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:

Click here to view code image

```
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.

Click here to view code image

```
= 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.

Click here to view code image

```
= 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.

Click here to view code image

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

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.

The result is a date.

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:

Click here to view code image

```
= 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:

Click here to view code image

```
= 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:

Click here to view code image

```
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	4 + 2
	–	Subtraction/negation	5 – 3
	*	Multiplication	4 * 2
	/	Division	4 / 2
Comparison	=	Equal to	[CountryRegion] = “USA”
	< >	Not equal to	[CountryRegion] <> “USA”
	>	Greater than	[Quantity] > 0
	> =	Greater than or equal to	[Quantity] >= 100
	<	Less than	[Quantity] < 0
	< =	Less than or equal to	

	<=		[Quantity] <= 100
Text concatenation	&	Concatenation of strings	“Value is” & [Amount]
Logical	&&	AND condition between two Boolean expressions	[CountryRegion] = “USA” && [Quantity]>0
	||	OR condition between two Boolean expressions	[CountryRegion] = “USA” || [Quantity] > 0
	IN	Inclusion of an element in a list	[CountryRegion] IN {“USA”, “Canada”}
	NOT	Boolean negation	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:

Click here to view code image

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

These examples are equivalent, respectively, to the following:

Click here to view code image

```
[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:

Click here to view code image

```
{ "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.

Click here to view code image

```
{
    ( "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:

Click here to view code image

```
'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:

Click here to view code image

```
( '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:

Click here to view code image

```
( '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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

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

However, it can also be defined as a measure:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
"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:

Click here to view code image

```
"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:

Click here to view code image

```
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:

Click here to view code image

```
=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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
= 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*:

Click here to view code image

```
= 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:

Click here to view code image

```
= 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):

Click here to view code image

```
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*.

Click here to view code image

```
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*.

Click here to view code image

```
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:

Click here to view code image

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

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

Click here to view code image

```
= 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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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 *COUNTROWS* 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 *DISTINCTCOUNTNOBLANK* 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:

Click here to view code image

```
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:

Click here to view code image

```
'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:

Click here to view code image

```
'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:

Click here to view code image

```
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

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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 (
        ISNUMBER ( People[Comma1] ),
        " " & LEFT ( People[Name],
People[Comma1] - 1 ),
        ""
    )
```

As you can see, the *FirstLastName* column is defined by a long DAX expression, but you must use it to avoid possible errors that would propagate to the whole column if even a single value generates an error.

Conversion functions

You learned previously that DAX performs automatic conversions of data types to adjust them to operator

needs. Although the conversion happens automatically, a set of functions can still perform explicit data type conversions.

CURRENCY can transform an expression into a Currency type, whereas *INT* transforms an expression into an Integer. *DATE* and *TIME* take the date and time parts as parameters and return a correct *DateTime*. *VALUE* transforms a string into a numeric format, whereas *FORMAT* gets a numeric value as its first parameter and a string format as its second parameter, and it can transform numeric values into strings. *FORMAT* is commonly used with *DateTime*. For example, the following expression returns "2019 Jan 12":

Click here to view code image

```
= FORMAT ( DATE ( 2019, 01, 12 ), "yyyy mmm
dd" )
```

The opposite operation, that is, converting strings into *DateTime* values, is performed using the *DATEVALUE* function.

DATEVALUE with dates in different format

DATEVALUE displays a special behavior regarding dates in different formats. In the European standard, dates are written with the format "dd/mm/yy", whereas Americans prefer to use "mm/dd/yy". For example, the 28th of February has different string representations in the two cultures. If you provide to *DATEVALUE* a date that cannot be converted using the default regional setting, instead of immediately raising an error, it tries a second conversion switching months and days. *DATEVALUE* also supports the unambiguous format "yyyy-mm-dd". As an example, the following three expressions evaluate to February 28, no matter which regional settings you have:

Click here to view code image

```
DATEVALUE ( "28/02/2018" ) -- This is February 28 in
European format
DATEVALUE ( "02/28/2018" ) -- This is February 28 in
American format
DATEVALUE ( "2018-02-28" ) -- This is February 28 (format
is not ambiguous)
```

Sometimes, *DATEVALUE* does not raise errors when you would expect them. However, this is the behavior of the function by design.

Date and time functions

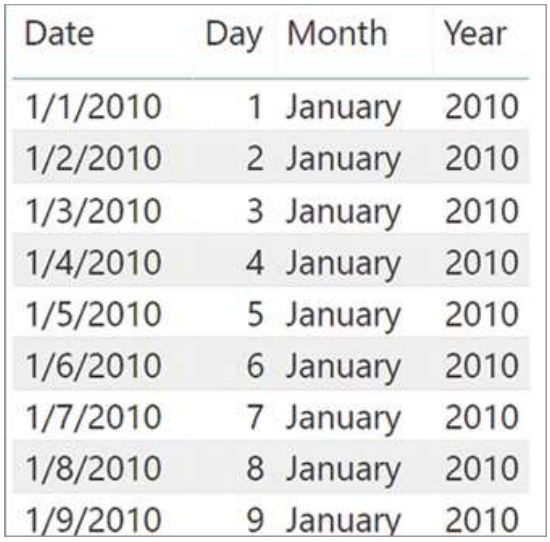
In almost every type of data analysis, handling time and dates is an important part of the job. Many DAX functions operate on date and time. Some of them correspond to similar functions in Excel and make simple transformations to and from a *DateTime* data type. The date and time functions are *DATE*, *DATEVALUE*, *DAY*, *EDATE*, *EOMONTH*, *HOUR*, *MINUTE*, *MONTH*, *NOW*, *SECOND*, *TIME*, *TIMEVALUE*, *TODAY*, *WEEKDAY*, *WEEKNUM*, *YEAR*, and *YEARFRAC*.

These functions are useful to compute values on top of dates, but they are not used to perform typical time intelligence calculations such as comparing aggregated values year over year or calculating the year-to-date value of a measure. To perform time intelligence calculations, you use another set of functions called time intelligence functions, which we describe in Chapter 8, "Time intelligence calculations."

As we mentioned earlier in this chapter, a *DateTime* data type internally uses a floating-point number wherein the integer part corresponds to the number of days after December 30, 1899, and the decimal part indicates the fraction of the day in time. Hours, minutes, and seconds are converted into decimal fractions of the day. Thus, adding an integer number to a *DateTime* value increments the value by a corresponding number of days. However, you will probably find it more convenient to use the conversion functions to extract the day, month, and year from a date. The following expressions used in Figure 2-10 show how to extract this information from a table containing a list of dates:

Click here to view code image

```
'Date'[Day] = DAY ( Calendar[Date] )
'Date'[Month] = FORMAT ( Calendar[Date],
"mmmm" )
'Date'[MonthNumber] = MONTH ( Calendar[Date]
)
'Date'[Year] = YEAR ( Calendar[Date] )
```



Date	Day	Month	Year
1/1/2010	1	January	2010
1/2/2010	2	January	2010
1/3/2010	3	January	2010
1/4/2010	4	January	2010
1/5/2010	5	January	2010
1/6/2010	6	January	2010
1/7/2010	7	January	2010
1/8/2010	8	January	2010
1/9/2010	9	January	2010

Figure 2-10 This example shows how to extract date information using date and time functions.

