Microsoft

The Definitive Guide to DAX

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

SECOND EDITION

Marco Russo and Alberto Ferrari

Sample files on the web

Sample pages

Contents

Foreword . . . xvii
Introduction to the second edition . . . xx
Introduction to the first edition . . . xxi

Chapter 1 What is DAX? 1

Understanding the data model . . . 1
Understanding the direction of a relationship . . . 3
DAX for Excel users . . . 5
Cells versus tables . . . 5
Excel and DAX: Two functional languages . . . 7
Iterators in DAX . . . 8
DAX requires theory . . . 8
DAX for SQL developers . . . 9
Relationship handling . . . 9
DAX is a functional language . . . 10
DAX as a programming and querying language . . . 10
Subqueries and conditions in DAX and SQL . . . 11
DAX for MDX developers . . . 12
Multidimensional versus Tabular . . . 12
DAX as a programming and querying language . . . 12
Hierarchies . . . 13
Leaf-level calculations . . . 14
DAX for Power BI users . . . 14

Chapter 2 Introducing DAX 17

Understanding DAX calculations . . . 17
DAX data types . . . 19
DAX operators . . . 23
Table constructors . . . 24
Conditional statements . . . 24

Sample pages

Understanding calculated columns and measures 25
Calculated columns 25
Measures 26
Introducing variables 30
Handling errors in DAX expressions 31
Conversion errors. 31
Arithmetic operations errors 32
Intercepting errors. 35
Generating errors. 38
Formatting DAX code. 39
Introducing aggregators and iterators 42
Using common DAX functions 45
Aggregation functions 45
Logical functions 46
Information functions. 48
Mathematical functions 49
Trigonometric functions 50
Text functions 50
Conversion functions 51
Date and time functions 52
Relational functions 53
Conclusions 55

Chapter 3 Using basic table functions 57
Introducing table functions 57
Introducing ***EVALUATE*** syntax 59
Understanding ***FILTER***. 61
Introducing ***ALL*** and ***ALLEXCEPT*** 63
Understanding ***VALUES***, ***DISTINCT***, and the blank row. 68
Using tables as scalar values. 72
Introducing ***ALLSELECTED*** 75
Conclusions 77

Sample pages

Chapter 4 Understanding evaluation contexts 79

Introducing evaluation contexts 80
    Understanding filter contexts 80
    Understanding the row context 85
Testing your understanding of evaluation contexts 88
    Using ***SUM*** in a calculated column 88
    Using columns in a measure 89
Using the row context with iterators 90
    Nested row contexts on different tables 91
    Nested row contexts on the same table 92
    Using the ***EARLIER*** function 97
Understanding ***FILTER***, ***ALL***, and context interactions 98
Working with several tables 101
    Row contexts and relationships 102
    Filter context and relationships 106
Using ***DISTINCT*** and ***SUMMARIZE*** in filter contexts 109
Conclusions 113

Chapter 5 Understanding *CALCULATE* and *CALCULATETABLE* 115

Introducing ***CALCULATE*** and ***CALCULATETABLE*** 115
    Creating filter contexts 115
    Introducing ***CALCULATE*** 119
    Using ***CALCULATE*** to compute percentages 124
    Introducing ***KEEPFILTERS*** 135
    Filtering a single column 138
    Filtering with complex conditions 140
    Evaluation order in ***CALCULATE*** 144
Understanding context transition 148
    Row context and filter context recap 148
    Introducing context transition 151
    Context transition in calculated columns 154
    Context transition with measures 157

Sample pages

Understanding circular dependencies . . . 161
CALCULATE modifiers. . . . 164
Understanding ***USERELATIONSHIP*** . . . 164
Understanding ***CROSSFILTER*** . . . 168
Understanding ***KEEPFILTERS*** . . . 168
Understanding ***ALL*** in ***CALCULATE*** . . . 169
Introducing ***ALL*** and ***ALLSELECTED*** with no parameters . . . 171
CALCULATE rules . . . 172

Chapter 6 Variables 175
Introducing ***VAR*** syntax. . . . 175
Understanding that variables are constant . . . 177
Understanding the scope of variables. . . . 178
Using table variables. . . . 181
Understanding lazy evaluation . . . 182
Common patterns using variables . . . 183
Conclusions . . . 185

Chapter 7 Working with iterators and with *CALCULATE* 187
Using iterators. . . . 187
Understanding iterator cardinality. . . . 188
Leveraging context transition in iterators . . . 190
Using ***CONCATENATEX***. . . . 194
Iterators returning tables. . . . 196
Solving common scenarios with iterators . . . 199
Computing averages and moving averages . . . 199
Using ***RANKX*** . . . 203
Changing calculation granularity . . . 211
Conclusions . . . 215

