuery.SumOfSales > 100
```

We can obtain the same result in DAX by nesting function calls:

```
EVALUATE
FILTER (
    SUMMARIZECOLUMNS (
        Customers[CustomerName],
        "SumOfSales", SUM ( Sales[SalesAmount] )
    ),
    [SumOfSales] > 100
)
```

In this code, the subquery that retrieves *CustomerName* and *SumOfSales* is later fed into a *FILTER* function that retains only the rows where *SumOfSales* is greater than 100. Right now, this code might seem unreadable to you. However, as soon as you start learning DAX, you will discover that using subqueries is much easier than in SQL, and it flows naturally because DAX is a functional language.

DAX for MDX developers

Many Business Intelligence professionals start learning DAX because it is the new language of Tabular. In the past, they used the MDX language to build and query Analysis Services Multidimensional models. If you are among them, be prepared to learn a completely new language: DAX and MDX do not share much in common. Worse, some concepts in DAX will remind you of similar existing concepts in MDX though they are different.

In our experience, we have found that learning DAX after learning MDX is the most challenging option. To learn DAX, you need to free your mind from MDX. Try to forget everything you know about multidimensional spaces and be prepared to learn this new language with a clear mind.

Multidimensional versus Tabular

MDX works in the multidimensional space defined by a model. The shape of the multidimensional space is based on the architecture of dimensions and hierarchies defined in the model, and this, in turn, defines the set of coordinates of the multidimensional space. Intersections of sets of members in different dimensions define points in the multidimensional space. You may have taken some time to realize that the *[All]* member of any attribute hierarchy is indeed a point in the multidimensional space.

DAX works in a much simpler way. There are no dimensions, no members, and no points in the multidimensional space. In other words, there is no multidimensional space at all. There are hierarchies, which we can define in the model, but they are different from hierarchies in MDX. The DAX space is built on top of tables, columns, and relationships. Each table in a Tabular model is neither a measure group nor a dimension: it is just a table, and to compute values, you scan it, filter it, or sum values inside it. Everything is based on the two simple concepts of tables and relationships.

You will soon discover that from the modeling point of view, Tabular offers fewer options than Multidimensional does. In this case, having fewer options does not mean being less powerful because you can use DAX as a programming language to enrich the model. The real modeling power of Tabular is the tremendous speed of DAX. In fact, you probably try to avoid overusing MDX in your model because optimizing MDX speed is often a challenge. DAX, on the other hand, is amazingly fast. Thus, most of the complexity of the calculations is not in the model but in the DAX formulas instead.

DAX as a programming and querying language

DAX and MDX are both programming languages and query languages. In MDX, the difference is made clear by the presence of the MDX script. You use MDX in the MDX script, along with several special statements that can be used in the script only, such as *SCOPE* statements. You use MDX in queries when you write *SELECT* statements that retrieve data. In DAX, this is somewhat different. You use DAX as a programming language to define calculated columns, calculated tables, and measures. The concept of calculated columns and calculated tables is new to DAX and does not exist in MDX; measures are similar to calculated members in MDX. You can also use DAX as a query language—for example, to retrieve data from a Tabular model using Reporting Services. Nevertheless, DAX functions do not have a specific role and can be used in both queries and calculation expressions. Moreover, you can also query a Tabular model using MDX. Thus, the querying part of MDX works with Tabular models, whereas DAX is the only option when it comes to programming a Tabular model.

Hierarchies

Using MDX, you rely on hierarchies to perform most of the calculations. If you wanted to compute the sales in the previous year, you would have to retrieve the *PrevMember* of the *CurrentMember* on the *Year* hierarchy and use it to override the MDX filter. For example, you can write the formula this way to define a previous year calculation in MDX:

```
CREATE MEMBER CURRENTCUBE.[Measures].[SamePeriodPreviousYearSales] AS
(
    [Measures].[Sales Amount],
    ParallelPeriod (
        [Date].[Calendar].[Calendar Year],
        1,
        [Date].[Calendar].CurrentMember
    )
);
```

The measure uses the *ParallelPeriod* function, which returns the cousin of the *CurrentMember* on the *Calendar* hierarchy. Thus, it is based on the hierarchies defined in the model. We would write the same calculation in DAX using filter contexts and standard time-intelligence functions:

```
SamePeriodPreviousYearSales :=
CALCULATE (
    SUM ( Sales[Sales Amount] ),
    SAMEPERIODLASTYEAR ( 'Date'[Date] )
)
```

We can write the same calculation in many other ways using *FILTER* and other DAX functions, but the idea remains the same: instead of using hierarchies, we filter tables. This difference is huge, and you will probably miss hierarchy calculations until you get used to DAX.

Another important difference is that in MDX you refer to *[Measures].[Sales Amount]*, and the aggregation function that you need to use is already defined in the model. In DAX, there is no predefined aggregation. In fact, as you might have noticed, the expression to compute is *SUM(Sales[Sales Amount])*. The predefined aggregation is no longer in the model. We need to define it whenever we want to use it. We can always create a measure that computes the sum of sales, but this would be beyond the scope of this section and is explained later in the book.

One more important difference between DAX and MDX is that MDX makes heavy use of the *SCOPE* statement to implement business logic (again, using hierarchies), whereas DAX needs a completely different approach. Indeed, hierarchy handling is missing in the language altogether.

For example, if we want to clear a measure at the *Year* level, in MDX we would write this statement:

```
SCOPE ( [Measures].[SamePeriodPreviousYearSales], [Date].[Month].[All] )
   THIS = NULL;
END SCOPE;
```

DAX does not have something like a *SCOPE* statement. To obtain the same result, we need to check for the presence of filters in the filter context, and the scenario is much more complex:

```
SamePeriodPreviousYearSales :=
IF (
    ISINSCOPE ( 'Date'[Month] ),
    CALCULATE (
        SUM ( Sales[Sales Amount] ),
        SAMEPERIODLASTYEAR ( 'Date'[Date] )
    ),
    BLANK ()
)
```

Intuitively, this formula returns a value only if the user is browsing the calendar hierarchy at the month level or below. Otherwise, it returns a *BLANK*. You will later learn what this formula computes in detail. It is much more error-prone than the equivalent MDX code. To be honest, hierarchy handling is one of the features that is really missing in DAX.

Leaf-level calculations

Finally, when using MDX, you probably got used to avoiding leaf-level calculations. Performing leaf-level computation in MDX turns out to be so slow that you should always prefer to precompute values and leverage aggregations to return results. In DAX, leaf-level calculations work incredibly fast and aggregations serve a different purpose, being useful only for large datasets. This requires a shift in your mind when it is time to build the data models. In most cases, a data model that fits perfectly in SSAS Multidimensional is not the right fit for Tabular and vice versa.

DAX for Power BI users

If you skipped the previous sections and directly came here, welcome! DAX is the native language of Power BI, and if you do not have experience in Excel, SQL, or MDX, Power BI will be the first place where you learn DAX. If you do not have previous experience in building models with other tools, you will learn that Power BI is a powerful analytical and modeling tool, with DAX as the perfect companion.

You might have started using Power BI a while ago and now you want to get to the next level. If this is the case, be prepared for a wonderful journey with DAX.

Here is our advice to you: do not expect to be able to write complex DAX code in a matter of a few days. DAX requires your time and dedication, and mastering it requires some practice. Based on our experience, you will be excited at first when you are rewarded with a few simple calculations.

The excitement fades away as soon as you start learning about evaluation contexts and *CALCULATE*, the most complex topics of the language. At that point, everything looks complex. Do not give up; most DAX developers had to move past that level. When you are there, you are so close to reaching a full understanding that it would be a real pity to stop. Read and practice again and again because a lightbulb will go off much sooner that you would expect. You will be able to finish the book quickly, reaching DAX guru status.

Evaluation contexts are at the core of the language. Mastering them takes time. We do not know anyone who was able to learn all about DAX in a couple of days. Besides, as with any complex topic, you will learn to appreciate a lot of the details over time. When you think you have learned everything, give the book a second read. You will discover many details that looked less important at first sight but, with a more trained mindset, really make a difference.

Enjoy the rest of this book!

CHAPTER 2

Introducing DAX

In this chapter, we start talking about the DAX language. Here you learn the syntax of the language, the difference between a calculated column and a measure (also called calculated field, in certain old Excel versions), and the most commonly used functions in DAX.

Because this is an introductory chapter, it does not cover many functions in depth. In later chapters, we explain them in more detail. For now, introducing the functions and starting to look at the DAX language in general are enough. When we reference features of the data model in Power BI, Power Pivot, or Analysis Services, we use the term *Tabular* even when the feature is not present in all the products. For example, "DirectQuery in Tabular" refers to the DirectQuery mode feature available in Power BI and Analysis Services but not in Excel.

Understanding DAX calculations

Before working on more complex formulas, you need to learn the basics of DAX. This includes DAX syntax, the different data types that DAX can handle, the basic operators, and how to refer to columns and tables. These concepts are discussed in the next few sections.

We use DAX to compute values over columns in tables. We can aggregate, calculate, and search for numbers, but in the end, all the calculations involve tables and columns. Thus, the first syntax to learn is how to reference a column in a table.

The general format is to write the table name enclosed in single quotation marks, followed by the column name enclosed in square brackets, as follows:

```
'Sales'[Quantity]
```

We can omit the single quotation marks if the table name does not start with a number, does not contain spaces, and is not a reserved word (like *Date* or *Sum*).

The table name is also optional in case we are referencing a column or a measure within the table where we define the formula. Thus, *[Quantity]* is a valid column reference, if written in a calculated column or in a measure defined in the *Sales* table. Although this option is available, we strongly discourage you from omitting the table name. At this point, we do not explain why this is so important, but the reason will become clear when you read Chapter 5, "Understanding CALCULATE and CALCULATETABLE." Nevertheless, it is of paramount importance to be able to distinguish between measures (discussed later) and columns when you read DAX code. The de facto standard is to always use the table name in column references and always avoid it in measure references. The earlier you start adopting this standard, the easier your life with DAX will be. Therefore, you should get used to this way of referencing columns and measures:

```
Sales[Quantity] * 2             -- This is a column reference
[Sales Amount] * 2              -- This is a measure reference
```

You will learn the rationale behind this standard after learning about context transition, which comes in Chapter 5. For now, just trust us and adhere to this standard.

Comments in DAX

The preceding code example shows comments in DAX for the first time. DAX supports single-line comments and multiline comments. Single-line comments start with either -- or //, and the remaining part of the line is considered a comment.

```
= Sales[Quantity] * Sales[Net Price]   -- Single-line comment
= Sales[Quantity] * Sales[Unit Cost]   // Another example of single-line comment
```

A multiline comment starts with /* and ends with */. The DAX parser ignores everything included between these markers and considers them a comment.