Relational functions

Two useful functions that you can use to navigate through relationships inside a DAX formula are *RELATED* and *RELATEDTABLE*.

You already know that a calculated column can reference column values of the table in which it is defined. Thus, a calculated column defined in *Sales* can reference any column of *Sales*. However, what if one must refer to a column in another table? In general, one cannot use columns in other tables unless a relationship is defined in the model between the two tables. If the two tables share a relationship, you can use the *RELATED* function to access columns in the related table.

For example, one might want to compute a calculated column in the *Sales* table that checks whether the product that has been sold is in the "Cell phones" category and, in that case, apply a reduction factor to the standard cost. To compute such a column, one must use a condition that checks the value of the product category, which is not in the *Sales* table. Nevertheless, a chain of relationships starts from *Sales*, reaching *Product Category* through *Product* and *Product Subcategory*, as shown in Figure 2-11.

Figure 2-11 Sales has a chained relationship with *Product Category*.

Regardless of how many steps are necessary to travel from the original table to the related table, DAX follows the complete chain of relationships, and it returns the related column value. Thus, the formula for the *AdjustedCost* column can look like this:

Click here to view code image

```
Sales[AdjustedCost] =
IF (
    RELATED ( 'Product Category'[Category] )
= "Cell Phone",
    Sales[Unit Cost] * 0.95,
    Sales[Unit Cost]
)
```

In a one-to-many relationship, *RELATED* can access the one-side from the many-side because in that case,

only one row in the related table exists, if any. If no such row exists, *RELATED* returns *BLANK*.

If an expression is on the one-side of the relationship and needs to access the many-side, *RELATED* is not helpful because many rows from the other side might be available for a single row. In that case, we can use *RELATEDTABLE*. *RELATEDTABLE* returns a table containing all the rows related to the current row. For example, if we want to know how many products are in each category, we can create a column in *Product Categor*y with this formula:

Click here to view code image

```
'Product Category'[NumOfProducts] =
COUNTROWS ( RELATEDTABLE ( Product ) )
```

For each product category, this calculated column shows the number of products related, as shown in Figure 2-12.

Category	NumOfProducts
Audio	115
Cameras and camcorders	372
Cell phones	285
Computers	606
Games and Toys	166
Home Appliances	661
Music, Movies and Audio Books	90
TV and Video	222

Figure 2-12 You can count the number of products by using *RELATEDTABLE*.

As is the case for *RELATED*, *RELATEDTABLE* can follow a chain of relationships always starting from the one-side and going toward the many-side. *RELATEDTABLE* is often used in conjunction with iterators. For example, if we want to compute the sum of

quantity multiplied by net price for each category, we can write a new calculated column as follows:

Click here to view code image

```
'Product Category'[CategorySales] =
SUMX (
    RELATEDTABLE ( Sales ),
    Sales[Quantity] * Sales[Net Price]
)
```

The result of this calculated column is shown in Figure 2-13.

Category	CategorySales
Audio	$384,518.16
Cameras and camcorders	$7,192,581.95
Cell phones	$1,604,610.26
Computers	$6,741,548.73
Games and Toys	$360,652.81
Home Appliances	$9,600,457.04
Music, Movies and Audio Books	$314,206.74
TV and Video	$4,392,768.29

Figure 2-13 Using *RELATEDTABLE* and iterators, we can compute the amount of sales per category.

Because the column is calculated, this result is consolidated in the table, and it does not change according to the user selection in the report, as it would if it were written in a measure.

CONCLUSIONS

In this chapter, you learned many new functions and started looking at some DAX code. You may not remember all the functions right away, but the more you use them, the more familiar they will become.

The more crucial topics you learned in this chapter are

- Calculated columns are columns in a table that are computed with a DAX expression. Calculated columns are computed at data refresh time and do not change their value depending on user selection.
- Measures are calculations expressed in DAX. Instead of being computed at refresh time like calculated columns are, measures are computed at query time. Consequently, the value of a measure depends on the user selection in the report.
- Errors might happen at any time in a DAX expression; it is preferable to detect the error condition beforehand rather than letting the error happen and intercepting it after the fact.
- Aggregators like SUM are useful to aggregate columns, whereas to aggregate expressions, you need to use iterators. Iterators work by scanning a table and evaluating an expression row by row. At the end of the iteration, iterators aggregate a result according to their semantics.

In the next chapter, you will continue on your learning path by studying the most important table functions available in DAX.

Chapter 3

Using basic table functions

In this chapter, you learn the basic table functions available in DAX. Table functions are regular DAX functions that—instead of returning a single value—return a table. Table functions are useful when writing both DAX queries and many advanced calculations that require iterating over tables. The chapter includes several examples of such calculations.

The goal of this chapter is to introduce the notion of table functions, but not to provide a detailed explanation of all the table functions in DAX. A larger number of table functions is included in Chapter 12, "Working with tables," and in Chapter 13, "Authoring queries." Here, we explain the role of most common and important table functions in DAX, and how to use them in common scenarios, including in scalar DAX expressions.

INTRODUCING TABLE FUNCTIONS

Until now, you have seen that a DAX expression usually returns a single value, such as a string or a number. An expression that results in a single value is called a *scalar expression*. When defining a measure or a calculated

column, you always write scalar expressions, as in the following examples:

Click here to view code image

```
= 4 + 3
= "DAX is a beautiful language"
= SUM ( Sales[Quantity] )
```

Indeed, the primary goal of a measure is to produce results that are rendered in a report, in a pivot table, or in a chart. At the end of the day, the source of all these reports is a number—in other words, a scalar expression. Nevertheless, as part of the calculation of a scalar value, you are likely to use tables. For example, a simple iteration like the following uses a table as part of the calculation of the sales amount:

Click here to view code image

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

In this example, *SUMX* iterates over the *Sales* table. Thus, though the result of the full calculation is a scalar value, during the computation the formula scans the *Sales* table. The same code could iterate the result of a table function, like the following code. This code computes the sales amount only for rows greater than one:

Click here to view code image

```
Sales Amount Multiple Items :=
SUMX (
    FILTER (
        Sales,
        Sales[Quantity] > 1
    ),
    Sales[Quantity] * Sales[Net Price]
)
```

In the example, we use a *FILTER* function in place of the reference to *Sales*. Intuitively, *FILTER* is a function that filters the content of a table based on a condition. We will describe *FILTER* in full later. For now, it is important to note that whenever you reference the content of a table, you can replace the reference with the result of a table function.

Important

In the previous code you see a filter applied to a sum aggregation. This is not a best practice. In the next chapters, you will learn how to use *CALCULATE* to implement more flexible and efficient filters. The purpose of the examples in this chapter is not to provide best practices for DAX measures, but rather to explain how table functions work using simple expressions. We will apply these concepts later in more complex scenarios.