Chapter 8 Time intelligence calculations 217
Introducing time intelligence. . . . 217
Automatic Date/Time in Power BI . . . 218
Automatic date columns in Power Pivot for Excel . . . 219
Date table template in Power Pivot for Excel . . . 220

Sample pages

Building a date table 220
Using ***CALENDAR*** and ***CALENDARAUTO*** 222
Working with multiple dates 224
Handling multiple relationships to the ***Date*** table 224
Handling multiple date tables 226
Understanding basic time intelligence calculations 228
Using Mark as Date Table 232
Introducing basic time intelligence functions 233
Using year-to-date, quarter-to-date, and month-to-date 235
Computing time periods from prior periods 237
Mixing time intelligence functions 239
Computing a difference over previous periods 241
Computing a moving annual total 243
Using the right call order for nested time intelligence functions 245
Understanding semi-additive calculations 246
Using ***LASTDATE*** and ***LASTNONBLANK*** 248
Working with opening and closing balances 254
Understanding advanced time intelligence calculations 258
Understanding periods to date 259
Understanding ***DATEADD*** 262
Understanding ***FIRSTDATE, LASTDATE, FIRSTNONBLANK,*** and ***LASTNONBLANK*** 269
Using drillthrough with time intelligence 271
Working with custom calendars 272
Working with weeks 272
Custom year-to-date, quarter-to-date, and month-to-date 276
Conclusions 277

Chapter 9 Calculation groups 279

Introducing calculation groups 279
Creating calculation groups 281
Understanding calculation groups 288
Understanding calculation item application 291
Understanding calculation group precedence 299
Including and excluding measures from calculation items 304

Sample pages

Understanding sideways recursion 306
Using the best practices. 311
Conclusions 311

Chapter 10 Working with the filter context 313

Using ***HASONEVALUE*** and ***SELECTEDVALUE*** 314
Introducing ***ISFILTERED*** and ***ISCROSSFILTERED*** 319
Understanding differences between ***VALUES*** and ***FILTERS*** 322
Understanding the difference between ***ALLEXCEPT*** and ***ALL***/***VALUES*** 324
Using ***ALL*** to avoid context transition. 328
Using ***ISEMPTY*** 330
Introducing data lineage and ***TREATAS*** 332
Understanding arbitrarily shaped filters. 336
Conclusions 343

Chapter 11 Handling hierarchies 345

Computing percentages over hierarchies 345
Handling parent/child hierarchies 350
Conclusions 362

Chapter 12 Working with tables 363

Using ***CALCULATETABLE*** 363
Manipulating tables 365
  Using ***ADDCOLUMNS*** 366
  Using ***SUMMARIZE*** 369
  Using ***CROSSJOIN*** 372
  Using ***UNION*** 374
  Using ***INTERSECT*** 378
  Using ***EXCEPT*** 379
Using tables as filters 381
  Implementing ***OR*** conditions. 381
  Narrowing sales computation to the first year's customers. 384

Sample pages

Computing new customers . . . 386
Reusing table expressions with ***DETAILROWS*** . . . 388
Creating calculated tables . . . 390
Using ***SELECTCOLUMNS*** . . . 390
Creating static tables with ***ROW*** . . . 391
Creating static tables with ***DATATABLE*** . . . 392
Using ***GENERATESERIES*** . . . 393
Conclusions . . . 394

Chapter 13 Authoring queries 395

Introducing DAX Studio . . . 395
Understanding ***EVALUATE*** . . . 396
Introducing the ***EVALUATE*** syntax . . . 396
Using ***VAR*** in ***DEFINE*** . . . 397
Using ***MEASURE*** in ***DEFINE*** . . . 399
Implementing common DAX query patterns . . . 400
Using ***ROW*** to test measures . . . 400
Using ***SUMMARIZE*** . . . 401
Using ***SUMMARIZECOLUMNS*** . . . 403
Using ***TOPN*** . . . 409
Using ***GENERATE*** and ***GENERATEALL*** . . . 415
Using ***ISONORAFTER*** . . . 418
Using ***ADDMISSINGITEMS*** . . . 420
Using ***TOPNSKIP*** . . . 421
Using ***GROUPBY*** . . . 421
Using ***NATURALINNERJOIN*** and ***NATURALLEFTOUTERJOIN*** . . . 424
Using ***SUBSTITUTEWITHINDEX*** . . . 426
Using ***SAMPLE*** . . . 428
Understanding the auto-exists behavior in DAX queries . . . 429
Conclusions . . . 435

Chapter 14 Advanced DAX concepts 437

Introducing expanded tables . . . 437
Understanding ***RELATED*** . . . 441
Using ***RELATED*** in calculated columns . . . 443

