FOR MICROSOFT EXCEL® 2010 & 2013

DAX Patterns

2015

ALBERTO FERRARI
MARCO RUSSO

sqlbi

DAX Patterns 2015

ALBERTO FERRARI
MARCO RUSSO

DAX Patterns 2015

Publisher: Loader
Editorial Production: SQLBI
Copyeditor: Lisa Maynard
Authors: Alberto Ferrari, Marco Russo
Cover Design: Daniele Perilli

ISBN-13: 9781505623635 (Paperback edition)

Download examples and resources at
http://www.daxpatterns.com

All the code in this book has been formatted with DAX Formatter

Introduction

We are accustomed to sharing nearly all the DAX code we develop by writing blog posts, speaking at conferences, and teaching courses. Often, people ask us about a scenario we have solved in the past and, every time, we have to search through our own blogs for the solution.

After four years of DAX coding, we had collected a good library of patterns and we decided to publish them on the http://www.daxpatterns.com website, sharing its content with all of our readers. As we reviewed the patterns, we took time to optimize the code, looking for the best solutions. Once we have published a solution on the web, we can easily find it when we need it again, freeing up our own minds (we have very active brain garbage collectors: DAX Patterns is our long-term DAX memory).

We have received many appreciations for the website, and several readers asked for a printed version of its content, so we did it! You are reading the printed version of the content of the DAX Patterns website as of December 2014.

All the patterns included in this book are also available for free on http://www.daxpatterns.com. We added only the first two chapters: DAX Fundamentals and Time Intelligence Functions, which are not patterns by themselves but can be useful to many readers.

Why should you pay for a book when its content is also available for free? Several reasons come to mind—a printed set of the patterns is generally faster to scan, it is available on a plane with no Wi-Fi, and studying "offline" generally leads to a better understanding of a topic.

You still need to access the website for the sample files. In fact, each chapter of this book contains a link to the corresponding article on http://www.daxpatterns.com, where you will find sample workbooks for both Excel 2010 and 2013. If needed, you can easily create a corresponding Analysis Services Tabular project starting from the Excel 2013 file.

We hope that you will find these patterns useful in your daily work and that you will improve your DAX knowledge by studying them. We had a lot of fun writing—now it is your time to enjoy reading!

If you have any feedback, share it with us by using comments on http://www.daxpatterns.com.

Alberto Ferrari

Marco Russo

CHAPTER 1

DAX Fundamentals

The Data Analysis Expression (DAX) language is a functional language used by PowerPivot for Excel and Analysis Services Tabular to define calculations within the data model and to query the data model itself. DAX has its roots in Microsoft Excel expressions and inherits a similar syntax and a functional approach, but it also extends its scope in order to handle calculations on data stored in multiple tables.

This chapter offers a summary of the DAX syntax and the important evaluation context behavior, which is fundamental knowledge for those hoping to master the language. Many of the patterns in this book are based on manipulating both the filter and the row context. Even if you can apply the patterns in your data model without a full understanding of the internal calculation behavior, the knowledge of evaluation context allows you to manipulate these patterns with full control.

DAX Syntax

A DAX expression returns either a table or a scalar value. A table in a DAX expression can have one or more columns and zero or more rows. A scalar value can be in any of the data types handled by the language. You can write DAX expressions to define calculated columns and measures (also known as calculated fields in Excel 2013), and as parts of a DAX query.

DAX Data Types

DAX supports the following data types:

- Integer
- Real
- Currency
- Date (datetime)
- TRUE/FALSE (Boolean)
- String
- BLOB (binary large object)

Usually type conversion happens automatically, and you do not have to call conversion functions in your expressions, even if in certain cases you might

want to do that to force a particular behavior or to make your statement more readable and explicit about intentions.

The Currency data type is a fixed-point decimal number that is very useful in financial calculations. The datetime data type internally stores the value using a floating-point number, wherein the integer corresponds to the number of days (starting from December 30, 1899), and the decimal identifies the fraction of the day (hours, minutes, and seconds are converted to decimal fractions of a day). Thus, the expression

```
= NOW() + 0.5
```

increases a date by 12 hours (exactly half a day). You should, however, consider using specific DAX functions such as DATEADD whenever possible to make your code more readable. If you need only the date part of a DATETIME, always remember to use TRUNC to get rid of the decimal part.

DAX Operators

Table 1-1 shows a list of operators available in the DAX language.