Moreover, in Chapter 2, "Introducing DAX," you learned that you can define variables as part of a DAX expression. There, we used variables to store scalar values. However, variables can store tables too. For example, the previous code could be written this way by using a variable:

Click here to view code image

```
Sales Amount Multiple Items :=
VAR
    MultipleItemSales = FILTER ( Sales,
Sales[Quantity] > 1 )
RETURN
    SUMX (
        MultipleItemSales,
        Sales[Quantity] * Sales[Unit Price]
    )
```

MultipleItemSales is a variable that stores a whole table because its expression is a table function. We strongly encourage using variables whenever possible because they make the code easier to read. By simply assigning a name to an expression, you already are documenting your code extremely well.

In a calculated column or inside an iteration, one can also use the *RELATEDTABLE* function to retrieve all the rows of a related table. For example, the following calculated column in the *Product* table computes the sales amount of the corresponding product:

Click here to view code image

```
'Product'[Product Sales Amount] =
SUMX (
    RELATEDTABLE ( Sales ),
    Sales[Quantity] * Sales[Unit Price]
)
```

Table functions can be nested too. For example, the following calculated column in the *Product* table computes the product sales amount considering only sales with a quantity greater than one:

Click here to view code image

```
'Product'[Product Sales Amount Multiple
Items] =
SUMX (
    FILTER (
        RELATEDTABLE ( Sales ),
        Sales[Quantity] > 1
    ),
    Sales[Quantity] * Sales[Unit Price]
)
```

In the sample code, *RELATEDTABLE* is nested inside *FILTER*. As a rule, when there are nested calls, DAX evaluates the innermost function first and then evaluates the others up to the outermost function.

Note

As you will see later, the execution order of nested calls can be a source of confusion because *CALCULATE* and *CALCULATETABLE* have a different order of evaluation from *FILTER*. In the next section, you learn the behavior of *FILTER*. You will find the description for *CALCULATE* and *CALCULATETABLE* in Chapter 5, "Understanding *CALCULATE* and *CALCULATETABLE*."

In general, we cannot use the result of a table function as the value of a measure or of a calculated column. Both measures and calculated columns require the expression

to be a scalar value. Instead, we can assign the result of a table expression to a *calculated table*. A calculated table is a table whose value is determined by a DAX expression rather than loaded from a data source.

For example, we can create a calculated table containing all the products with a unit price greater than 3,000 by using a table expression like the following:

Click here to view code image

```
ExpensiveProducts =
FILTER (
    'Product',
    'Product'[Unit Price] > 3000
)
```

Calculated tables are available in Power BI and Analysis Services, but not in Power Pivot for Excel (as of 2019). The more you use table functions, the more you will use them to create more complex data models by using calculated tables and/or complex table expressions inside your measures.

INTRODUCING *EVALUATE* SYNTAX

Query tools such as DAX Studio are useful to author complex table expressions. In that case, a common statement used to inspect the result of a table expression is *EVALUATE*:

Click here to view code image

```
EVALUATE
FILTER (
    'Product',
    'Product'[Unit Price] > 3000
)
```

One can execute the preceding DAX query in any tool that executes DAX queries (DAX Studio, Microsoft Excel, SQL Server Management Studio, Reporting Services, and so on). A DAX query is a DAX expression that returns a table, used with the *EVALUATE* statement. *EVALUATE* has a complex syntax, which we fully cover in Chapter 13. Here we only introduce the more commonly used *EVALUATE* syntax, which is as follows:

Click here to view code image

```
[DEFINE { MEASURE <tableName>[<name>] =
<expression> }]
EVALUATE <table>
[ORDER BY {<expression> [{ASC | DESC}]} [,
...]]
```

The initial *DEFINE MEASURE* part can be useful to define measures that are local to the query. It becomes useful when we are debugging formulas because we can define a local measure, test it, and then deploy the code in the model once it behaves as expected. Most of the syntax is optional. Indeed, the simplest query one can

author retrieves all the rows and columns from an existing table, as shown in Figure 3-1:

```
EVALUATE 'Product'
```

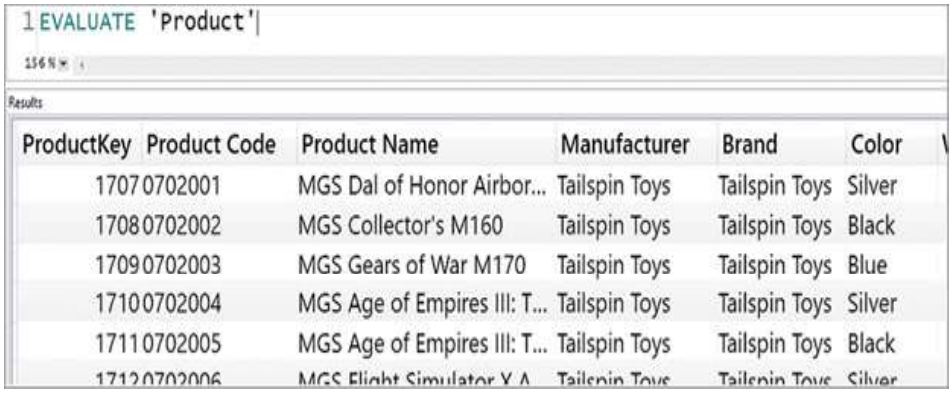


Figure 3-1 The result of the query execution in DAX Studio.

The *ORDER BY* clause controls the sort order:

Click here to view code image

```
EVALUATE
FILTER (
    'Product',
    'Product'[Unit Price] > 3000
)
ORDER BY
    'Product'[Color],
    'Product'[Brand] ASC,
    'Product'[Class] DESC
```

Note

Please note that the Sort By Column property defined in a model does not affect the sort order in a DAX query. The sort order specified by *EVALUATE* can only use columns included in the result. Thus, a client that generates a

dynamic DAX query should read the Sort By Column property in a model's metadata, include the column for the sort order in the query, and then generate a corresponding *ORDER BY* condition.

EVALUATE is not a powerful statement by itself. The power of querying with DAX comes from the power of using the many DAX table functions that are available in the language. In the next sections, you learn how to create advanced calculations by using and combining different table functions.

UNDERSTANDING *FILTER*

Now that we have introduced what table functions are, it is time to describe in full the basic table functions. Indeed, by combining and nesting the basic functions, you can already compute many powerful expressions. The first function you learn is *FILTER*. The syntax of *FILTER* is the following:

Click here to view code image

```
FILTER ( <table>, <condition> )
```

FILTER receives a table and a logical condition as parameters. As a result, *FILTER* returns all the rows satisfying the condition. *FILTER* is both a table function and an iterator at the same time. In order to return a result, it scans the table evaluating the condition on a row-by-row basis. In other words, it iterates the table.

For example, the following calculated table returns the Fabrikam products (Fabrikam being a brand).

