Microsoft

Tabular Modeling in Microsoft SQL Server Analysis Services

Second Edition

Professional

Marco Russo
Alberto Ferrari

Microsoft

Tabular Modeling in Microsoft SQL Server Analysis Services, Second Edition

Marco Russo
and Alberto Ferrari

In practice, this expression is valid only when it is possible to identify something similar to the generic concept of "current row" in the Sales table. This concept is formally defined as *row context*. A column reference in a DAX expression is valid only when there is an active row context for the table that is referenced. You have a row context active for the DAX expressions written in the following:

- A calculated column
- The argument executed in an iterator function in DAX (all the functions with an X suffix and any other function that iterates a table, such as FILTER, ADDCOLUMNS, SELECTCOLUMNS, and many others)
- The filter expression for a security role

If you try to evaluate a column reference when there is no row context active for the referenced table, you get a syntax error.

A row context does not propagate to other tables automatically. You can use a relationship to propagate a row context to another table, but this requires the use of a specific DAX function called RELATED.

CALCULATE and CALCULATETABLE

DAX has two functions that can modify a filter context before executing an expression: CALCULATE and CALCULATETABLE. The only difference between the two functions is that the former returns a single value (string or numeric), whereas the latter executes a table expression and returns a table.

The filter context is a set of filters that are applied to columns and/or tables of the data model. Each filter corresponds to a list of the values allowed for one or more columns, or for the entire table. By invoking CALCULATE or CALCULATETABLE, you modify the existing filter context, overriding any existing filters and/or adding new filters. To simplify the explanation, we consider the syntax of CALCULATE, but all the considerations apply to CALCULATETABLE, too.

The syntax for CALCULATE is as follows:

```
CALCULATE ( expression, filter1, filter2, …, filterN )
```

CALCULATE accepts any number of parameters. The only mandatory one is the first parameter in the expression. We call the conditions following the first parameter the *filter arguments*.

CALCULATE does the following:

- It places a copy of the current filter context into a new filter context.
- It evaluates each filter argument and produces for each condition the list of valid values for that specific column.
- If two or more filter arguments affect the same column filters, they are merged together using an AND operator (or, in mathematical terms, using the set intersection).

- It uses the new condition to replace the existing filters on the columns in the model. If a column already has a filter, then the new filter replaces the existing one. If, on the other hand, the column does not have a filter, then DAX simply applies the new column filter.
- After the new filter context is evaluated, CALCULATE computes the first argument (the expression) in the new filter context. At the end, it will restore the original filter context, returning the computed result.

The filters accepted by CALCULATE can be of the following two types:

- **List of values** This appears in the form of a table expression. In this case, you provide the exact list of values that you want to see in the new filter context. The filter can be a table with a single column or with many columns, as is the case of a filter on a whole table.
- **Boolean conditions** An example of this might be `Product[Color] = "White"`. These filters need to work on a single column because the result must be a list of values from a single column.

If you use the syntax with a Boolean condition, DAX will transform it into a list of values. For example, you might write the following expression:

```
CALCULATE (
    SUM ( Sales[SalesAmount] ),
    Product[Color] = "Red"
)
```

Internally, DAX transforms the expression into the following one:

```
CALCULATE (
    SUM ( Sales[SalesAmount] ),
    FILTER (
        ALL ( Product[Color] ),
        Product[Color] = "Red"
    )
)
```

Note The ALL function ignores any existing filter context, returning a table with all the unique values of the column specified.

For this reason, you can reference only one column in a filter argument with a Boolean condition. DAX must detect the column to iterate in the FILTER expression, which is generated in the background automatically. If the Boolean expression references more columns, then you must write the FILTER iteration in an explicit way.

Context transition

CALCULATE performs another very important task: It transforms any existing row context into an equivalent filter context. This is important when you have an aggregation within an iterator or when, in general, you have a row context. For example, the following expression (defined in the No CT measure shown later in Figure 4-2) computes the quantity of all the sales (of any product previously selected in the filter context) and multiplies it by the number of products:

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

The SUM aggregation function ignores the row context on the Product table produced by the iteration made by SUMX. However, by embedding the SUM in a CALCULATE function, you transform the row context on Product into an equivalent filter context. This automatically propagates to the Sales table thanks to the existing relationship in the data model between Product and Sales. The following expression is defined in the Explicit CT measure:

```
SUMX (
    Product,
    CALCULATE ( SUM ( Sales[Quantity] ) )
)
```