Sample pages

Understanding the difference between table filters and column filters. . . . 444
Using table filters in measures 447
Understanding active relationships 451
Difference between table expansion and filtering 453
Context transition in expanded tables 455
Understanding ***ALLSELECTED*** and shadow filter contexts 456
Introducing shadow filter contexts. 457
ALLSELECTED returns the iterated rows 461
ALLSELECTED without parameters 463
The ***ALL**** family of functions 463
ALL 465
ALLEXCEPT 466
ALLNOBLANKROW 466
ALLSELECTED 466
ALLCROSSFILTERED 466
Understanding data lineage. 466
Conclusions 469

Chapter 15 Advanced relationships 471

Implementing calculated physical relationships 471
Computing multiple-column relationships 471
Implementing relationships based on ranges. 474
Understanding circular dependency in calculated physical relationships 476
Implementing virtual relationships 480
Transferring filters in DAX 480
Transferring a filter using ***TREATAS*** 482
Transferring a filter using ***INTERSECT*** 483
Transferring a filter using ***FILTER***. 484
Implementing dynamic segmentation using virtual relationships. 485
Understanding physical relationships in DAX. 488
Using bidirectional cross-filters. 491

Sample pages

Understanding one-to-many relationships. **493**
Understanding one-to-one relationships **493**
Understanding many-to-many relationships **494**
Implementing many-to-many using a bridge table **494**
Implementing many-to-many using a common dimension **500**
Implementing many-to-many using MMR weak relationships **504**
Choosing the right type of relationships **506**
Managing granularities **507**
Managing ambiguity in relationships **512**
Understanding ambiguity in active relationships **514**
Solving ambiguity in non-active relationships **515**
Conclusions **517**

Chapter 16 Advanced calculations in DAX 519

Computing the working days between two dates **519**
Showing budget and sales together **527**
Computing same-store sales **530**
Numbering sequences of events **536**
Computing previous year sales up to last date of sales **539**
Conclusions **544**

Chapter 17 The DAX engines 545

Understanding the architecture of the DAX engines **545**
Introducing the formula engine **547**
Introducing the storage engine. **547**
Introducing the VertiPaq (in-memory) storage engine. **548**
Introducing the DirectQuery storage engine **549**
Understanding data refresh **549**
Understanding the VertiPaq storage engine **550**
Introducing columnar databases **550**
Understanding VertiPaq compression **553**
Understanding segmentation and partitioning. **562**
Using Dynamic Management Views **563**

Sample pages

Understanding the use of relationships in VertiPaq 565
Introducing materialization 568
Introducing aggregations 571
Choosing hardware for VertiPaq 573
    Hardware choice as an option 573
    Set hardware priorities 574
    CPU model 574
    Memory speed 575
    Number of cores 576
    Memory size 576
    Disk I/O and paging 576
    Best practices in hardware selection 577
Conclusions 577

Chapter 18 Optimizing VertiPaq 579

Gathering information about the data model 579
Denormalization 584
Columns cardinality 591
Handling date and time 592
Calculated columns 595
    Optimizing complex filters with *Boolean* calculated columns 597
    Processing of calculated columns 599
Choosing the right columns to store 599
Optimizing column storage 602
    Using column split optimization 602
    Optimizing high-cardinality columns 603
    Disabling attribute hierarchies 604
    Optimizing drill-through attributes 604
Managing VertiPaq Aggregations 604
Conclusions 607

Sample pages

Chapter 19 Analyzing DAX query plans **609**

Capturing DAX queries **609**

Introducing DAX query plans. **612**

Collecting query plans **613**

Introducing logical query plans. **614**

Introducing physical query plans **614**

Introducing storage engine queries. **616**

Capturing profiling information **617**

Using DAX Studio. **617**

Using the SQL Server Profiler **620**

Reading VertiPaq storage engine queries **624**

Introducing xmSQL syntax **624**

Understanding scan time **632**

Understanding ***DISTINCTCOUNT*** internals **634**

Understanding parallelism and datacache **635**

Understanding the VertiPaq cache **637**

Understanding ***CallbackDataID*** **640**

Reading DirectQuery storage engine queries **645**

Analyzing composite models. **646**

Using aggregations in the data model **647**

Reading query plans. **649**

Conclusions **655**

Chapter 20 Optimizing DAX **657**

Defining optimization strategies **658**

Identifying a single DAX expression to optimize **658**

Creating a reproduction query **661**

Analyzing server timings and query plan information. **664**

Identifying bottlenecks in the storage engine or formula engine **667**

Implementing changes and rerunning the test query **668**

Optimizing bottlenecks in DAX expressions **668**

Optimizing filter conditions **668**

Optimizing context transitions **672**

Sample pages

Optimizing *IF* conditions **678**
Reducing the impact of ***CallbackDataID*** **690**
Optimizing nested iterators **693**
Avoiding table filters for ***DISTINCTCOUNT*** **699**
Avoiding multiple evaluations by using variables **704**
Conclusions **709**

Index *711*

Sample pages

CHAPTER 4

Understanding evaluation contexts

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

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

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

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

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

Sample pages

Introducing evaluation contexts

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

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

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

Sample pages

Understanding filter contexts

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

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

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

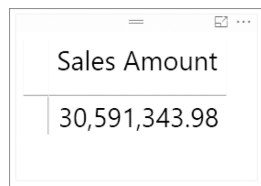


FIGURE 4-1 The measure *Sales Amount*, without a context, shows the grand total of sales.