Click here to view code image

```
FabrikamProducts =
FILTER (
    'Product',
    'Product'[Brand] = "Fabrikam"
)
```

FILTER is often used to reduce the number of rows in iterations. For example, if a developer wants to compute the sales of red products, they can author a measure like the following one:

Click here to view code image

```
RedSales :=
SUMX (
    FILTER (
        Sales,
        RELATED ( 'Product'[Color] ) = "Red"
    ),
    Sales[Quantity] * Sales[Net Price]
)
```

You can see the result in Figure 3-2, along with the total sales.

Category	Sales Amount	RedSales
Audio	384,518.16	33,123.82
Cameras and camcorders	7,192,581.95	1,514.39
Cell phones	1,604,610.26	38,227.47
Computers	6,741,548.73	240,222.29
Games and Toys	360,652.81	19,938.31
Home Appliances	9,600,457.04	770,373.33
Music, Movies and Audio Books	314,206.74	6,702.49
TV and Video	4,392,768.29	
Total	**30,591,343.98**	**1,110,102.10**

Figure 3-2 *RedSales* shows the amount of sales of only red products.

The *RedSales* measure iterated over a subset of the *Sales* table—namely the set of sales that are related to a red product. *FILTER* adds a condition to the existing conditions. For example, *RedSales* in the Audio row shows the sales of products that are both of Audio category and of Red color.

It is possible to nest *FILTER* in another *FILTER* function. In general, nesting two filters produces the same result as combining the conditions of the two *FILTER* functions with an *AND* function. In other words, the following two queries produce the same result:

Click here to view code image

```
FabrikamHighMarginProducts =
FILTER (
    FILTER (
        'Product',
        'Product'[Brand] = "Fabrikam"
```

```
    ),
    'Product'[Unit Price] > 'Product'[Unit
Cost] * 3
)

FabrikamHighMarginProducts =
FILTER (
    'Product',
    AND (
        'Product'[Brand] = "Fabrikam",
        'Product'[Unit Price] >
'Product'[Unit Cost] * 3
    )
)
```

However, performance might be different on large tables depending on the selectivity of the conditions. If one condition is more selective than the other, applying the most selective condition first by using a nested *FILTER* function is considered best practice.

For example, if there are many products with the Fabrikam brand, but few products priced at three times their cost, then the following query applies the filter over *Unit Price* and *Unit Cost* in the innermost *FILTER*. By doing so, the formula applies the most restrictive filter first, in order to reduce the number of iterations needed to check for the brand:

Click here to view code image

```
FabrikamHighMarginProducts =
FILTER (
    FILTER (
        'Product',
```

```
        'Product'[Unit Price] >
'Product'[Unit Cost] * 3
    ),
    'Product'[Brand] = "Fabrikam"
)
```

Using *FILTER*, a developer can often produce code that is easier to read and to maintain over time. For example, imagine you need to compute the number of red products. Without using table functions, one possible implementation might be the following:

Click here to view code image

```
NumOfRedProducts :=
SUMX (
    'Product',
    IF ( 'Product'[Color] = "Red", 1, 0 )
)
```

The inner *IF* returns either 1 or 0 depending on the color of the product, and summing this expression returns the number of red products. Although it works, this code is somewhat tricky. A better implementation of the same measure is the following:

Click here to view code image

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

This latter expression better shows what the developer wanted to obtain. Moreover, not only is the code easier to read for a human being, but the DAX optimizer is also better able to understand the developer's intention. Therefore, the optimizer produces a better query plan, leading in turn to better performance.

INTRODUCING *ALL* AND *ALLEXCEPT*

In the previous section you learned *FILTER*, which is a useful function whenever we want to restrict the number of rows in a table. Sometimes we want to do the opposite; that is, we want to extend the number of rows to consider for a certain calculation. In that case, DAX offers a set of functions designed for that purpose: *ALL*, *ALLEXCEPT*, *ALLCROSSFILTERED*, *ALLNOBLANKROW*, and *ALLSELECTED*. In this section, you learn *ALL* and *ALLEXCEPT*, whereas the latter two are described later in this chapter and *ALLCROSSFILTERD* is introduced in Chapter 14, "Advanced DAX concepts."

ALL returns all the rows of a table or all the values of one or more columns, depending on the parameters used. For example, the following DAX expression returns a *ProductCopy* calculated table with a copy of all the rows in the Product table:

Click here to view code image

```
ProductCopy = ALL ( 'Product' )
```

Note

ALL is not necessary in a calculated table because there are no report filters influencing it. However, *ALL* is useful in measures, as shown in the next examples.

ALL is extremely useful whenever we need to compute percentages or ratios because it ignores the filters automatically introduced by a report. Imagine we need a report like the one in Figure 3-3, which shows on the same row both the sales amount and the percentage of the given amount against the grand total.

Category	Sales Amount	Sales Pct
Audio	384,518.16	1.26%
Cameras and camcorders	7,192,581.95	23.51%
Cell phones	1,604,610.26	5.25%
Computers	6,741,548.73	22.04%
Games and Toys	360,652.81	1.18%
Home Appliances	9,600,457.04	31.38%
Music, Movies and Audio Books	314,206.74	1.03%
TV and Video	4,392,768.29	14.36%
Total	**30,591,343.98**	**100.00%**

Figure 3-3 The report shows the sales amounts and each percentage against the grand total.

The *Sales Amount* measure computes a value by iterating over the *Sales* table and performing the multiplication of *Sales[Quantity]* by *Sales[Net Price]*:

Click here to view code image

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

To compute the percentage, we divide the sales amount by the grand total. Thus, the formula must compute the grand total of sales even when the report is deliberately filtering one given category. This can be obtained by using the *ALL* function. Indeed, the following measure produces the total of all sales, no matter what filter is being applied to the report:

Click here to view code image

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

In the formula we replaced the reference to *Sales* with *ALL (Sales)*, making good use of the *ALL* function. At this point, we can compute the percentage by performing a simple division:

Click here to view code image

```
Sales Pct := DIVIDE ( [Sales Amount], [All
Sales Amount] )
```

Figure 3-4 shows the result of the three measures together.

Category	Sales Amount	All Sales Amount	Sales Pct
Audio	384,518.16	30,591,343.98	1.26%
Cameras and camcorders	7,192,581.95	30,591,343.98	23.51%
Cell phones	1,604,610.26	30,591,343.98	5.25%
Computers	6,741,548.73	30,591,343.98	22.04%
Games and Toys	360,652.81	30,591,343.98	1.18%
Home Appliances	9,600,457.04	30,591,343.98	31.38%
Music, Movies and Audio Books	314,206.74	30,591,343.98	1.03%
TV and Video	4,392,768.29	30,591,343.98	14.36%
Total	**30,591,343.98**	**30,591,343.98**	**100.00%**

Figure 3-4 The *All Sales Amount* measure always produces the grand total as a result.

The parameter of *ALL* cannot be a table expression. It needs to be either a table name or a list of column names. You have already learned what *ALL* does with a table. What is its result if we use a column instead? In that case, *ALL* returns all the distinct values of the column in the entire table. The *Categories* calculated table is obtained from the *Category* column of the *Product* table:

Click here to view code image

```
Categories = ALL ( 'Product'[Category] )
```

Figure 3-5 shows the result of the *Categories* calculated table.