```
= IF (
    Sales[Quantity] > 1,
    /* First example of a multiline comment
       Anything can be written here and is ignored by DAX
    */
    "Multi",
    /* A common use case of multiline comments is to comment-out a part of
       the existing code
       The next IF statement is ignored because it falls within a multiline comment
        IF (
            Sales[Quantity] = 1,
            "Single",
            "Special note"
        )
    */
    "Single"
)
```

It is better to avoid comments at the end of a DAX expression in a measure, calculated column, or calculated table definition. These comments might be not visible at first, and they might not be supported by tools such as DAX Formatter, which is discussed later in this chapter.

DAX data types

DAX can perform computations with different numeric types, of which there are seven. Over time, Microsoft introduced different names for the same data types, creating some sort of confusion. Table 2-1 provides the different names under which you might find each DAX data type.

TABLE 2-1 Data Types

DAX Data Type	Power BI Data Type	Power Pivot and Analysis Services Data Type	Correspondent Conventional Data Type (e.g., SQL Server)	Tabular Object Model (TOM) Data Type
Integer	Whole Number	Whole Number	Integer / INT	int64
Decimal	Decimal Number	Decimal Number	Floating point / DOUBLE	double
Currency	Fixed Decimal Number	Currency	Currency / MONEY	decimal
DateTime	DateTime, Date, Time	Date	Date / DATETIME	dateTime
Boolean	True/False	True/False	Boolean / BIT	boolean
String	Text	Text	String / NVARCHAR(MAX)	string
Variant	-	-	-	variant
Binary	Binary	Binary	Blob / VARBINARY(MAX)	binary

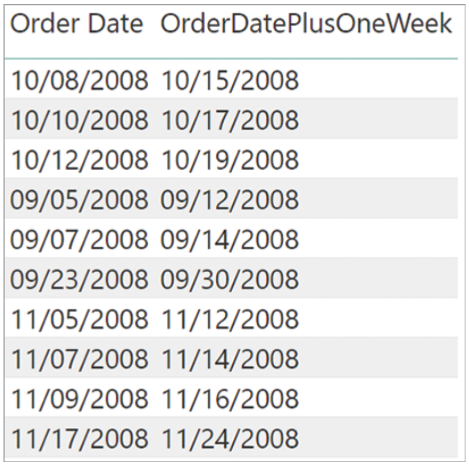
In this book, we use the names in the first column of Table 2-1 adhering to the de facto standards in the database and Business Intelligence community. For example, in Power BI, a column containing either *TRUE* or *FALSE* would be called *TRUE/FALSE*, whereas in SQL Server, it would be called a *BIT*. Nevertheless, the historical and most common name for this type of value is Boolean.

DAX comes with a powerful type-handling system so that we do not have to worry about data types. In a DAX expression, the resulting type is based on the type of the term used in the expression. You need to be aware of this in case the type returned from a DAX expression is not the expected type; you would then have to investigate the data type of the terms used in the expression itself.

For example, if one of the terms of a sum is a date, the result also is a date; likewise, if the same operator is used with integers, the result is an integer. This behavior is known as *operator overloading*, and an example is shown in Figure 2-1, where the *OrderDatePlusOneWeek* column is calculated by adding 7 to the value of the *Order Date* column.

```
Sales[OrderDatePlusOneWeek] = Sales[Order Date] + 7
```

The result is a date.



Order Date	OrderDatePlusOneWeek
10/08/2008	10/15/2008
10/10/2008	10/17/2008
10/12/2008	10/19/2008
09/05/2008	09/12/2008
09/07/2008	09/14/2008
09/23/2008	09/30/2008
11/05/2008	11/12/2008
11/07/2008	11/14/2008
11/09/2008	11/16/2008
11/17/2008	11/24/2008

FIGURE 2-1 Adding an integer to a date results in a date increased by the corresponding number of days.

In addition to operator overloading, DAX automatically converts strings into numbers and numbers into strings whenever required by the operator. For example, if we use the *&* operator, which concatenates strings, DAX converts its arguments into strings. The following formula returns "54" as a string:

```
= 5 & 4
```

On the other hand, this formula returns an integer result with the value of 9:

```
= "5" + "4"
```

The resulting value depends on the operator and not on the source columns, which are converted following the requirements of the operator. Although this behavior looks convenient, later in this chapter you see what kinds of errors might happen during these automatic conversions. Moreover, not all the operators follow this behavior. For example, comparison operators cannot compare strings with numbers. Consequently, you can add one number with a string, but you cannot compare a number with a string. You can find a complete reference here: https://docs.microsoft.com/en-us/power-bi/desktop-data-types. Because the rules are so complex, we suggest you avoid automatic conversions altogether. If a conversion needs to happen, we recommend that you control it and make the conversion explicit. To be more explicit, the previous example should be written like this:

```
= VALUE ( "5" ) + VALUE ( "4" )
```

People accustomed to working with Excel or other languages might be familiar with DAX data types. Some details about data types depend on the engine, and they might be different for Power BI, Power

Pivot, or Analysis Services. You can find more detailed information about Analysis Services DAX data types at http://msdn.microsoft.com/en-us/library/gg492146.aspx, and Power BI information is available at https://docs.microsoft.com/en-us/power-bi/desktop-data-types. However, it is useful to share a few considerations about each of these data types.

Integer

DAX has only one *Integer* data type that can store a 64-bit value. All the internal calculations between integer values in DAX also use a 64-bit value.

Decimal

A *Decimal* number is always stored as a double-precision floating-point value. Do not confuse this DAX data type with the *decimal* and *numeric* data type of *Transact-SQL*. The corresponding data type of a DAX decimal number in SQL is *Float*.

Currency

The *Currency* data type, also known as *Fixed Decimal Number* in Power BI, stores a fixed decimal number. It can represent four decimal points and is internally stored as a 64-bit integer value divided by 10,000. Summing or subtracting *Currency* data types always ignores decimals beyond the fourth decimal point, whereas multiplication and division produce a floating-point value, thus increasing the precision of the result. In general, if we need more accuracy than the four digits provided, we must use a *Decimal* data type.

The default format of the *Currency* data type includes the currency symbol. We can also apply the currency formatting to *Integer* and decimal numbers, and we can use a format without the currency symbol for a *Currency* data type.

DateTime

DAX stores dates as a *DateTime* data type. This format uses a floating-point number internally, wherein the integer corresponds to the number of days since December 30, 1899, and the decimal part identifies the fraction of the day. Hours, minutes, and seconds are converted to decimal fractions of a day. Thus, the following expression returns the current date plus one day (exactly 24 hours):

```
= TODAY () + 1
```

The result is tomorrow’s date at the time of the evaluation. If you need to take only the date part of a *DateTime,* always remember to use *TRUNC* to get rid of the decimal part.

Power BI offers two additional data types: *Date* and *Time*. Internally, they are a simple variation of *DateTime*. Indeed, *Date* and *Time* store only the integer or the decimal part of the *DateTime*, respectively.

The leap year bug

Lotus 1-2-3, a popular spreadsheet released in 1983, presented a bug in the handling of the *DateTime* data type. It considered 1900 as a leap year, even though it was not. The final year in a century is a leap year only if the first two digits can be divided by 4 without a remainder. At that time, the development team of the first version of Excel deliberately replicated the bug, to maintain compatibility with Lotus 1-2-3. Since then, each new version of Excel has maintained the bug for compatibility.

At the time of printing in 2019, the bug is still there in DAX, introduced for backward compatibility with Excel. The presence of the bug (should we call it a feature?) might lead to errors on time periods prior to March 1, 1900. Thus, by design, the first date officially supported by DAX is March 1, 1900. Date calculations executed on time periods prior to that date might lead to errors and should be considered as inaccurate.

Boolean

The *Boolean* data type is used to express logical conditions. For example, a calculated column defined by the following expression is of *Boolean* type:

```
= Sales[Unit Price] > Sales[Unit Cost]
```

You will also see *Boolean* data types as numbers where *TRUE* equals 1 and *FALSE* equals 0. This notation sometimes proves useful for sorting purposes because *TRUE* > *FALSE*.

String

Every string in DAX is stored as a *Unicode* string, where each character is stored in 16 bits. By default, the comparison between strings is not case sensitive, so the two strings "Power BI" and "POWER BI" are considered equal.

Variant

The *Variant* data type is used for expressions that might return different data types, depending on the conditions. For example, the following statement can return either an integer or a string, so it returns a variant type:

```
IF ( [measure] > 0, 1, "N/A" )
```

The *Variant* data type cannot be used as a data type for a column in a regular table. A DAX measure, and in general, a DAX expression can be *Variant*.

Binary

The *Binary* data type is used in the data model to store images or other nonstructured types of information. It is not available in DAX. It was mainly used by Power View, but it might not be available in other tools such as Power BI.

DAX operators

Now that you have seen the importance of operators in determining the type of an expression, see Table 2-2, which provides a list of the operators available in DAX.

TABLE 2-2 Operators

Operator Type	Symbol	Use	Example
Parenthesis	( )	Precedence order and grouping of arguments	(5 + 2) * 3
Arithmetic	+ − * /	Addition Subtraction/negation Multiplication Division	4 + 2 5 − 3 4 * 2 4 / 2
Comparison	= <> > >= < <=	Equal to Not equal to Greater than Greater than or equal to Less than Less than or equal to	[CountryRegion] = "USA" [CountryRegion] <> "USA" [Quantity] > 0 [Quantity] >= 100 [Quantity] < 0 [Quantity] <= 100
Text concatenation	&	Concatenation of strings	"Value is" & [Amount]
Logical	&& || IN NOT	AND condition between two Boolean expressions OR condition between two Boolean expressions Inclusion of an element in a list Boolean negation	[CountryRegion] = "USA" && [Quantity]>0 [CountryRegion] = "USA" || [Quantity] > 0 [CountryRegion] IN {"USA", "Canada"} NOT [Quantity] > 0

Moreover, the logical operators are also available as DAX functions, with a syntax similar to Excel's. For example, we can write expressions like these:

```
AND ( [CountryRegion] = "USA", [Quantity] > 0 )
OR ( [CountryRegion] = "USA", [Quantity] > 0 )
```

These examples are equivalent, respectively, to the following:

```
[CountryRegion] = "USA" && [Quantity] > 0
[CountryRegion] = "USA" || [Quantity] > 0
```

Using functions instead of operators for Boolean logic becomes helpful when writing complex conditions. In fact, when it comes to formatting large sections of code, functions are much easier to format and to read than operators are. However, a major drawback of functions is that we can pass in only two parameters at a time. Therefore, we must nest functions if we have more than two conditions to evaluate.

Table constructors

In DAX we can define anonymous tables directly in the code. If the table has a single column, the syntax requires only a list of values—one for each row—delimited by curly braces. We can delimit multiple rows by parentheses, which are optional if the table is made of a single column. The two following definitions, for example, are equivalent:

```
{ "Red", "Blue", "White" }
{ ( "Red" ), ( "Blue" ), ( "White" ) }
```

If the table has multiple columns, parentheses are mandatory. Every column should have the same data type throughout all its rows; otherwise, DAX will automatically convert the column to a data type that can accommodate all the data types provided in different rows for the same column.