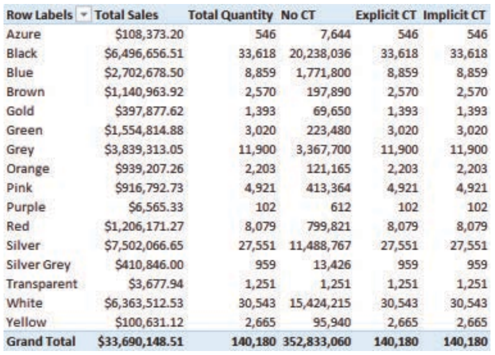
When you use a measure reference in a row context, there is always an implicit CALCULATE function surrounding the expression executed in the measure, so the previous expression corresponds to the following one, defined in the Implicit CT measure:

```
SUMX (
    Product,
    [Total Quantity]
)
```

The measure from Total Quantity in the previous expression corresponds to the following expression:

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

As you see in the results shown in Figure 4-2, replacing a measure with the underlying DAX expression is not correct. You must wrap such an expression within a CALCULATE function, which performs the same context transition made by invoking a measure reference.



Row Labels	Total Sales	Total Quantity	No CT	Explicit CT	Implicit CT
Azure	$108,373.20	546	7,644	546	546
Black	$6,496,656.51	33,618	20,238,036	33,618	33,618
Blue	$2,702,678.50	8,859	1,771,800	8,859	8,859
Brown	$1,140,963.92	2,570	197,890	2,570	2,570
Gold	$397,877.62	1,393	69,650	1,393	1,393
Green	$1,554,814.88	3,020	223,480	3,020	3,020
Grey	$3,839,313.05	11,900	3,367,700	11,900	11,900
Orange	$939,207.26	2,203	121,165	2,203	2,203
Pink	$916,792.73	4,921	413,364	4,921	4,921
Purple	$6,565.33	102	612	102	102
Red	$1,206,171.27	8,079	799,821	8,079	8,079
Silver	$7,502,066.65	27,551	11,488,767	27,551	27,551
Silver Grey	$410,846.00	959	13,426	959	959
Transparent	$3,677.94	1,251	1,251	1,251	1,251
White	$6,363,512.53	30,543	15,424,215	30,543	30,543
Yellow	$100,631.12	2,665	95,940	2,665	2,665
Grand Total	**$33,690,148.51**	**140,180**	**352,833,060**	**140,180**	**140,180**

FIGURE 4-2 The different results of a similar expression, with and without context transition.

Variables

When writing a DAX expression, you can avoid repeating the same expression by using variables. For example, look at the following expression:

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

You can define many variables, and they are local to the expression in which you define them. Variables are very useful both to simplify the code and because they enable you to avoid repeating the same subexpression. Variables are computed using lazy evaluation. This means that if you define a variable that, for any reason, is not used in your code, then the variable will never be evaluated. If it needs to be computed, then this happens only once. Later usages of the variable will read the previously computed value. Thus, they are also useful as an optimization technique when you use a complex expression multiple times.

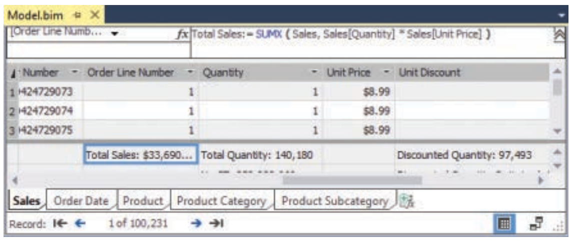
Measures

You define a *measure* whenever you want to aggregate values from many rows in a table. The following convention is used in this book to define a measure:

```
Table[MeasureName] := <expression>
```

This syntax does not correspond to what you write in the formula editor in Visual Studio because you do not specify the table name there. We use this writing convention in the book to optimize the space required for a measure definition. For example, the definition of the Total Sales measure in the Sales table (which you can see in Figure 4-3) is written in this book as the following expression:

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



Model.bim

[Order Line Numb... fx Total Sales: = SUMX (Sales, Sales[Quantity] * Sales[Unit Price])

	Number	Order Line Number	Quantity	Unit Price	Unit Discount
1	424729073	1	1	$8.99	
2	424729074	1	1	$8.99	
3	424729075	1	1	$8.99	
		Total Sales: $33,690...	Total Quantity: 140,180		Discounted Quantity: 97,493

FIGURE 4-3 Defining a measure in Visual Studio that includes only the measure name; the table is implicit.

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. In fact, you can move a measure from one table to another one without losing its functionality.