Figure 3-5 Using *ALL* with a column produces the list of distinct values of that column.

We can specify multiple columns from the same table in the parameters of the *ALL* function. In that case, *ALL* returns all the existing combinations of values in those columns. For example, we can obtain the list of all categories and subcategories by adding the *Product[Subcategory]* column to the list of values, obtaining the result shown in Figure 3-6:

Click here to view code image

```
Categories =
ALL (
    'Product'[Category],
    'Product'[Subcategory]
)
```

Category	Subcategory
Audio	Bluetooth Headphones
Audio	MP4&MP3
Audio	Recording Pen
Cameras and camcorders	Camcorders
Cameras and camcorders	Cameras & Camcorders Accessories
Cameras and camcorders	Digital Cameras
Cameras and camcorders	Digital SLR Cameras
Cell phones	Cell phones Accessories
Cell phones	Home & Office Phones
Cell phones	Smart phones & PDAs
Cell phones	Touch Screen Phones

Figure 3-6 The list contains the distinct, existing values of category and subcategory.

Throughout all its variations, *ALL* ignores any existing filter in order to produce a result. We can use *ALL* as an argument of an iteration function, such as *SUMX* and *FILTER*, or as a filter argument in a *CALCULATE* function. You learn the *CALCULATE* function in Chapter 5.

If we want to include most, but not all the columns of a table in an *ALL* function call, we can use *ALLEXCEPT* instead. The syntax of *ALLEXCEPT* requires a table followed by the columns we want to exclude. As a result, *ALLEXCEPT* returns a table with a unique list of existing combinations of values in the other columns of the table.

ALLEXCEPT is a way to write a DAX expression that will automatically include in the result any additional columns that could appear in the table in the future. For

example, if we have a *Product* table with five columns (*ProductKey*, *Product Name*, *Brand*, *Class*, *Color*), the following two expressions produce the same result:

Click here to view code image

```
ALL ( 'Product'[Product Name],
'Product'[Brand], 'Product'[Class] )
ALLEXCEPT ( 'Product',
'Product'[ProductKey], 'Product'[Color] )
```

However, if we later add the two columns *Product[Unit Cost]* and *Product[Unit Price]*, then the result of *ALL* will ignore them, whereas *ALLEXCEPT* will return the equivalent of:

```
ALL (
    'Product'[Product Name],
    'Product'[Brand],
    'Product'[Class],
    'Product'[Unit Cost],
    'Product'[Unit Price]
)
```

In other words, with *ALL* we declare the columns we want, whereas with *ALLEXCEPT* we declare the columns that we want to remove from the result. *ALLEXCEPT* is mainly useful as a parameter of *CALCULATE* in advanced calculations, and it is seldomly adopted with simpler formulas. Thus, even if we included its description here for completeness, it will become useful only later in the learning path.

Top categories and subcategories

As an example of using *ALL* as a table function, imagine we want to produce a dashboard that shows the category and subcategory of products that sold more than twice the average sales amount. To produce this report, we need to first compute the average sales per subcategory and then, once the value has been determined, retrieve from the list of subcategories the ones that have a sales amount larger than twice that average.

The following code produces that table, and it is worth examining deeper to get a feeling of the power of table functions and variables:

Click here to view code image

```
BestCategories =
VAR Subcategories =
    ALL ( 'Product'[Category],
'Product'[Subcategory] )
VAR AverageSales =
    AVERAGEX (
        Subcategories,
        SUMX ( RELATEDTABLE ( Sales ),
Sales[Quantity] * Sales[Net Price] )
    )
VAR TopCategories =
    FILTER (
        Subcategories,
        VAR SalesOfCategory =
            SUMX ( RELATEDTABLE ( Sales ),
Sales[Quantity] * Sales[Net Price] )
        RETURN
            SalesOfCategory >= AverageSales * 2
    )
RETURN
    TopCategories
```

The first variable (*Subcategories*) stores the list of all categories and subcategories. Then, *AverageSales* computes the average of the sales amount for each subcategory. Finally, *Top-Categories* removes from *Subcategories* the subcategories that do not have a sales amount larger than twice the value of *AverageSales*.

The result of this table is visible in Figure 3-7.

Category	Subcategory
Cameras and camcorders	Camcorders
Cameras and camcorders	Digital SLR Cameras
Computers	Laptops
Computers	Projectors & Screens
Home Appliances	Washers & Dryers

Figure 3-7 These are the top subcategories that sold more than twice the average.

Once you master *CALCULATE* and filter contexts, you will be able to author the same calculations with a shorter and more efficient syntax. Nevertheless, in this example you can already appreciate how combining table functions can produce powerful results, which are useful for dashboards and reports.

UNDERSTANDING *VALUES*, *DISTINCT*, AND THE BLANK ROW

In the previous section, you saw that *ALL* used with one column returns a table with all its unique values. DAX provides two other similar functions that return a list of unique values for a column: *VALUES* and *DISTINCT*. These two functions look almost identical, the only difference being in how they handle the blank row that might exist in a table. You will learn about the optional blank row later in this section; for now let us focus on what these two functions perform.

ALL always returns all the distinct values of a column. On the other hand, *VALUES* returns only the distinct visible values. You can appreciate the difference between

the two behaviors by looking at the two following measures:

Click here to view code image

```
NumOfAllColors := COUNTROWS ( ALL (
'Product'[Color] ) )
NumOfColors := COUNTROWS ( VALUES (
'Product'[Color] ) )
```

NumOfAllColors counts all the colors of the *Product* table, whereas *NumOfColors* counts only the ones that—given the filter in the report—are visible. The result of these two measures, sliced by category, is visible in Figure 3-8.

Category	NumOfColors	NumOfAllColors
Audio	10	16
Cameras and camcorders	14	16
Cell phones	8	16
Computers	12	16
Games and Toys	11	16
Home Appliances	13	16
Music, Movies and Audio Books	8	16
TV and Video	4	16
Total	**16**	**16**

Figure 3-8 For a given category, only a subset of the colors is returned by *VALUES*.

Because the report slices by category, each given category contains products with some, but not all, the colors. *VALUES* returns the distinct values of a column evaluated in the current filter. If we use *VALUES* or

DISTINCT in a calculated column or in a calculated table, then their behavior is identical to that of *ALL* because there is no active filter. On the other hand, when used in a measure, these two functions compute their result considering the existing filters, whereas *ALL* ignores any filter.

As you read earlier, the two functions are nearly identical. It is now important to understand why *VALUES* and *DISTINCT* are two variations of the same behavior. The difference is the way they consider the presence of a blank row in the table. First, we need to understand how come a blank row might appear in our table if we did not explicitly create a blank row.

The fact is that the engine automatically creates a blank row in any table that is on the one-side of a relationship in case the relationship is invalid. To demonstrate the behavior, we removed all the silver-colored products from the *Product* table. Since there were 16 distinct colors initially and we removed one color, one would expect the total number of colors to be 15. Instead, the report in Figure 3-9 shows something unexpected: *NumOfAllColors* is still 16 and the report shows a new row at the top, with no name.