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
	= <>	Equal to Not equal to	[Country] = "USA" [Country] <> "USA"

Comparison	> >= < <=	Greater than Greater than or equal to Less than Less than or equal to	[Quantity] > 0 [Quantity] >= 100 [Quantity] < 0 [Quantity] <= 100
Text concatenation	&	Concatenation of strings	"Value is " & [Amount]
Logical	&& ||	AND condition between two Boolean expressions OR condition between two Boolean expressions	[Country] = "USA" && [Quantity] > 0 [Country] = "USA" || [Quantity] > 0

TABLE 1-1 Operators

Because of the compatibility with Excel syntax, the logical operators are also available as DAX functions. For example, you can write

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


which correspond, respectively, to these:

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

DAX Values

In a DAX expression, you can use scalar values such as "USA" or 0, which are called literals, or you can refer to the value of a column in a table. When you reference a column in order to get its value, you use the following basic syntax:

```
'Table Name'[Column Name]
```

Here is an example:

```
'Products'[ListPrice]
```

The table name precedes the column name. You can omit the single quote character that encloses the table name whenever the table name is a single name without spaces or other special characters and does not correspond to a reserved word. For example, in the following formula you can omit the quotes:

```
Products[ListPrice]
```

The square brackets that enclose the column name are mandatory. Even if the table name is optional, it is a best practice always to include it when you reference a column and to omit it when you reference a measure (which you access with the same syntax of a column name).

Empty or Missing Values

In DAX, BLANK represents any missing or empty value. You can obtain a blank value in DAX by calling the BLANK function, which has no arguments. Such a value is useful only as result of a DAX expression, because you cannot use it in a comparison statement (see ISBLANK for that in the “Conditional Statements” section later in this chapter). In a numeric expression, a blank is automatically converted into 0, whereas in a string expression, a blank is automatically converted into an empty string--with certain exceptions in which BLANK is retained in the expression result (such

as a product, numerator in a division, or sum of blanks). In the following examples, you can see how DAX handles BLANK in different operations involving numbers, string, and Boolean data types:

```
BLANK() + BLANK()   = BLANK()
10 * BLANK()        = BLANK()
BLANK() / 3         = BLANK()
BLANK() / BLANK()   = BLANK()
BLANK() || BLANK()  = FALSE
BLANK() && BLANK()  = FALSE
BLANK() - 10        = -10
18 + BLANK()        = 18
4 / BLANK()         = Infinity
0 / BLANK()         = NaN
FALSE || BLANK()    = FALSE
FALSE && BLANK()    = FALSE
TRUE || BLANK()     = TRUE
TRUE && BLANK()     = FALSE
```


Usually, you will use BLANK as a result for an expression assign

Conditional Statements

DAX is a functional language, and classical conditional statements are available as functions. The first and most used conditional statement is IF, which has three arguments: the first is the condition to test, the second is the value returned if the first argument evaluates to TRUE, and the third is the value returned otherwise. If you omit the third argument, it defaults to BLANK. Here you can see a few examples of expressions using the IF function:

```
IF ( 20 < 30, "second", "third" ) = "second"
IF ( 20 < 15, "second", BLANK() ) = BLANK()
IF ( 20 < 15, "second" ) = BLANK()
```


You might use nested IF statements to check different values for an expression. For example, you might decode the single-letter Status column of the Customer table into a more meaningful description using the following nested IF calls:

```
IF ( Customer[Status] = "A", "Platinum",
    IF ( Customer[Status] = "B", "Gold",
        IF ( Customer[Status] = "C", "Silver",
            IF ( Customer[Status] = "D", "White", "None" )
        )
    )
)
```

However, you can obtain much more readable code by using the SWITCH function--in fact, you can rewrite the previous expression in this way:

```
SWITCH (
    Customer[Status],
    "A", "Platinum",
    "B", "Gold",
    "C", "Silver",
    "D", "White",
    "None"
)
```

The SWITCH syntax is much more readable, but internally it generates exactly the same code as the nested IF statements and the performance is the same. In this syntax, you can see that the first argument is the expression evaluated once, and the following arguments are in pairs: if the first argument in the pair matches the result of the first expression, the SWITCH function returns the value of the second expression in the pair. The last argument is the expression value to return if there are no matches.

You can also use SWITCH to test different, unrelated conditions instead of matching the value returned by a single expression. You can obtain that by passing TRUE() as the first argument, and then writing the logical expression of each condition you want to test. For example, you can write the following SWITCH statement:

```
SWITCH (
    TRUE (),
    Products[UnitPrice] < 10, "LOW",
    Products[UnitPrice] < 50, "MEDIUM",
    Products[UnitPrice] < 100, "HIGH",
    "VERY HIGH"
)
```