```
{
    ( "A", 10, 1.5, DATE ( 2017, 1, 1 ), CURRENCY ( 199.99 ), TRUE ),
    ( "B", 20, 2.5, DATE ( 2017, 1, 2 ), CURRENCY ( 249.99 ), FALSE ),
    ( "C", 30, 3.5, DATE ( 2017, 1, 3 ), CURRENCY ( 299.99 ), FALSE )
}
```

The table constructor is commonly used with the *IN* operator. For example, the following are possible, valid syntaxes in a DAX predicate:

```
'Product'[Color] IN { "Red", "Blue", "White" }

( 'Date'[Year], 'Date'[MonthNumber] ) IN { ( 2017, 12 ), ( 2018, 1 ) }
```

This second example shows the syntax required to compare a set of columns (tuple) using the *IN* operator. Such syntax cannot be used with a comparison operator. In other words, the following syntax is not valid:

```
( 'Date'[Year], 'Date'[MonthNumber] ) = ( 2007, 12 )
```

However, we can rewrite it using the *IN* operator with a table constructor that has a single row, as in the following example:

```
( 'Date'[Year], 'Date'[MonthNumber] ) IN { ( 2007, 12 ) }
```

Conditional statements

In DAX we can write a conditional expression using the *IF* function. For example, we can write an expression returning MULTI or SINGLE depending on the quantity value being greater than one or not, respectively.

```
IF (
    Sales[Quantity] > 1,
    "MULTI",
    "SINGLE"
)
```

The *IF* function has three parameters, but only the first two are mandatory. The third is optional, and it defaults to *BLANK*. Consider the following code:

```
IF (
    Sales[Quantity] > 1,
    Sales[Quantity]
)
```

It corresponds to the following explicit version:

```
IF (
    Sales[Quantity] > 1,
    Sales[Quantity],
    BLANK ()
)
```

Understanding calculated columns and measures

Now that you know the basics of DAX syntax, you need to learn one of the most important concepts in DAX: the difference between calculated columns and measures. Even though calculated columns and measures might appear similar at first sight because you can make certain calculations using either, they are, in reality, different. Understanding the difference is key to unlocking the power of DAX.

Calculated columns

Depending on the tool you are using, you can create a calculated column in different ways. Indeed, the concept remains the same: a calculated column is a new column added to your model, but instead of being loaded from a data source, it is created by resorting to a DAX formula.

A calculated column is just like any other column in a table, and we can use it in rows, columns, filters, or values of a matrix or any other report. We can also use a calculated column to define a relationship, if needed. The DAX expression defined for a calculated column operates in the context of the current row of the table that the calculated column belongs to. Any reference to a column returns the value of that column for the current row. We cannot directly access the values of other rows.

If you are using the default *Import Mode* of Tabular and are not using DirectQuery, one important concept to remember about calculated columns is that these columns are computed during database processing and then stored in the model. This concept might seem strange if you are accustomed to SQL-computed columns (not persisted), which are evaluated at query time and do not use memory. In Tabular, however, all calculated columns occupy space in memory and are computed during table processing.

This behavior is helpful whenever we create complex calculated columns. The time required to compute complex calculated columns is always process time and not query time, resulting in a better user experience. Nevertheless, be mindful that a calculated column uses precious RAM. For example, if we have a complex formula for a calculated column, we might be tempted to separate the steps of computation into different intermediate columns. Although this technique is useful during project development, it is a bad habit in production because each intermediate calculation is stored in RAM and wastes valuable space.

If a model is based on DirectQuery instead, the behavior is hugely different. In DirectQuery mode, calculated columns are computed on the fly when the Tabular engine queries the data source. This might result in heavy queries executed by the data source, therefore producing slow models.

Computing the duration of an order

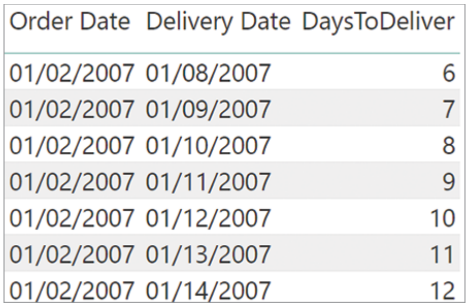
Imagine we have a *Sales* table containing both the order and the delivery dates. Using these two columns, we can compute the number of days involved in delivering the order. Because dates are stored as number of days after 12/30/1899, a simple subtraction computes the difference in days between two dates:

```
Sales[DaysToDeliver] = Sales[Delivery Date] - Sales[Order Date]
```

Nevertheless, because the two columns used for subtraction are dates, the result also is a date. To produce a numeric result, convert the result to an integer this way:

```
Sales[DaysToDeliver] = INT ( Sales[Delivery Date] - Sales[Order Date] )
```

The result is shown in Figure 2-2.



Order Date	Delivery Date	DaysToDeliver
01/02/2007	01/08/2007	6
01/02/2007	01/09/2007	7
01/02/2007	01/10/2007	8
01/02/2007	01/11/2007	9
01/02/2007	01/12/2007	10
01/02/2007	01/13/2007	11
01/02/2007	01/14/2007	12

FIGURE 2-2 By subtracting two dates and converting the result to an integer, DAX computes the number of days between the two dates.

Measures

Calculated columns are useful, but you can define calculations in a DAX model in another way. Whenever you do not want to compute values for each row but rather want to aggregate values from many rows in a table, you will find these calculations useful; they are called *measures*.

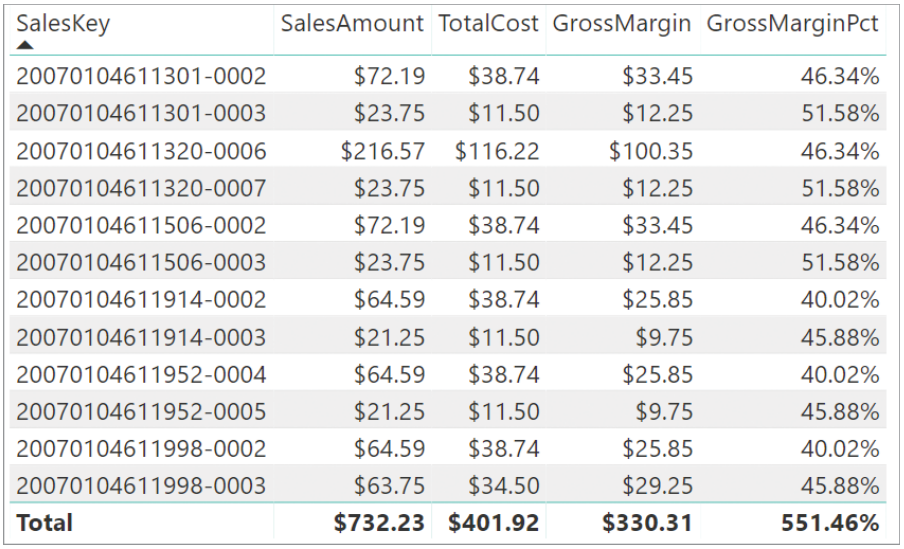
For example, you can define a few calculated columns in the *Sales* table to compute the gross margin amount:

```
Sales[SalesAmount] = Sales[Quantity] * Sales[Net Price]
Sales[TotalCost] = Sales[Quantity] * Sales[Unit Cost]
Sales[GrossMargin] = Sales[SalesAmount] – Sales[TotalCost]
```

What happens if you want to show the gross margin as a percentage of the sales amount? You could create a calculated column with the following formula:

```
Sales[GrossMarginPct] = Sales[GrossMargin] / Sales[SalesAmount]
```

This formula computes the correct value at the row level—as you can see in Figure 2-3—but at the grand total level the result is clearly wrong.

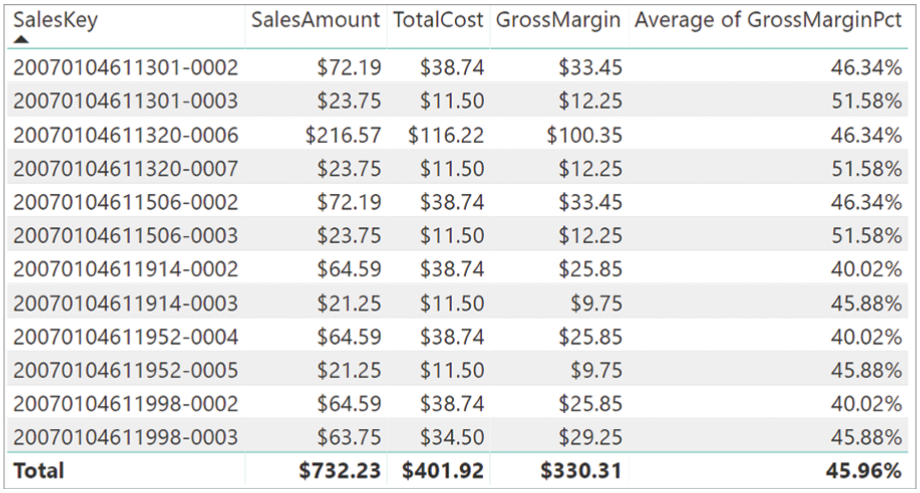


SalesKey	SalesAmount	TotalCost	GrossMargin	GrossMarginPct
20070104611301-0002	$72.19	$38.74	$33.45	46.34%
20070104611301-0003	$23.75	$11.50	$12.25	51.58%
20070104611320-0006	$216.57	$116.22	$100.35	46.34%
20070104611320-0007	$23.75	$11.50	$12.25	51.58%
20070104611506-0002	$72.19	$38.74	$33.45	46.34%
20070104611506-0003	$23.75	$11.50	$12.25	51.58%
20070104611914-0002	$64.59	$38.74	$25.85	40.02%
20070104611914-0003	$21.25	$11.50	$9.75	45.88%
20070104611952-0004	$64.59	$38.74	$25.85	40.02%
20070104611952-0005	$21.25	$11.50	$9.75	45.88%
20070104611998-0002	$64.59	$38.74	$25.85	40.02%
20070104611998-0003	$63.75	$34.50	$29.25	45.88%
Total	**$732.23**	**$401.92**	**$330.31**	**551.46%**

FIGURE 2-3 The *GrossMarginPct* column shows a correct value on each row, but the grand total is incorrect.

The value shown at the grand total level is the sum of the individual percentages computed row by row within the calculated column. When we compute the aggregate value of a percentage, we cannot rely on calculated columns. Instead, we need to compute the percentage based on the sum of individual columns. We must compute the aggregated value as the sum of gross margin divided by the sum of sales amount. In this case, we need to compute the ratio on the aggregates; you cannot use an aggregation of calculated columns. In other words, we compute the ratio of the sums, not the sum of the ratios.

It would be equally wrong to simply change the aggregation of the *GrossMarginPct* column to an average and rely on the result because doing so would provide an incorrect evaluation of the percentage, not considering the differences between amounts. The result of this averaged value is visible in Figure 2-4, and you can easily check that (330.31 / 732.23) is not equal to the value displayed, 45.96%; it should be 45.11% instead.



SalesKey	SalesAmount	TotalCost	GrossMargin	Average of GrossMarginPct
20070104611301-0002	$72.19	$38.74	$33.45	46.34%
20070104611301-0003	$23.75	$11.50	$12.25	51.58%
20070104611320-0006	$216.57	$116.22	$100.35	46.34%
20070104611320-0007	$23.75	$11.50	$12.25	51.58%
20070104611506-0002	$72.19	$38.74	$33.45	46.34%
20070104611506-0003	$23.75	$11.50	$12.25	51.58%
20070104611914-0002	$64.59	$38.74	$25.85	40.02%
20070104611914-0003	$21.25	$11.50	$9.75	45.88%
20070104611952-0004	$64.59	$38.74	$25.85	40.02%
20070104611952-0005	$21.25	$11.50	$9.75	45.88%
20070104611998-0002	$64.59	$38.74	$25.85	40.02%
20070104611998-0003	$63.75	$34.50	$29.25	45.88%
Total	**$732.23**	**$401.92**	**$330.31**	**45.96%**

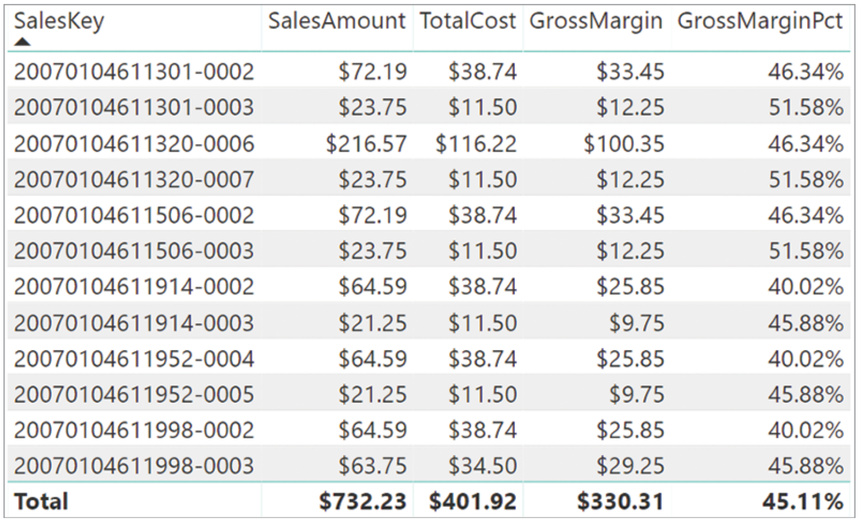
FIGURE 2-4 Changing the aggregation method to *AVERAGE* does not provide the correct result.

The correct implementation for *GrossMarginPct* is with a measure:

```
GrossMarginPct := SUM ( Sales[GrossMargin] ) / SUM (Sales[SalesAmount] )
```

As we have already stated, the correct result cannot be achieved with a calculated column. If you need to operate on aggregated values instead of operating on a row-by-row basis, you must create measures. You might have noticed that we used := to define a measure instead of the equal sign (=). This is a standard we used throughout the book to make it easier to differentiate between measures and calculated columns in code.

After you define *GrossMarginPct* as a measure, the result is correct, as you can see in Figure 2-5.



SalesKey	SalesAmount	TotalCost	GrossMargin	GrossMarginPct
20070104611301-0002	$72.19	$38.74	$33.45	46.34%
20070104611301-0003	$23.75	$11.50	$12.25	51.58%
20070104611320-0006	$216.57	$116.22	$100.35	46.34%
20070104611320-0007	$23.75	$11.50	$12.25	51.58%
20070104611506-0002	$72.19	$38.74	$33.45	46.34%
20070104611506-0003	$23.75	$11.50	$12.25	51.58%
20070104611914-0002	$64.59	$38.74	$25.85	40.02%
20070104611914-0003	$21.25	$11.50	$9.75	45.88%
20070104611952-0004	$64.59	$38.74	$25.85	40.02%
20070104611952-0005	$21.25	$11.50	$9.75	45.88%
20070104611998-0002	$64.59	$38.74	$25.85	40.02%
20070104611998-0003	$63.75	$34.50	$29.25	45.88%
Total	**$732.23**	**$401.92**	**$330.31**	**45.11%**

FIGURE 2-5 *GrossMarginPct* defined as a measure shows the correct grand total.

Measures and calculated columns both use DAX expressions; the difference is the context of evaluation. A measure is evaluated in the context of a visual element or in the context of a DAX query. However, a calculated column is computed at the row level of the table it belongs to. The context of the visual element (later in the book, you will learn that this is a filter context) depends on user selections in the report or on the format of the DAX query. Therefore, when using *SUM(Sales[SalesAmount])* in a measure, we mean the sum of all the rows that are aggregated under a visualization. However, when we use *Sales[SalesAmount]* in a calculated column, we mean the value of the *SalesAmount* column in the current row.

A measure needs to be defined in a table. This is one of the requirements of the DAX language. However, the measure does not really belong to the table. Indeed, we can move a measure from one table to another table without losing its functionality.

Differences between calculated columns and measures

Although they look similar, there is a big difference between calculated columns and measures. The value of a calculated column is computed during data refresh, and it uses the current row as a context. The result does not depend on user activity on the report. A measure operates on aggregations of data defined by the current context. In a matrix or in a pivot table, for example, source tables are filtered according to the coordinates of cells, and data is aggregated and calculated using these filters. In other words, a measure always operates on aggregations of data under the evaluation context. The evaluation context is explained further in Chapter 4, "Understanding evaluation contexts."

Choosing between calculated columns and measures

Now that you have seen the difference between calculated columns and measures, it is useful to discuss when to use one over the other. Sometimes either is an option, but in most situations, the computation requirements determine the choice.

As a developer, you must define a calculated column whenever you want to do the following:

- Place the calculated results in a slicer or see results in rows or columns in a matrix or in a pivot table (as opposed to the Values area), or use the calculated column as a filter condition in a DAX query.
- Define an expression that is strictly bound to the current row. For example, *Price * Quantity* cannot work on an average or on a sum of those two columns.
- Categorize text or numbers. For example, a range of values for a measure, a range of ages of customers, such as 0–18, 18–25, and so on. These categories are often used as filters or to slice and dice values.

However, it is mandatory to define a measure whenever one wants to display calculation values that reflect user selections, and the values need to be presented as aggregates in a report, for example:

- To calculate the profit percentage of a report selection
- To calculate ratios of a product compared to all products but keep the filter both by year and by region

We can express many calculations both with calculated columns and with measures, although we need to use different DAX expressions for each. For example, one can define the *GrossMargin* as a calculated column:

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

However, it can also be defined as a measure:

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

We suggest you use a measure in this case because, being evaluated at query time, it does not consume memory and disk space. As a rule, whenever you can express a calculation both ways, measures are the preferred way to go. You should limit the use of calculated columns to the few cases where they are strictly needed. Users with Excel experience typically prefer calculated columns over measures because calculated columns closely resemble the way of performing calculations in Excel. Nevertheless, the best way to compute a value in DAX is through a measure.

Using measures in calculated columns

It is obvious that a measure can refer to one or more calculated columns. Although less intuitive, the opposite is also true. A calculated column can refer to a measure. This way, the calculated column forces the calculation of a measure for the context defined by the current row. This operation transforms and consolidates the result of a measure into a column, which will not be influenced by user actions. Obviously, only certain operations can produce meaningful results because a measure usually makes computations that strongly depend on the selection made by the user in the visualization. Moreover, whenever you, as the developer, use measures in a calculated column, you rely on a feature called *context transition*, which is an advanced calculation technique in DAX. Before you use a measure in a calculated column, we strongly suggest you read and understand Chapter 4, which explains in detail evaluation contexts and context transitions.

Introducing variables

When writing a DAX expression, one can avoid repeating the same expression and greatly enhance the code readability by using variables. For example, look at the following expression:

```
VAR TotalSales = SUM ( Sales[SalesAmount] )
VAR TotalCosts = SUM ( Sales[TotalProductCost] )
VAR GrossMargin = TotalSales - TotalCosts
RETURN
    GrossMargin / TotalSales
```

Variables are defined with the *VAR* keyword. After you define a variable, you need to provide a *RETURN* section that defines the result value of the expression. One can define many variables, and the variables are local to the expression in which they are defined.

A variable defined in an expression cannot be used outside the expression itself. There is no such thing as a global variable definition. This means that you cannot define variables used through the whole DAX code of the model.

Variables are computed using lazy evaluation. This means that if one defines a variable that, for any reason, is not used in the code, the variable itself will never be evaluated. If it needs to be computed, this happens only once. Later uses of the variable will read the value computed previously. Thus, variables are also useful as an optimization technique when used in a complex expression multiple times.

Variables are an important tool in DAX. As you will learn in Chapter 4, variables are extremely useful because they use the definition evaluation context instead of the context where the variable is used. In Chapter 6, "Variables," we will fully cover variables and how to use them. We will also use variables extensively throughout the book.

Handling errors in DAX expressions

Now that you have seen some of the basics of the syntax, it is time to learn how to handle invalid calculations gracefully. A DAX expression might contain invalid calculations because the data it references is not valid for the formula. For example, the formula might contain a division by zero or reference a column value that is not a number while being used in an arithmetic operation such as multiplication. It is good to learn how these errors are handled by default and how to intercept these conditions for special handling.

Before discussing how to handle errors, though, we describe the different kinds of errors that might appear during a DAX formula evaluation. They are

- Conversion errors
- Arithmetic operations errors
- Empty or missing values

Conversion errors

The first kind of error is the conversion error. As we showed previously in this chapter, DAX automatically converts values between strings and numbers whenever the operator requires it. All these examples are valid DAX expressions:

```
"10" + 32 = 42
"10" & 32 = "1032"
```

```
10 & 32 = "1032"
DATE (2010,3,25) = 3/25/2010
DATE (2010,3,25) + 14 = 4/8/2010
DATE (2010,3,25) & 14 = "3/25/201014"
```

These formulas are always correct because they operate with constant values. However, what about the following formula if *VatCode* is a string?