Category	NumOfColors	NumOfAllColors
	1	16
Audio	9	16
Cameras and camcorders	13	16
Cell phones	7	16
Computers	11	16
Games and Toys	10	16
Home Appliances	12	16
Music, Movies and Audio Books	7	16
TV and Video	3	16
Total	**16**	**16**

Figure 3-9 The first rows shows a blank for the category, and the total number of colors is 16 instead of 15.

Because *Product* is on the one-side of a relationship with *Sales*, for each row in the *Sales* table there is a related row in the *Product* table. Nevertheless, because we deliberately removed all the products with one color, there are now many rows in *Sales* that no longer have a valid relationship with the *Product* table. Be mindful, we did not remove any row from *Sales*; we removed a color with the intent of breaking the relationship.

To guarantee that these rows are considered in all the calculations, the engine automatically added to the *Product* table a row containing blank in all its columns. All the orphaned rows in *Sales* are linked to this newly introduced blank row.

Important

Only one blank row is added to the *Product* table, despite the fact that multiple different products referenced in the *Sales* table no longer have a

corresponding *ProductKey* in the *Product* table.

Indeed, in Figure 3-9 you can see that the first row shows a blank for the *Category* and accounts for one color. The number comes from a row containing blank in the category, blank in the color, and blank in all the columns of the table. You will not see the row if you inspect the table because it is an automatic row created during the loading of the data model. If, at some point, the relationship becomes valid again—if you were to add the silver products back—then the blank row will disappear from the table.

Certain functions in DAX consider the blank row as part of their result, whereas others do not. Specifically, *VALUES* considers the blank row as a valid row, and it returns it. On the other hand, *DISTINCT* does not return it. You can appreciate the difference by looking at the following new measure, which counts the *DISTINCT* colors instead of *VALUES*:

Click here to view code image

```
NumOfDistinctColors := COUNTROWS ( DISTINCT
( 'Product'[Color] ) )
```

The result is visible in Figure 3-10.

Category	NumOfColors	NumOfDistinctColors	NumOfAllColors
	1		16
Audio	9	9	16
Cameras and camcorders	13	13	16
Cell phones	7	7	16
Computers	11	11	16
Games and Toys	10	10	16
Home Appliances	12	12	16
Music, Movies and Audio Books	7	7	16
TV and Video	3	3	16
Total	**16**	**15**	**16**

Figure 3-10 *NumOfDistinctColors* shows a blank for the blank row, and its total shows 15 instead of 16.

A well-designed model should not present any invalid relationships. Thus, if your model is perfect, then the two functions always return the same values. Nevertheless, when dealing with invalid relationships, you need to be aware of this behavior because otherwise you might end up writing incorrect calculations. For example, imagine that we want to compute the average sales per product. A possible solution is to compute the total sales and divide that by the number of products, by using this code:

Click here to view code image

```
AvgSalesPerProduct :=
DIVIDE (
    SUMX (
        Sales,
        Sales[Quantity] * Sales[Net Price]
    ),
    COUNTROWS (
```

```
            VALUES ( 'Product'[Product Code] )
        )
    )
```

The result is visible in Figure 3-11. It is obviously wrong because the first row is a huge, meaningless number.

Category	AvgSalesPerProduct
	6,798,560.86
Audio	2,959.80
Cameras and camcorders	18,954.27
Cell phones	5,522.99
Computers	9,903.37
Games and Toys	2,242.14
Home Appliances	14,611.76
Music, Movies and Audio Books	3,337.06
TV and Video	14,698.67
Total	**14,560.37**

Figure 3-11 The first row shows a huge value accounted for a category with no name.

The number shown in the first row, where *Category* is blank, corresponds to the sales of all the silver products —which no longer exist in the *Product* table. This blank row associates all the products that were silver and are no longer in the *Product* table. The numerator of *DIVIDE* considers all the sales of silver products. The denominator of *DIVIDE* counts a single blank row returned by *VALUES*. Thus, a single non-existing product (the blank row) is cumulating the sales of many other products referenced in *Sales* and not available in the *Product* table, leading to a huge number. Here, the

problem is the invalid relationship, not the formula by itself. Indeed, no matter what formula we create, there are many sales of products in the *Sales* table for which the database has no information. Nevertheless, it is useful to look at how different formulations of the same calculation return different results. Consider these two other variations:

Click here to view code image

```
AvgSalesPerDistinctProduct :=
DIVIDE (
    SUMX ( Sales, Sales[Quantity] *
Sales[Net Price] ),
    COUNTROWS ( DISTINCT ( 'Product'[Product
Code] ) )
)

AvgSalesPerDistinctKey :=
DIVIDE (
    SUMX ( Sales, Sales[Quantity] *
Sales[Net Price] ),
    COUNTROWS ( VALUES ( Sales[ProductKey] )
)
)
```

In the first variation, we used *DISTINCT* instead of *VALUES*. As a result, *COUNTROWS* returns a blank and the result will be a blank. In the second variation, we still used *VALUES*, but this time we are counting the number of *Sales[ProductKey]*. Keep in mind that there are many different *Sales[ProductKey]* values, all related to the same blank row. The result is visible in Figure 3-12.

Category	AvgSalesPerProduct	AvgSalesPerDistinctProduct	AvgSalesPerDistinctKey
	6,798,560.86		18,474.35
Audio	2,959.80	2,959.80	3,634.18
Cameras and camcorders	18,954.27	18,954.27	20,786.51
Cell phones	5,522.99	5,522.99	6,163.00
Computers	9,903.37	9,903.37	11,416.98
Games and Toys	2,242.14	2,242.14	2,386.79
Home Appliances	14,611.76	14,611.76	16,238.64
Music, Movies and Audio Books	3,337.06	3,337.06	3,883.12
TV and Video	14,698.67	14,698.67	16,687.96
Total	**14,560.37**	**14,567.31**	**13,687.40**

Figure 3-12 In the presence of invalid relationships, the measures are most likely wrong—each in their own way.

It is interesting to note that *AvgSalesPerDistinctKey* is the only correct calculation. Since we sliced by *Category*, each category had a different number of invalid product keys—all of which collapsed to the single blank row.

However, the correct approach should be to fix the relationship so that no sale is orphaned of its product. The golden rule is to not have any invalid relationships in the model. If, for any reason, you have invalid relationships, then you need to be extremely cautious in how you handle the blank row, as well as how its presence might affect your calculations.

As a final note, consider that the *ALL* function always returns the blank row, if present. In case you need to remove the blank row from the result, then *ALLNOBLANKROW* is the function you will want to use.