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

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

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

Sample pages

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

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

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

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

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

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

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

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

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

Sample pages

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

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

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

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

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

Sample pages

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

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

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

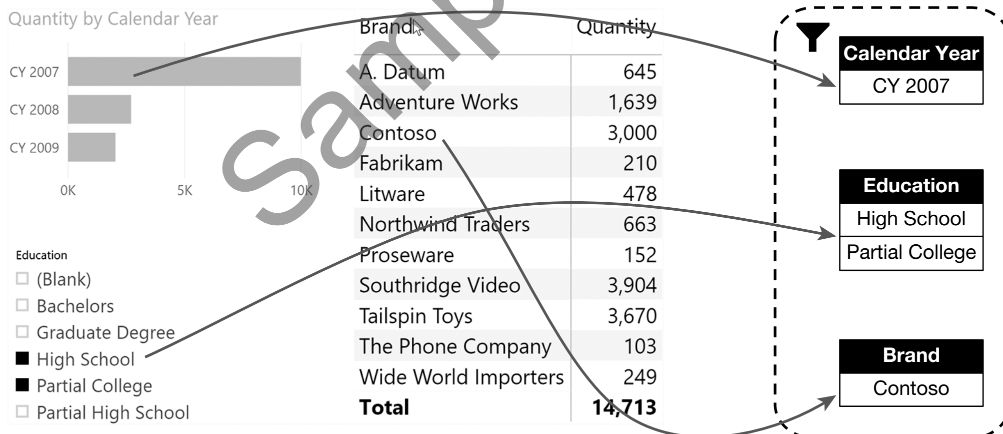
Brand	CY 2007	CY 2008	CY 2009	Total
A. Datum	57,276.00			**57,276.00**
Adventure Works	77,413.46		8,110.53	**85,523.99**
Contoso	125,596.01	2,638.18	14,156.95	**142,391.14**
Fabrikam	4,340.62	8,640.00	29,854.98	**42,835.60**
Litware	17,910.87		7,956.00	**25,866.87**
Northwind Traders	34,161.39	12,733.92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

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

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

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

Sample pages



Brand	Quantity
A. Datum	645
Adventure Works	1,639
Contoso	3,000
Fabrikam	210
Litware	478
Northwind Traders	663
Proseware	152
Southridge Video	3,904
Tailspin Toys	3,670
The Phone Company	103
Wide World Importers	249
Total	**14,713**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Sample pages

Understanding the row context

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

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

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

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

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

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

Sample pages

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

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

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

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

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

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

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

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

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

Sample pages

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

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

Testing your understanding of evaluation contexts

Before moving on to more complex descriptions about evaluation contexts, it is useful to test your understanding of contexts with a couple of examples. Please do not look at the explanation immediately; stop after the question and try to answer it. Then read the explanation to make sense of it. As a hint, try to remember, while thinking*, "The filter context filters*; *the row context iterates. This means that the row context does not filter, and the filter context does not iterate*."

Using *SUM* in a calculated column

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

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

Remember, this internally corresponds to this equivalent syntax:

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

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

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

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

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

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

Sample pages

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

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

Sample pages

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

Using columns in a measure

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

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

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

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

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

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

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

Using the row context with iterators

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

For example, look at the following DAX measure:

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

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

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

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

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

Sample pages

All iterators behave the same way:

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

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

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

Sample pages

Nested row contexts on different tables

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

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

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

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

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

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

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

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

Nested row contexts on the same table

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

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

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

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

Sample pages

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

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

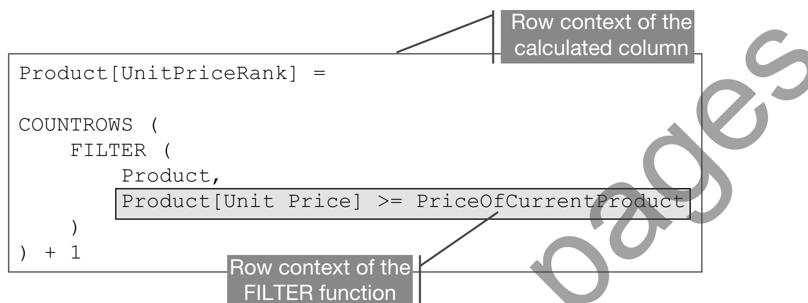


FIGURE 4-9 During the evaluation of the innermost expression, there are two row contexts on the same table.

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

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

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

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

Sample pages