```
Sales[VatCode] + 100
```

Because the first operand of this sum is a column that is of *Text* data type, you as a developer must be confident that DAX can convert all the values in that column into numbers. If DAX fails in converting some of the content to suit the operator needs, a conversion error will occur. Here are some typical situations:

```
"1 + 1" + 0 = Cannot convert value '1 + 1' of type Text to type Number
DATEVALUE ("25/14/2010") = Type mismatch
```

If you want to avoid these errors, it is important to add error detection logic in DAX expressions to intercept error conditions and return a result that makes sense. One can obtain the same result by intercepting the error after it has happened or by checking the operands for the error situation beforehand. Nevertheless, checking for the error situation proactively is better than letting the error happen and then catching it.

Arithmetic operations errors

The second category of errors is arithmetic operations, such as the division by zero or the square root of a negative number. These are not conversion-related errors: DAX raises them whenever we try to call a function or use an operator with invalid values.

The division by zero requires special handling because its behavior is not intuitive (except, maybe, for mathematicians). When one divides a number by zero, DAX returns the special value *Infinity*. In the special cases of 0 divided by 0 or *Infinity* divided by *Infinity*, DAX returns the special *NaN* (not a number) value.

Because this is unusual behavior, it is summarized in Table 2-3.

TABLE 2-3 Special Result Values for Division by Zero

Expression	Result
10 / 0	Infinity
7 / 0	Infinity
0 / 0	NaN
(10 / 0) / (7 / 0)	NaN

It is important to note that *Infinity* and *NaN* are not errors but special values in DAX. In fact, if one divides a number by *Infinity*, the expression does not generate an error. Instead, it returns 0:

```
9954 / ( 7 / 0 ) = 0
```

Apart from this special situation, DAX can return arithmetic errors when calling a function with an incorrect parameter, such as the square root of a negative number:

```
SQRT ( -1 ) = An argument of function 'SQRT' has the wrong data type or the result is too
large or too small
```

If DAX detects errors like this, it blocks any further computation of the expression and raises an error. One can use the *ISERROR* function to check if an expression leads to an error. We show this scenario later in this chapter.

Keep in mind that special values like *NaN* are displayed in the user interface of several tools such as Power BI as regular values. They can, however, be treated as errors when shown by other client tools such as an Excel pivot table. Finally, these special values are detected as errors by the error detection functions.

Empty or missing values

The third category that we examine is not a specific error condition but rather the presence of empty values. Empty values might result in unexpected results or calculation errors when combined with other elements in a calculation.

DAX handles missing values, blank values, or empty cells in the same way, using the value *BLANK*. *BLANK* is not a real value but instead is a special way to identify these conditions. We can obtain the value *BLANK* in a DAX expression by calling the *BLANK* function, which is different from an empty string. For example, the following expression always returns a blank value, which can be displayed as either an empty string or as "(blank)" in different client tools:

```
= BLANK ()
```

On its own, this expression is useless, but the *BLANK* function itself becomes useful every time there is the need to return an empty value. For example, one might want to display an empty result instead of 0. The following expression calculates the total discount for a sale transaction, leaving the blank value if the discount is 0:

```
=IF (
    Sales[DiscountPerc] = 0,             -- Check if there is a discount
    BLANK (),                            -- Return a blank if no discount is present
    Sales[DiscountPerc] * Sales[Amount]  -- Compute the discount otherwise
)
```

BLANK, by itself, is not an error; it is just an empty value. Therefore, an expression containing a *BLANK* might return a value or a blank, depending on the calculation required. For example, the following expression returns *BLANK* whenever *Sales[Amount]* is *BLANK*:

```
= 10 * Sales[Amount]
```

In other words, the result of an arithmetic product is *BLANK* whenever one or both terms are *BLANK*. This creates a challenge when it is necessary to check for a blank value. Because of the implicit conversions, it is impossible to distinguish whether an expression is 0 (or empty string) or *BLANK* using an equal operator. Indeed, the following logical conditions are always true:

```
BLANK () = 0     -- Always returns TRUE
BLANK () = ""    -- Always returns TRUE
```

Therefore, if the columns *Sales[DiscountPerc]* or *Sales[Clerk]* are blank, the following conditions return *TRUE* even if the test is against 0 and empty string, respectively:

```
Sales[DiscountPerc] = 0  -- Returns TRUE if DiscountPerc is either BLANK or 0
Sales[Clerk] = ""        -- Returns TRUE if Clerk is either BLANK or ""
```

In such cases, one can use the *ISBLANK* function to check whether a value is *BLANK* or not:

```
ISBLANK ( Sales[DiscountPerc] )  -- Returns TRUE only if DiscountPerc is BLANK
ISBLANK ( Sales[Clerk] )         -- Returns TRUE only if Clerk is BLANK
```

The propagation of *BLANK* in a DAX expression happens in several other arithmetic and logical operations, as shown in the following examples:

```
BLANK () + BLANK () = BLANK ()
10 * BLANK () = BLANK ()
BLANK () / 3 = BLANK ()
BLANK () / BLANK () = BLANK ()
```

However, the propagation of *BLANK* in the result of an expression does not happen for all formulas. Some calculations do not propagate *BLANK*. Instead, they return a value depending on the other terms of the formula. Examples of these are addition, subtraction, division by *BLANK*, and a logical operation including a *BLANK*. The following expressions show some of these conditions along with their results:

```
BLANK () - 10 = -10
18 + BLANK () = 18
4 / BLANK () = Infinity
0 / BLANK () = NaN
BLANK () || BLANK () = FALSE
BLANK () && BLANK () = FALSE
( BLANK () = BLANK () ) = TRUE
( BLANK () = TRUE ) = FALSE
( BLANK () = FALSE ) = TRUE
( BLANK () = 0 ) = TRUE
( BLANK () = "" ) = TRUE
ISBLANK ( BLANK() ) = TRUE
FALSE || BLANK () = FALSE
FALSE && BLANK () = FALSE
TRUE || BLANK () = TRUE
TRUE && BLANK () = FALSE
```

Empty values in Excel and SQL

Excel has a different way of handling empty values. In Excel, all empty values are considered 0 whenever they are used in a sum or in a multiplication, but they might return an error if they are part of a division or of a logical expression.

In SQL, null values are propagated in an expression differently from what happens with *BLANK* in DAX. As you can see in the previous examples, the presence of a *BLANK* in a DAX expression does not always result in a *BLANK* result, whereas the presence of *NULL* in SQL often evaluates to *NULL* for the entire expression. This difference is relevant whenever you use DirectQuery on top of a relational database because some calculations are executed in SQL and others are executed in DAX. The different semantics of *BLANK* in the two engines might result in unexpected behaviors.

Understanding the behavior of empty or missing values in a DAX expression and using *BLANK* to return an empty cell in a calculation are important skills to control the results of a DAX expression. One can often use *BLANK* as a result when detecting incorrect values or other errors, as we demonstrate in the next section.

Intercepting errors

Now that we have detailed the various kinds of errors that can occur, we still need to show you the techniques to intercept errors and correct them or, at least, produce an error message containing meaningful information. The presence of errors in a DAX expression frequently depends on the value of columns used in the expression itself. Therefore, one might want to control the presence of these error conditions and return an error message. The standard technique is to check whether an expression returns an error and, if so, replace the error with a specific message or a default value. There are a few DAX functions for this task.

The first of them is the *IFERROR* function, which is similar to the *IF* function, but instead of evaluating a Boolean condition, it checks whether an expression returns an error. Two typical uses of the *IFERROR* function are as follows:

```
= IFERROR ( Sales[Quantity] * Sales[Price], BLANK () )
= IFERROR ( SQRT ( Test[Omega] ), BLANK () )
```

In the first expression, if either *Sales[Quantity]* or *Sales[Price]* is a string that cannot be converted into a number, the returned expression is an empty value. Otherwise, the product of *Quantity* and *Price* is returned.

In the second expression, the result is an empty cell every time the *Test[Omega]* column contains a negative number.

Using *IFERROR* this way corresponds to a more general pattern that requires using *ISERROR* and *IF*:

```
= IF (
    ISERROR ( Sales[Quantity] * Sales[Price] ),
    BLANK (),
    Sales[Quantity] * Sales[Price]
)

= IF (
    ISERROR ( SQRT ( Test[Omega] ) ),
    BLANK (),
    SQRT ( Test[Omega] )
)
```

In these cases, *IFERROR* is a better option. One can use *IFERROR* whenever the result is the same expression tested for an error; there is no need to duplicate the expression in two places, and the code is safer and more readable. However, a developer should use *IF* when they want to return the result of a different expression.

Besides, one can avoid raising the error altogether by testing parameters before using them. For example, one can detect whether the argument for *SQRT* is positive, returning *BLANK* for negative values:

```
= IF (
    Test[Omega] >= 0,
    SQRT ( Test[Omega] ),
    BLANK ()
)
```

Considering that the third argument of an *IF* statement defaults to *BLANK*, one can also write the same expression more concisely:

```
= IF (
    Test[Omega] >= 0,
    SQRT ( Test[Omega] )
)
```

A frequent scenario is to test against empty values. *ISBLANK* detects empty values, returning *TRUE* if its argument is *BLANK*. This capability is important especially when a value being unavailable does not imply that it is 0. The following example calculates the cost of shipping for a sale transaction, using a default shipping cost for the product if the transaction itself does not specify a weight:

```
= IF (
   ISBLANK ( Sales[Weight] ),             -- If the weight is missing
   Sales[DefaultShippingCost],            -- then return the default cost
   Sales[Weight] * Sales[ShippingPrice]   -- otherwise multiply weight by shipping price
)
```

If we simply multiply product weight by shipping price, we get an empty cost for all the sales transactions without weight data because of the propagation of *BLANK* in multiplications.

When using variables, errors must be checked at the time of variable definition rather than where we use them. In fact, the first formula in the following code returns zero, the second formula always throws an error, and the last one produces different results depending on the version of the product using DAX (the latest version throws an error also):

```
IFERROR ( SQRT ( -1 ), 0 )                 -- This returns 0

VAR WrongValue = SQRT ( -1 )               -- Error happens here, so the result is
RETURN                                     -- always an error
    IFERROR ( WrongValue, 0 )              -- This line is never executed

IFERROR (                                  -- Different results depending on versions
    VAR WrongValue = SQRT ( -1 )           -- IFERROR throws an error in 2017 versions
    RETURN                                 -- IFERROR returns 0 in versions until 2016
        WrongValue,
    0
)
```

The error happens when *WrongValue* is evaluated. Thus, the engine will never execute the *IFERROR* function in the second example, whereas the outcome of the third example depends on product versions. If you need to check for errors, take some extra precautions when using variables.

Avoid using error-handling functions

Although we will cover optimizations later in the book, you need to be aware that error-handling functions might create severe performance issues in your code. It is not that they are slow in and of themselves. The problem is that the DAX engine cannot use optimized paths in its code when errors happen. In most cases, checking operands for possible errors is more efficient than using the error-handling engine. For example, instead of writing this:

```
IFERROR (
   SQRT ( Test[Omega] ),
   BLANK ()
)
```

It is much better to write this:

```
IF (
   Test[Omega] >= 0,
   SQRT ( Test[Omega] ),
   BLANK ()
)
```

This second expression does not need to detect the error and is faster than the previous expression. This, of course, is a general rule. For a detailed explanation, see Chapter 19, “Optimizing DAX.”

Another reason to avoid *IFERROR* is that it cannot intercept errors happening at a deeper level of execution. For example, the following code intercepts any error happening in the conversion of the *Table[Amount]* column considering a blank value in case *Amount* does not contain a number. As discussed previously, this execution is expensive because it is evaluated for every row in *Table*.