***VALUES* of multiple columns**

The functions *VALUES* and *DISTINCT* only accept a single column as a parameter. There is no corresponding version for two or more columns, as there is for *ALL* and *ALLNOBLANKROW*. In case we need to obtain the distinct, visible combinations of values from different columns, then *VALUES* is of no help. Later in Chapter 12 you will learn that:

Click here to view code image

```
VALUES ( 'Product'[Category],
'Product'[Subcategory] )
```

can be obtained by writing:

Click here to view code image

```
SUMMARIZE ( 'Product', 'Product'[Category],
'Product'[Subcategory] )
```

Later, you will see that *VALUES* and *DISTINCT* are often used as a parameter of iterator functions. There are no differences in their results whenever the relationships are valid. In such a case, when you iterate over the values of a column, you need to consider the blank row as a valid row, in order to make sure that you iterate all the possible values. As a rule of thumb, *VALUES* should be your default choice, only leaving *DISTINCT* to cases when you want to explicitly exclude the possible blank value. Later in this book, you will also learn how to leverage *DISTINCT* instead of *VALUES* to avoid circular dependencies. We will cover it in Chapter 15, "Advanced relationships handling."

VALUES and *DISTINCT* also accept a table as an argument. In that case, they exhibit different behaviors:

- *DISTINCT* returns the distinct values of the table, not considering the blank row. Thus, duplicated rows are removed from the result.

- *VALUES* returns all the rows of the table, without removing duplicates, plus the additional blank row if present. Duplicated rows, in this case, are kept untouched.

USING TABLES AS SCALAR VALUES

Although *VALUES* is a table function, we will often use it to compute scalar values because of a special feature in DAX: a table with a single row and a single column can be used as if it were a scalar value. Imagine we produce a report like the one in Figure 3-13, reporting the number of brands sliced by category and subcategory.

Category	NumOfBrands
Audio	**3**
Bluetooth Headphones	2
MP4&MP3	1
Recording Pen	1
Cameras and camcorders	**3**
Camcorders	1
Cameras & Camcorders Accessories	1
Digital Cameras	1
Digital SLR Cameras	3
Cell phones	**2**
Cell phones Accessories	1

Figure 3-13 The report shows the number of brands available for each category and subcategory.

One might also want to see the names of the brands beside their number. One possible solution is to use *VALUES* to retrieve the different brands and, instead of counting them, return their value. This is possible only

in the special case when there is only one value for the brand. Indeed, in that case it is possible to return the result of *VALUES* and DAX automatically converts it into a scalar value. To make sure that there is only one brand, one needs to protect the code with an *IF* statement:

Click here to view code image

```
Brand Name :=
IF (
    COUNTROWS ( VALUES ( Product[Brand] ) )
= 1,
    VALUES ( Product[Brand] )
)
```

The result is visible in Figure 3-14. When the *Brand Name* column contains a blank, it means that there are two or more different brands.

Category	NumOfBrands	Brand Name
Audio	**3**	
Bluetooth Headphones	2	
MP4&MP3	1	Contoso
Recording Pen	1	Wide World Importers
Cameras and camcorders	**3**	
Camcorders	1	Fabrikam
Cameras & Camcorders Accessories	1	Contoso
Digital Cameras	1	A. Datum
Digital SLR Cameras	3	
Cell phones	**2**	
Cell phones Accessories	1	Contoso

Figure 3-14 When *VALUES* returns a single row, we can use it as a scalar value, as in the *Brand Name* measure.

The *Brand Name* measure uses *COUNTROWS* to check whether the *Color* column of the *Products* table only has one value selected. Because this pattern is frequently used in DAX code, there is a simpler function that checks whether a column only has one visible value: *HASONEVALUE*. The following is a better implementation of the *Brand Name* measure, based on *HASONEVALUE*:

Click here to view code image

```
Brand Name :=
IF (
    HASONEVALUE ( 'Product'[Brand] ),
    VALUES ( 'Product'[Brand] )
)
```

Moreover, to make the lives of developers easier, DAX also offers a function that automatically checks if a column contains a single value and, if so, it returns the value as a scalar. In case there are multiple values, it is also possible to define a default value to be returned. That function is *SELECTEDVALUE*. The previous measure can also be defined as

Click here to view code image

```
Brand Name := SELECTEDVALUE (
'Product'[Brand] )
```

By including the second optional argument, one can provide a message stating that the result contains

multiple results:

Click here to view code image

```
Brand Name := SELECTEDVALUE (
'Product'[Brand], "Multiple brands" )
```

The result of this latest measure is visible in Figure 3-15.

Category	NumOfBrands	Brand Name
Audio	**3**	**Multiple brands**
Bluetooth Headphones	2	Multiple brands
MP4&MP3	1	Contoso
Recording Pen	1	Wide World Importers
Cameras and camcorders	**3**	**Multiple brands**
Camcorders	1	Fabrikam
Cameras & Camcorders Accessories	1	Contoso
Digital Cameras	1	A. Datum
Digital SLR Cameras	3	Multiple brands
Cell phones	**2**	**Multiple brands**
Cell phones Accessories	1	Contoso

Figure 3-15 *SELECTEDVALUE* returns a default value in case there are multiple rows for the *Brand Name* column.

What if, instead of returning a message like "Multiple brands," one wants to list all the brands? In that case, an option is to iterate over the *VALUES* of *Product[Brand]* and use the *CONCATENATEX* function, which produces a good result even if there are multiple values:

Click here to view code image

```
[Brand Name] :=
CONCATENATEX (
    VALUES ( 'Product'[Brand] ),
    'Product'[Brand],
    ", "
)
```

Now the result contains the different brands separated by a comma instead of the generic message, as shown in Figure 3-16.

Category	NumOfBrands	Brand Name
Audio	**3**	**Contoso, Wide World Importers, Northwind Traders**
Bluetooth Headphones	2	Wide World Importers, Northwind Traders
MP4&MP3	1	Contoso
Recording Pen	1	Wide World Importers
Cameras and camcorders	**3**	**Contoso, Fabrikam, A. Datum**
Camcorders	1	Fabrikam
Cameras & Camcorders Accessories	1	Contoso
Digital Cameras	1	A. Datum
Digital SLR Cameras	3	Contoso, Fabrikam, A. Datum
Cell phones	**2**	**Contoso, The Phone Company**
Cell phones Accessories	1	Contoso

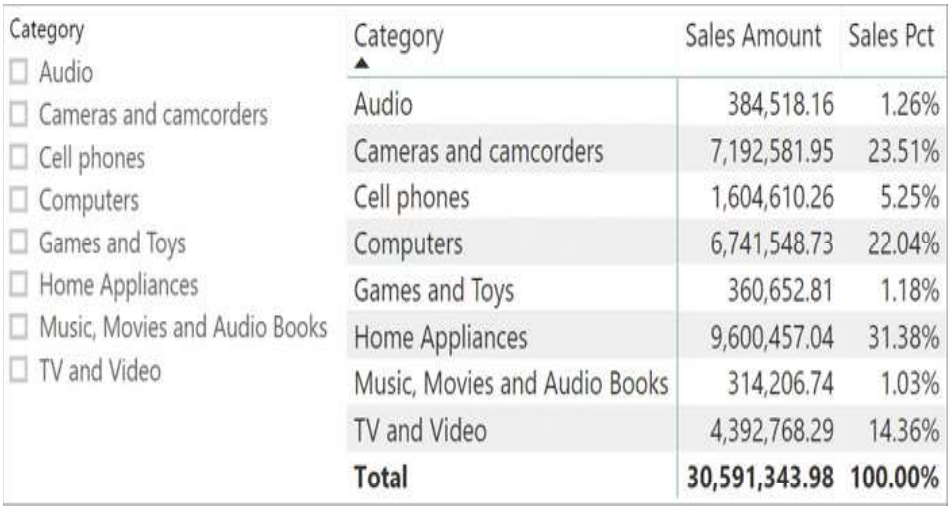
Figure 3-16 *CONCATENATEX* builds strings out of tables, concatenating expressions.