The expression is executed in a filter context and does not have a row context. For this reason, you must use aggregation functions, and you cannot use a direct column reference in the expression of a measure. However, a measure can reference other measures. You can write the formula to calculate the margin of sales as a percentage by using an explicit DAX syntax, or by referencing measures that perform part of the calculation. The following example defines four measures, where the Margin and Margin % measures reference other measures:

```
Sales[Total Sales] := SUMX ( Sales, Sales[Quantity] * Sales[Unit Price] )
Sales[Total Cost]  := SUMX ( Sales, Sales[Quantity] * Sales[Unit Cost] )
Sales[Margin]      := [Total Sales] - [Total Cost]
Sales[Margin %]    := DIVIDE ( [Margin], [Total Sales] )
```

The following Margin % Expanded measure corresponds to Margin %. All the referenced measures are expanded in a single DAX expression without the measure references. The column references are always executed in a row context generated by a DAX function (always `SUMX` in the following example):

```
Sales[Margin % Expanded] :=
DIVIDE (
    SUMX ( Sales, Sales[Quantity] * Sales[Unit Price] )
        - SUMX ( Sales, Sales[Quantity] * Sales[Unit Cost] ),
    SUMX ( Sales, Sales[Quantity] * Sales[Unit Price] )
)
```

You can also write the same expanded measure using variables, making the code more readable and avoiding the duplication of the same DAX subexpression (as it is the case for TotalSales, in this case):

```
Sales[Margin % Variables]:=
VAR TotalSales =
    SUMX ( Sales, Sales[Quantity] * Sales[Unit Price] )
VAR TotalCost =
    SUMX ( Sales, Sales[Quantity] * Sales[Unit Cost] )
VAR Margin = TotalSales – TotalCost
RETURN
    DIVIDE ( Margin, TotalSales )
```

Calculated columns

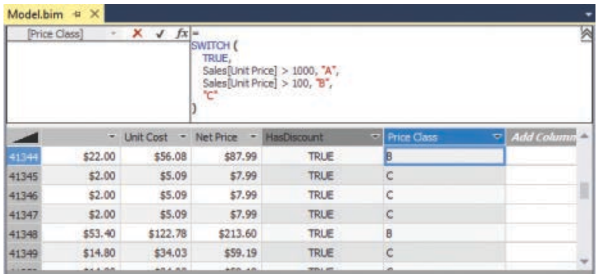
A calculated column is just like any other column in a table. You can use it in the rows, columns, filters, or values of a PivotTable or in any other report. You can also use a calculated column to define a relationship. The DAX expression defined for a calculated column operates in the context of the current row of the table to which it belongs (a *row context*). Any column reference returns the value of that column for the current row. You cannot directly access the values of the other rows. If you write an aggregation, the initial filter context is always empty (there are no filters active in a row context).

The following convention is used in this book to define a calculated column:

```
Table[ColumnName] = <expression>
```

This syntax does not correspond to what you write in the formula editor in Visual Studio because you do not specify the table and column names there. We use this writing convention in the book to optimize the space required for a calculated column definition. For example, the definition of the Price Class calculated column in the Sales table (see Figure 4-4) is written in this book as follows:

```
Sales[Price Class] =
SWITCH (
    TRUE,
    Sales[Unit Price] > 1000, "A",
    Sales[Unit Price] > 100, "B",
    "C"
)
```



```
=
SWITCH (
    TRUE,
    Sales[Unit Price] > 1000, "A",
    Sales[Unit Price] > 100, "B",
    "C"
)
```

		Unit Cost	Net Price	HasDiscount	Price Class	Add Column
41344	$22.00	$56.08	$87.99	TRUE	B	
41345	$2.00	$5.09	$7.99	TRUE	C	
41346	$2.00	$5.09	$7.99	TRUE	C	
41347	$2.00	$5.09	$7.99	TRUE	C	
41348	$53.40	$122.78	$213.60	TRUE	B	
41349	$14.80	$34.03	$59.19	TRUE	C	

FIGURE 4-4 How the definition of a calculated column in Visual Studio does not include the table and column names.

Calculated columns are computed during the database processing and then stored in the model. This might seem strange if you are accustomed to SQL-computed columns (not persisted), which are computed at query time and do not use memory. In Tabular, however, all the calculated columns occupy space in memory and are computed during table processing.

This behavior is helpful whenever you create very complex calculated columns. The time required to compute them is always at process time and not query time, resulting in a better user experience. Nevertheless, you must remember that a calculated column uses precious RAM. If, for example, you have a complex formula for a calculated column, you might be tempted to separate the steps of computation in 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 precious space.

For example, if you have a calculated column called LineAmount, it is defined as follows:

```
Sales[LineAmount] = Sales[Quantity] * Sales[UnitPrice]
```

You might create the following Total Amount measure:

```
Sales[Total Amount] := SUM ( Sales[LineAmount] )
```