```
SUMX (
    Table,
    IFERROR ( VALUE ( Table[Amount] ), BLANK () )
)
```

Be mindful that, due to optimizations in the DAX engine, the following code does not intercept the same errors intercepted by the preceding example. If *Table[Amount]* contains a string that is not a number in just one row, the entire expression generates an error that is not intercepted by *IFERROR*.

```
IFERROR (
    SUMX (
        Table,
        VALUE ( Table[Amount] )
    ),
    BLANK ()
)
```

ISERROR has the same behavior as *IFERROR*. Be sure to use them carefully and only to intercept errors raised directly by the expression evaluated within *IFERROR/ISERROR* and not in nested calculations.

Generating errors

Sometimes, an error is just an error, and the formula should not return a default value in case of an error. Indeed, returning a default value would end up producing an actual result that would be incorrect. For example, a configuration table that contains inconsistent data should produce an invalid report rather than numbers that are unreliable, and yet it might be considered correct.

Moreover, instead of a generic error, one might want to produce an error message that is more meaningful to the users. Such a message would help users find where the problem is.

Consider a scenario that requires the computation of the square root of the absolute temperature measured in Kelvin, to approximately adjust the speed of sound in a complex scientific calculation. Obviously, we do not expect that temperature to be a negative number. If that happens due to a problem in the measurement, we need to raise an error and stop the calculation.

In that case, this code is dangerous because it hides the problem:

```
= IFERROR (
    SQRT ( Test[Temperature] ),
    0
)
```

Instead, to protect the calculations, one should write the formula like this:

```
= IF (
    Test[Temperature] >= 0,
    SQRT ( Test[Temperature] ),
    ERROR ( "The temperature cannot be a negative number. Calculation aborted." )
)
```

Formatting DAX code

Before we continue explaining the DAX language, we would like to cover an important aspect of DAX—that is, formatting the code. DAX is a functional language, meaning that no matter how complex it is, a DAX expression is like a single function call. The complexity of the code translates into the complexity of the expressions that one uses as parameters for the outermost function.

For this reason, it is normal to see expressions that span over 10 lines or more. Seeing a 20-line DAX expression is common, so you will become acquainted with it. Nevertheless, as formulas start to grow in length and complexity, it is extremely important to format the code to make it human-readable.

There is no "official" standard to format DAX code, yet we believe it is important to describe the standard that we use in our code. It is likely not the perfect standard, and you might prefer something different. We have no problem with that: find your optimal standard and use it. The only thing you need to remember is: *format your code and never write everything on a single line; otherwise, you will be in trouble sooner than you expect.*

To understand why formatting is important, look at a formula that computes a time intelligence calculation. This somewhat complex formula is still not the most complex you will write. Here is how the expression looks if you do not format it in some way:

```
IF(CALCULATE(NOT ISEMPTY(Balances), ALLEXCEPT (Balances, BalanceDate)),SUMX (ALL(Balances
[Account]), CALCULATE(SUM (Balances[Balance]),LASTNONBLANK(DATESBETWEEN(BalanceDate[Date],
BLANK(),MAX(BalanceDate[Date])),CALCULATE(COUNTROWS(Balances))))),BLANK())
```

Trying to understand what this formula computes in its present form is nearly impossible. There is no clue which is the outermost function and how DAX evaluates the different parameters to create the complete flow of execution. We have seen too many examples of formulas written this way by students who, at some point, ask for help in understanding why the formula returns incorrect results. Guess what? The first thing we do is format the expression; only later do we start working on it.

The same expression, properly formatted, looks like this:

```
IF (
    CALCULATE (
        NOT ISEMPTY ( Balances ),
        ALLEXCEPT (
            Balances,
            BalanceDate
        )
    ),
    SUMX (
        ALL ( Balances[Account] ),
        CALCULATE (
            SUM ( Balances[Balance] ),
            LASTNONBLANK (
                DATESBETWEEN (
                    BalanceDate[Date],
                    BLANK (),
                    MAX ( BalanceDate[Date] )
                ),
                CALCULATE (
                    COUNTROWS ( Balances )
                )
            )
        )
    ),
    BLANK ()
)
```

The code is the same, but this time it is much easier to see the three parameters of *IF*. Most important, it is easier to follow the blocks that arise naturally from indenting lines and how they compose the complete flow of execution. The code is still hard to read, but now the problem is DAX, not poor formatting. A more verbose syntax using variables can help you read the code, but even in this case, the formatting is important in providing a correct understanding of the scope of each variable:

```
IF (
    CALCULATE (
        NOT ISEMPTY ( Balances ),
        ALLEXCEPT (
            Balances,
            BalanceDate
        )
    ),
    SUMX (
        ALL ( Balances[Account] ),
        VAR PreviousDates =
            DATESBETWEEN (
                BalanceDate[Date],
                BLANK (),
                MAX ( BalanceDate[Date] )
            )
        VAR LastDateWithBalance =
            LASTNONBLANK (
                PreviousDates,
                CALCULATE (
                    COUNTROWS ( Balances )
                )
            )
        RETURN
            CALCULATE (
                SUM ( Balances[Balance] ),
                LastDateWithBalance
            )
    ),
    BLANK ()
)
```

DAXFormatter.com

We created a website dedicated to formatting DAX code. We created this site for ourselves because formatting code is a time-consuming operation and we did not want to spend our time doing it for every formula we write. After the tool was working, we decided to donate it to the public domain so that users can format their own DAX code (by the way, we have been able to promote our formatting rules this way).

You can find the website at www.daxformatter.com. The user interface is simple: just copy your DAX code, click FORMAT, and the page refreshes showing a nicely formatted version of your code, which you can then copy and paste in the original window.

This is the set of rules that we use to format DAX:

- Always separate function names such as *IF*, *SUMX*, and *CALCULATE* from any other term using a space and always write them in uppercase.
- Write all column references in the form *TableName[ColumnName]*, with no space between the table name and the opening square bracket. Always include the table name.
- Write all measure references in the form *[MeasureName]*, without any table name.
- Always use a space following commas and never precede them with a space.
- If the formula fits one single line, do not apply any other rule.
- If the formula does not fit a single line, then
    - Place the function name on a line by itself, with the opening parenthesis.
    - Keep all parameters on separate lines, indented with four spaces and with the comma at the end of the expression except for the last parameter.
    - Align the closing parenthesis with the function call so that the closing parenthesis stands on its own line.

These are the basic rules we use. A more detailed list of these rules is available at http://sql.bi/daxrules.

If you find a way to express formulas that best fits your reading method, use it. The goal of formatting is to make the formula easier to read, so use the technique that works best for you. The most important point to remember when defining your personal set of formatting rules is that you always need to be able to see errors as soon as possible. If, in the unformatted code shown previously, DAX complained about a missing closing parenthesis, it would be hard to spot where the error is. In the formatted formula, it is much easier to see how each closing parenthesis matches the opening function call.

Help on formatting DAX

Formatting DAX is not an easy task because often we write it using a small font in a text box. Depending on the version, Power BI, Excel, and Visual Studio provide different text editors for DAX. Nevertheless, a few hints might help in writing DAX code:

- To increase the font size, hold down Ctrl while rotating the wheel button on the mouse, making it easier to look at the code.
- To add a new line to the formula, press Shift+Enter.
- If editing in the text box is not for you, copy the code into another editor, such as Notepad or DAX Studio, and then copy and paste the formula back into the text box.

When you look at a DAX expression, at first glance it may be hard to understand whether it is a calculated column or a measure. Thus, in our books and articles we use an equal sign (=) whenever we define a calculated column and the assignment operator (:=) to define measures:

```
CalcCol = SUM ( Sales[SalesAmount] )        -- is a calculated column
Store[CalcCol] = SUM ( Sales[SalesAmount] )  -- is a calculated column in Store table
CalcMsr := SUM ( Sales[SalesAmount]  )      -- is a measure
```

Finally, when using columns and measures in code, we recommend to always put a table name before a column and never before a measure, as we do in every example.

Introducing aggregators and iterators

Almost every data model needs to operate on aggregated data. DAX offers a set of functions that aggregate the values of a column in a table and return a single value. We call this group of functions *aggregation functions*. For example, the following measure calculates the sum of all the numbers in the *SalesAmount* column of the *Sales* table:

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

SUM aggregates all the rows of the table if it is used in a calculated column. Whenever it is used in a measure, it considers only the rows that are being filtered by slicers, rows, columns, and filter conditions in the report.

There are many aggregation functions (*SUM*, *AVERAGE*, *MIN*, *MAX*, and *STDEV*), and their behavior changes only in the way they aggregate values: *SUM* adds values, whereas *MIN* returns the minimum value. Nearly all these functions operate only on numeric values or on dates. Only *MIN* and *MAX* can operate on text values also. Moreover, DAX never considers empty cells when it performs the aggregation, and this behavior is different from their counterpart in Excel (more on this later in this chapter).

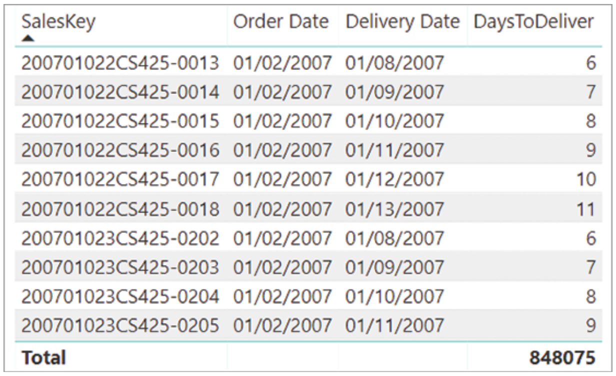
Note *MIN* and *MAX* offer another behavior: if used with two parameters, they return the minimum or maximum of the two parameters. Thus, *MIN* (1, 2) returns 1 and *MAX* (1, 2) returns 2. This functionality is useful when one needs to compute the minimum or maximum of complex expressions because it saves having to write the same expression multiple times in *IF* statements.

All the aggregation functions we have described so far work on columns. Therefore, they aggregate values from a single column only. Some aggregation functions can aggregate an expression instead of a single column. Because of the way they work, they are known as *iterators*. This set of functions is useful, especially when you need to make calculations using columns of different related tables, or when you need to reduce the number of calculated columns.

Iterators always accept at least two parameters: the first is a table that they scan; the second is typically an expression that is evaluated for each row of the table. After they have completed scanning the table and evaluating the expression row by row, iterators aggregate the partial results according to their semantics.

For example, if we compute the number of days needed to deliver an order in a calculated column called *DaysToDeliver* and build a report on top of that, we obtain the report shown in Figure 2-6. Note that the grand total shows the sum of all the days, which is not useful for this metric:

```
Sales[DaysToDeliver] = INT ( Sales[Delivery Date] - Sales[Order Date] )
```



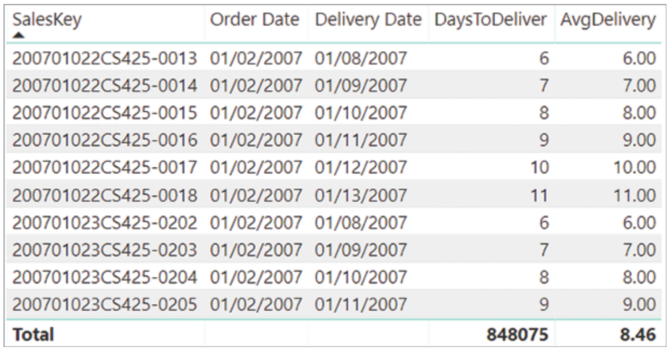
SalesKey	Order Date	Delivery Date	DaysToDeliver
200701022CS425-0013	01/02/2007	01/08/2007	6
200701022CS425-0014	01/02/2007	01/09/2007	7
200701022CS425-0015	01/02/2007	01/10/2007	8
200701022CS425-0016	01/02/2007	01/11/2007	9
200701022CS425-0017	01/02/2007	01/12/2007	10
200701022CS425-0018	01/02/2007	01/13/2007	11
200701023CS425-0202	01/02/2007	01/08/2007	6
200701023CS425-0203	01/02/2007	01/09/2007	7
200701023CS425-0204	01/02/2007	01/10/2007	8
200701023CS425-0205	01/02/2007	01/11/2007	9
Total			**848075**

FIGURE 2-6 The grand total is shown as a sum, when you might want an average instead.

A grand total that we can actually use requires a measure called *AvgDelivery* showing the delivery time for each order and the average of all the durations at the grand total level:

```
AvgDelivery := AVERAGE ( Sales[DaysToDeliver] )
```

The result of this new measure is visible in the report shown in Figure 2-7.



SalesKey	Order Date	Delivery Date	DaysToDeliver	AvgDelivery
200701022CS425-0013	01/02/2007	01/08/2007	6	6.00
200701022CS425-0014	01/02/2007	01/09/2007	7	7.00
200701022CS425-0015	01/02/2007	01/10/2007	8	8.00
200701022CS425-0016	01/02/2007	01/11/2007	9	9.00
200701022CS425-0017	01/02/2007	01/12/2007	10	10.00
200701022CS425-0018	01/02/2007	01/13/2007	11	11.00
200701023CS425-0202	01/02/2007	01/08/2007	6	6.00
200701023CS425-0203	01/02/2007	01/09/2007	7	7.00
200701023CS425-0204	01/02/2007	01/10/2007	8	8.00
200701023CS425-0205	01/02/2007	01/11/2007	9	9.00
Total			**848075**	**8.46**

FIGURE 2-7 The measure aggregating by average shows the average delivery days at the grand total level.

The measure computes the average value by averaging a calculated column. One could remove the calculated column, thus saving space in the model, by leveraging an iterator. Indeed, although it is true that *AVERAGE* cannot average an expression, its counterpart *AVERAGEX* can iterate the *Sales* table and compute the delivery days row by row, averaging the results at the end. This code accomplishes the same result as the previous definition:

```
AvgDelivery :=
AVERAGEX (
    Sales,
    INT ( Sales[Delivery Date] - Sales[Order Date] )
)
```

The biggest advantage of this last expression is that it does not rely on the presence of a calculated column. Thus, we can build the entire report without creating expensive calculated columns.

Most iterators have the same name as their noniterative counterpart. For example, *SUM* has a corresponding *SUMX*, and *MIN* has a corresponding *MINX*. Nevertheless, keep in mind that some iterators do not correspond to any aggregator. Later in this book, you will learn about *FILTER*, *ADDCOLUMNS*, *GENERATE*, and other functions that are iterators even if they do not aggregate their results.

When you first learn DAX, you might think that iterators are inherently slow. The concept of performing calculations row by row looks like a CPU-intensive operation. Actually, iterators are fast, and no performance penalty is caused by using iterators instead of standard aggregators. Aggregators are just a syntax-sugared version of iterators.

Indeed, the basic aggregation functions are a shortened version of the corresponding X-suffixed function. For example, consider the following expression:

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

It is internally translated into this corresponding version of the same code:

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

The only advantage in using *SUM* is a shorter syntax. However, there are no differences in performance between *SUM* and *SUMX* aggregating a single column. They are in all respects the same function.

We will cover more details about this behavior in Chapter 4. There we introduce the concept of evaluation contexts to describe properly how iterators work.

Using common DAX functions

Now that you have seen the fundamentals of DAX and how to handle error conditions, what follows is a brief tour through the most commonly used functions and expressions of DAX.

Aggregation functions

In the previous sections, we described the basic aggregators like *SUM*, *AVERAGE*, *MIN*, and *MAX*. You learned that *SUM* and *AVERAGE*, for example, work only on numeric columns.

DAX also offers an alternative syntax for aggregation functions inherited from Excel, which adds the suffix A to the name of the function, just to get the same name and behavior as Excel. However, these functions are useful only for columns containing *Boolean* values because *TRUE* is evaluated as 1 and *FALSE* as 0. Text columns are always considered 0. Therefore, no matter what is in the content of a column, if one uses *MAXA* on a text column, the result will always be a 0. Moreover, DAX never considers empty cells when it performs the aggregation. Although these functions can be used on nonnumeric columns without retuning an error, their results are not useful because there is no automatic conversion to numbers for text columns. These functions are named *AVERAGEA*, *COUNTA*, *MINA*, and *MAXA*. We suggest that you do not use these functions, whose behavior will be kept unchanged in the future because of compatibility with existing code that might rely on current behavior.

Note Despite the names being identical to statistical functions, they are used differently in DAX and Excel because in DAX a column has a data type, and its data type determines the behavior of aggregation functions. Excel handles a different data type for each cell, whereas DAX handles a single data type for the entire column. DAX deals with data in tabular form with well-defined types for each column, whereas Excel formulas work on heterogeneous cell values without well-defined types. If a column in Power BI has a numeric data type, all the values can be only numbers or empty cells. If a column is of a text type, it is always 0 for these functions (except for *COUNTA*), even if the text can be converted into a number, whereas in Excel the value is considered a number on a cell-by-cell basis. For these reasons, these functions are not very useful for Text columns. Only *MIN* and *MAX* also support text values in DAX.

The functions you learned earlier are useful to perform the aggregation of values. Sometimes, you might not be interested in aggregating values but only in counting them. DAX offers a set of functions that are useful to count rows or values:

- *COUNT* operates on any data type, apart from *Boolean*.
- *COUNTA* operates on any type of column.
- *COUNTBLANK* returns the number of empty cells (blanks or empty strings) in a column.
- *COUNTROWS* returns the number of rows in a table.
- *DISTINCTCOUNT* returns the number of distinct values of a column, blank value included if present.
- *DISTINCTCOUNTNOBLANK* returns the number of distinct values of a column, no blank value included.

COUNT and *COUNTA* are nearly identical functions in DAX. They return the number of values of the column that are not empty, regardless of their data type. They are inherited from Excel, where *COUNTA* accepts any data type including strings, whereas *COUNT* accepts only numeric columns. If we want to count all the values in a column that contain an empty value, you can use the *COUNTBLANK* function. Both blanks and empty values are considered empty values by *COUNTBLANK*. Finally, if we want to count the number of rows of a table, you can use the *COUNTROWS* function. Beware that *COUNT-ROWS* requires a table as a parameter, not a column.

The last two functions, *DISTINCTCOUNT* and *DISTINCTCOUNTNOBLANK*, are useful because they do exactly what their names suggest: count the distinct values of a column, which it takes as its only parameter. *DISTINCTCOUNT* counts the *BLANK* value as one of the possible values, whereas *DISTINCT-COUNTNOBLANK* ignores the *BLANK* value.

Note *DISTINCTCOUNT* is a function introduced in the 2012 version of DAX. The earlier versions of DAX did not include *DISTINCTCOUNT*; to compute the number of distinct values of a column, we had to use *COUNTROWS (DISTINCT (table[column]))*. The two patterns return the same result although *DISTINCTCOUNT* is easier to read, requiring only a single function call. *DISTINCTCOUNTNOBLANK* is a function introduced in 2019 and it provides the same semantic of a *COUNT DISTINCT* operation in SQL without having to write a longer expression in DAX.

Logical functions

Sometimes we want to build a logical condition in an expression—for example, to implement different calculations depending on the value of a column or to intercept an error condition. In these cases, we can use one of the logical functions in DAX. The earlier section titled "Handling errors in DAX expressions" described the two most important functions of this group: *IF* and *IFERROR*. We described the *IF* function in the "Conditional statements" section, earlier in this chapter.

Logical functions are very simple and do what their names suggest. They are *AND*, *FALSE*, *IF*, *IFERROR*, *NOT*, *TRUE*, and *OR*. For example, if we want to compute the amount as quantity multiplied by price only when the *Price* column contains a numeric value, we can use the following pattern:

```
Sales[Amount] = IFERROR ( Sales[Quantity] * Sales[Price], BLANK ( ) )
```

If we did not use *IFERROR* and if the *Price* column contained an invalid number, the result for the calculated column would be an error because if a single row generates a calculation error, the error propagates to the whole column. The use of *IFERROR*, however, intercepts the error and replaces it with a blank value.

Another interesting function in this category is *SWITCH*, which is useful when we have a column containing a low number of distinct values, and we want to get different behaviors depending on its value. For example, the column *Size* in the *Product* table contains S, M, L, XL, and we might want to decode this value in a more explicit column. We can obtain the result by using nested *IF* calls:

```
'Product'[SizeDesc] =
IF (
    'Product'[Size] = "S",
    "Small",
    IF (
        'Product'[Size] = "M",
        "Medium",
        IF (
            'Product'[Size] = "L",
            "Large",
            IF (
                'Product'[Size] = "XL",
                "Extra Large",
                "Other"
            )
        )
    )
)
```

A more convenient way to express the same formula, using *SWITCH*, is like this:

```
'Product'[SizeDesc] =
SWITCH (
    'Product'[Size],
    "S", "Small",
    "M", "Medium",
    "L", "Large",
    "XL", "Extra Large",
    "Other"
)
```

The code in this latter expression is more readable, though not faster, because internally DAX translates *SWITCH* statements into a set of nested *IF* functions.

Note *SWITCH* is often used to check the value of a parameter and define the result of a measure. For example, one might create a parameter table containing *YTD*, *MTD*, *QTD* as three rows and let the user choose from the three available which aggregation to use in a measure. This was a common scenario before 2019. Now it is no longer needed thanks to the introduction of calculation groups, covered in Chapter 9, "Calculation groups." Calculation groups are the preferred way of computing values that the user can parameterize.

Tip Here is an interesting way to use the *SWITCH* function to check for multiple conditions in the same expression. Because *SWITCH* is converted into a set of nested *IF* functions, where the first one that matches wins, you can test multiple conditions using this pattern:

```
SWITCH (
   TRUE (),
   Product[Size] = "XL" && Product[Color] = "Red", "Red and XL",
   Product[Size] = "XL" && Product[Color] = "Blue", "Blue and XL",
   Product[Size] = "L" && Product[Color] = "Green", "Green and L"
)
```

Using *TRUE* as the first parameter means, "Return the first result where the condition evaluates to *TRUE*."

Information functions

Whenever there is the need to analyze the type of an expression, you can use one of the information functions. All these functions return a *Boolean* value and can be used in any logical expression. They are *ISBLANK*, *ISERROR*, *ISLOGICAL*, *ISNONTEXT*, *ISNUMBER*, and *ISTEXT*.

It is important to note that when a column is passed as a parameter instead of an expression, the functions *ISNUMBER*, *ISTEXT*, and *ISNONTEXT* always return *TRUE* or *FALSE* depending on the data type of the column and on the empty condition of each cell. This makes these functions nearly useless in DAX; they have been inherited from Excel in the first DAX version.

You might be wondering whether you can use *ISNUMBER* with a text column just to check whether a conversion to a number is possible. Unfortunately, this approach is not possible. If you want to test whether a text value is convertible to a number, you must try the conversion and handle the error if it fails. For example, to test whether the column *Price* (which is of type *string*) contains a valid number, one must write

```
Sales[IsPriceCorrect] = NOT ISERROR ( VALUE ( Sales[Price] ) )
```

DAX tries to convert from a string value to a number. If it succeeds, it returns *TRUE* (because *ISERROR* returns *FALSE*); otherwise, it returns *FALSE* (because *ISERROR* returns *TRUE*). For example, the conversion fails if some of the rows have an "N/A" string value for price.

However, if we try to use *ISNUMBER*, as in the following expression, we always receive *FALSE* as a result:

```
Sales[IsPriceCorrect] = ISNUMBER ( Sales[Price] )
```

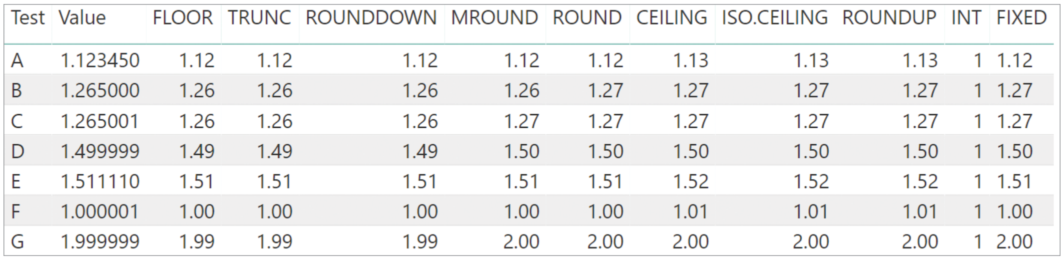
In this case, *ISNUMBER* always returns *FALSE* because, based on the definition in the model, the *Price* column is not a number but a string, regardless of the content of each row.

Mathematical functions

The set of mathematical functions available in DAX is similar to the set available in Excel, with the same syntax and behavior. The mathematical functions of common use are *ABS*, *EXP*, *FACT*, *LN*, *LOG*, *LOG10*, *MOD*, *PI*, *POWER*, *QUOTIENT*, *SIGN*, and *SQRT*. Random functions are *RAND* and *RANDBETWEEN*. By using *EVEN* and *ODD*, you can test numbers. *GCD* and *LCM* are useful to compute the greatest common denominator and least common multiple of two numbers. *QUOTIENT* returns the integer division of two numbers.

Finally, several rounding functions deserve an example; in fact, we might use several approaches to get the same result. Consider these calculated columns, along with their results in Figure 2-8:

```
FLOOR = FLOOR ( Tests[Value], 0.01 )
TRUNC = TRUNC ( Tests[Value], 2 )
ROUNDDOWN = ROUNDDOWN ( Tests[Value], 2 )
MROUND = MROUND ( Tests[Value], 0.01 )
ROUND = ROUND ( Tests[Value], 2 )
CEILING = CEILING ( Tests[Value], 0.01 )
ISO.CEILING = ISO.CEILING ( Tests[Value], 0.01 )
ROUNDUP = ROUNDUP ( Tests[Value], 2 )
INT = INT ( Tests[Value] )
FIXED = FIXED ( Tests[Value], 2, TRUE )
```



Test	Value	FLOOR	TRUNC	ROUNDDOWN	MROUND	ROUND	CEILING	ISO.CEILING	ROUNDUP	INT	FIXED
A	1.123450	1.12	1.12	1.12	1.12	1.12	1.13	1.13	1.13	1	1.12
B	1.265000	1.26	1.26	1.26	1.26	1.27	1.27	1.27	1.27	1	1.27
C	1.265001	1.26	1.26	1.26	1.27	1.27	1.27	1.27	1.27	1	1.27
D	1.499999	1.49	1.49	1.49	1.50	1.50	1.50	1.50	1.50	1	1.50
E	1.511110	1.51	1.51	1.51	1.51	1.51	1.52	1.52	1.52	1	1.51
F	1.000001	1.00	1.00	1.00	1.00	1.00	1.01	1.01	1.01	1	1.00
G	1.999999	1.99	1.99	1.99	2.00	2.00	2.00	2.00	2.00	1	2.00

FIGURE 2-8 This summary shows the results of using different rounding functions.

FLOOR, *TRUNC*, and *ROUNDDOWN* are similar except in the way we can specify the number of digits to round. In the opposite direction, *CEILING* and *ROUNDUP* are similar in their results. You can see a few differences in the way the rounding is done between *MROUND* and *ROUND* function.

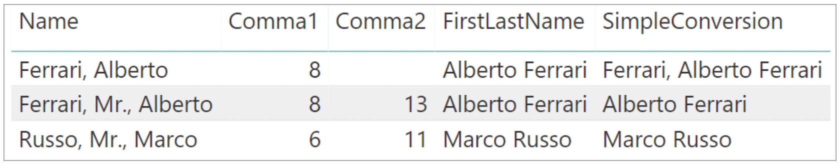
Trigonometric functions

DAX offers a rich set of trigonometric functions that are useful for certain calculations: *COS*, *COSH*, *COT*, *COTH*, *SIN*, *SINH*, *TAN*, and *TANH*. Prefixing them with A computes the arc version (arcsine, arccosine, and so on). We do not go into the details of these functions because their use is straightforward.

DEGREES and *RADIANS* perform conversion to degrees and radians, respectively, and *SQRTPI* computes the square root of its parameter after multiplying it by pi.

Text functions

Most of the text functions available in DAX are similar to those available in Excel, with only a few exceptions. The text functions are *CONCATENATE*, *CONCATENATEX*, *EXACT*, *FIND*, *FIXED*, *FORMAT*, *LEFT*, *LEN*, *LOWER*, *MID*, *REPLACE*, *REPT*, *RIGHT*, *SEARCH*, *SUBSTITUTE*, *TRIM*, *UPPER*, and *VALUE*. These functions are useful for manipulating text and extracting data from strings that contain multiple values. For example, Figure 2-9 shows an example of the extraction of first and last names from a string that contains these values separated by commas, with the title in the middle that we want to remove.



Name	Comma1	Comma2	FirstLastName	SimpleConversion
Ferrari, Alberto	8		Alberto Ferrari	Ferrari, Alberto Ferrari
Ferrari, Mr., Alberto	8	13	Alberto Ferrari	Alberto Ferrari
Russo, Mr., Marco	6	11	Marco Russo	Marco Russo

FIGURE 2-9 This example shows first and last names extracted using text functions.

To achieve this result, you start calculating the position of the two commas. Then we use these numbers to extract the right part of the text. The *SimpleConversion* column implements a formula that might return inaccurate values if there are fewer than two commas in the string, and it raises an error if there are no commas at all. The *FirstLastName* column implements a more complex expression that does not fail in case of missing commas:

```
People[Comma1] = IFERROR ( FIND ( ",", People[Name] ), BLANK ( ) )
People[Comma2] = IFERROR ( FIND ( " ,", People[Name], People[Comma1] + 1 ), BLANK ( ) )
People[SimpleConversion] =
MID ( People[Name], People[Comma2] + 1, LEN ( People[Name] ) )
    & " "
    & LEFT ( People[Name], People[Comma1] - 1 )
People[FirstLastName] =
TRIM (
    MID (
        People[Name],
        IF ( ISNUMBER ( People[Comma2] ), People[Comma2], People[Comma1] ) + 1,
        LEN ( People[Name] )
    )
)
    & IF (
```