INTRODUCING *ALLSELECTED*

The last table function that belongs to the set of basic table functions is *ALLSELECTED*. Actually, *ALLSELECTED* is a very complex table function—probably the most complex table function in DAX. In Chapter 14, we will uncover all the secrets of

ALLSELECTED. Nevertheless, *ALLSELECTED* is useful even in its basic implementation. For that reason, it is worth mentioning in this introductory chapter.

ALLSELECTED is useful when retrieving the list of values of a table, or a column, as visible in the current report and considering all and only the filters outside of the current visual. To see when *ALLSELECTED* becomes useful, look at the report in Figure 3-17.



Category	Sales Amount	Sales Pct
Audio	384,518.16	1.26%
Cameras and camcorders	7,192,581.95	23.51%
Cell phones	1,604,610.26	5.25%
Computers	6,741,548.73	22.04%
Games and Toys	360,652.81	1.18%
Home Appliances	9,600,457.04	31.38%
Music, Movies and Audio Books	314,206.74	1.03%
TV and Video	4,392,768.29	14.36%
Total	**30,591,343.98**	**100.00%**

Figure 3-17 The report contains a matrix and a slicer, on the same page.

The value of *Sales Pct* is computed by the following measure:

Click here to view code image

```
Sales Pct :=
DIVIDE (
    SUMX ( Sales, Sales[Quantity] *
Sales[Net Price] ),
    SUMX ( ALL ( Sales ), Sales[Quantity] *
Sales[Net Price] )
)
```

Because the denominator uses the *ALL* function, it always computes the grand total of all sales, regardless of any filter. As such, if one uses the slicer to reduce the number of categories shown, the report still computes the percentage against all the sales. For example, Figure 3-18 shows what happens if one selects some categories with the slicer.

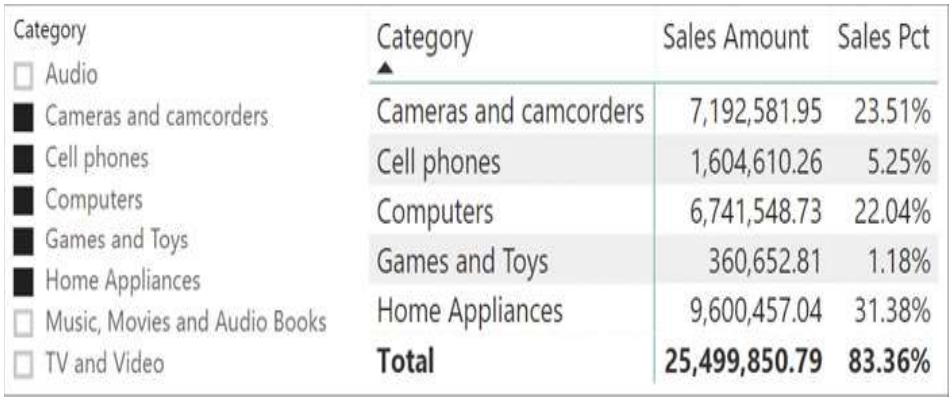


Figure 3-18 Using *ALL*, the percentage is still computed against the grand total of all sales.

Some rows disappeared as expected, but the amounts reported in the remaining rows are unchanged. Moreover, the grand total of the matrix no longer accounts for 100%. If this is not the expected result, meaning that you want the percentage to be computed not against the grand total of sales but rather only on the selected values, then *ALLSELECTED* becomes useful.

Indeed, by writing the code of *Sales Pct* using *ALLSELECTED* instead of *ALL*, the denominator computes the sales of all categories considering all and only the filters outside of the matrix. In other words, it returns the sales of all categories except Audio, Music, and TV.

Click here to view code image

```
Sales Pct :=
DIVIDE (
    SUMX ( Sales, Sales[Quantity] *
Sales[Net Price] ),
    SUMX ( ALLSELECTED ( Sales ),
Sales[Quantity] * Sales[Net Price] )
)
```

The result of this latter version is visible in Figure 3-19.

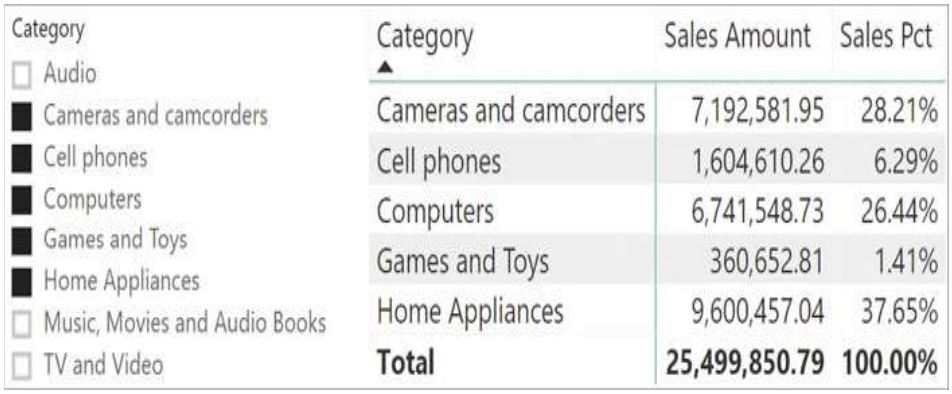


Figure 3-19 Using *ALLSELECTED*, the percentage is computed against the sales only considering outer filters.

The total is now 100% and the numbers reported reflect the percentage against the visible total, not against the grand total of all sales. *ALLSELECTED* is a powerful and useful function. Unfortunately, to achieve this purpose, it ends up being an extraordinarily complex function too. Only much later in the book will we be able to explain it in full. Because of its complexity, *ALLSELECTED* sometimes returns unexpected results. By unexpected we do not mean wrong, but rather,

ridiculously hard to understand even for seasoned DAX developers.

When used in simple formulas like the one we have shown here, *ALLSELECTED* proves to be particularly useful, anyway.

CONCLUSIONS

As you have seen in this chapter, basic table functions are already immensely powerful, and they allow you to start creating many useful calculations. *FILTER*, *ALL*, *VALUES* and *ALLSELECTED* are extremely common functions that appear in many DAX formulas.

Learning how to mix table functions to produce the result you want is particularly important because it will allow you to seamlessly achieve advanced calculations. Moreover, when mixed with the power of *CALCULATE* and of context transition, table functions produce compact, neat, and powerful calculations. In the next chapters, we introduce evaluation contexts and the *CALCULATE* function. After having learned *CALCULATE*, you will probably revisit this chapter to use table functions as parameters of *CALCULATE*, thus leveraging their full potential.