Microsoft

The Definitive Guide to DAX

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

SECOND EDITION

Marco Russo and Alberto Ferrari

Sample files on the web

Microsoft

The Definitive Guide to DAX: Business intelligence with Microsoft Power BI, SQL Server Analysis Services, and Excel

Second Edition

Marco Russo and Alberto Ferrari

In this scenario, *RANKX* is the function to use. *RANKX* is an iterator and it is a simple function. Nevertheless, its use hides some complexities that are worth a deeper explanation.

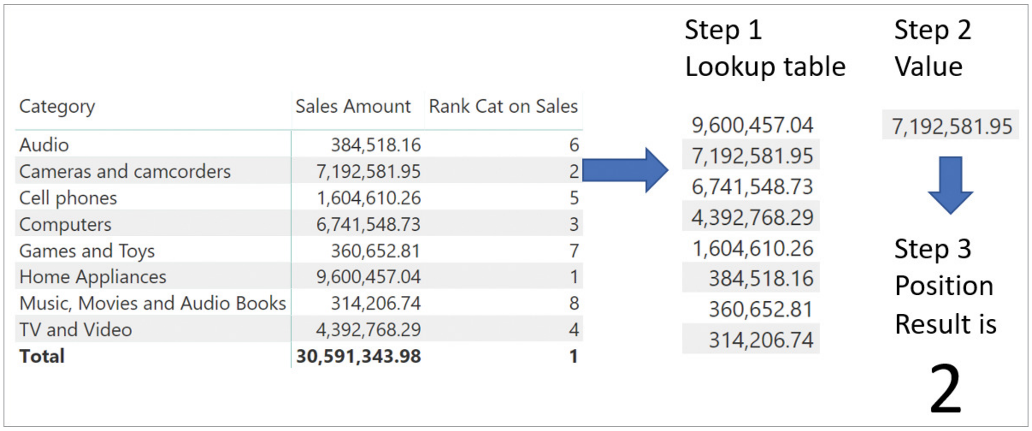
The code of *Rank Cat on Sales* is the following:

```
Rank Cat on Sales :=
RANKX (
    ALL ( 'Product'[Category] ),
    [Sales Amount]
)
```

RANKX operates in three steps:

1. *RANKX* builds a lookup table by iterating over the table provided as the first parameter. During the iteration it evaluates its second parameter in the row context of the iteration. At the end, it sorts the lookup table.
2. *RANKX* evaluates its second parameter in the original evaluation context.
3. *RANKX* returns the position of the value computed in the second step by searching its place in the sorted lookup table.

The algorithm is outlined in Figure 7-11, where we show the steps needed to compute the value of 2, the ranking of Cameras and camcorders according to *Sales Amount*.



Category	Sales Amount	Rank Cat on Sales
Audio	384,518.16	6
Cameras and camcorders	7,192,581.95	2
Cell phones	1,604,610.26	5
Computers	6,741,548.73	3
Games and Toys	360,652.81	7
Home Appliances	9,600,457.04	1
Music, Movies and Audio Books	314,206.74	8
TV and Video	4,392,768.29	4
Total	**30,591,343.98**	**1**

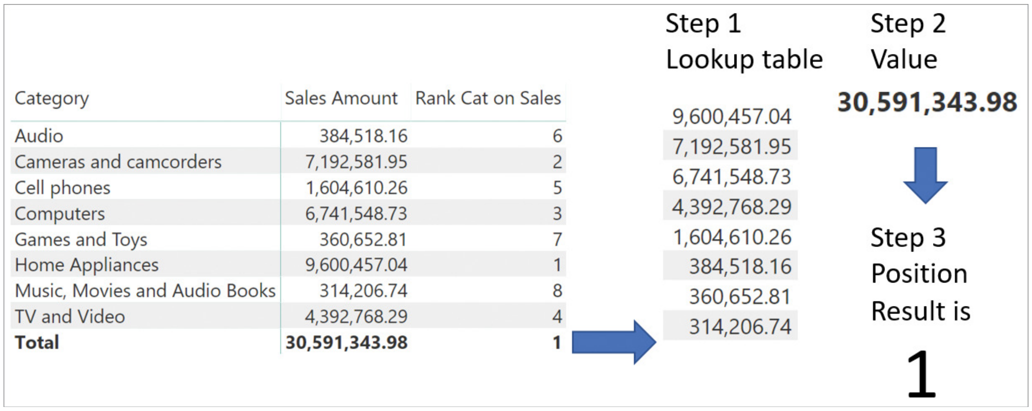
FIGURE 7-11 *RANKX* requires three steps to determine the ranking of Cameras and camcorders.

Here is a more detailed description of the behavior of *RANKX* in our example:

- The lookup table is built during the iteration. In the code, we had to use *ALL* on the product category to ignore the current filter context that would otherwise filter the only category visible, producing a lookup table with only one row.
- The value of *Sales Amount* is a different one for each category because of context transition. Indeed, during the iteration there is a row context. Because the expression to evaluate is a measure that contains a hidden *CALCULATE*, context transition makes DAX compute the value of *Sales Amount* only for the given category.

- The lookup table only contains values. Any reference to the category is lost: Ranking takes place only on values, once they are sorted correctly.
- The value determined in step 2 comes from the evaluation of the *Sales Amount* measure outside of the iteration, in the original evaluation context. The original filter context is filtering Cameras and camcorders. Therefore, the result is the amount of sales of cameras and camcorders.
- The value of 2 is the result of finding the place of *Sales Amount* of cameras and camcorders in the sorted lookup table.

You might have noticed that at the grand total, *RANKX* shows 1. This value does not make any sense from a human point of view because a ranking should not have any total at all. Nevertheless, this value is the result of the same process of evaluation, which at the grand total always shows a meaningless value. In Figure 7-12 you can see the evaluation process at the grand total.



Category	Sales Amount	Rank Cat on Sales
Audio	384,518.16	6
Cameras and camcorders	7,192,581.95	2
Cell phones	1,604,610.26	5
Computers	6,741,548.73	3
Games and Toys	360,652.81	7
Home Appliances	9,600,457.04	1
Music, Movies and Audio Books	314,206.74	8
TV and Video	4,392,768.29	4
Total	**30,591,343.98**	**1**

FIGURE 7-12 The grand total always shows 1 if sorting of the lookup table is descending.

The value computed during step 2 is the grand total of sales, which is always greater than the sum of individual categories. Thus, the value shown at the grand total is not a bug or a defect; it is the standard *RANKX* behavior that loses its intended meaning at the grand total level. The correct way of handling the total is to hide it by using DAX code. Indeed, the ranking of a category against all other categories has meaning if (and only if) the current filter context only filters one category. Consequently, a better formulation of the measure relies on *HASONEVALUE* in order to avoid computing the ranking in a filter context that produces a meaningless result:

```
Rank Cat on Sales :=
IF (
    HASONEVALUE ( 'Product'[Category] ),
    RANKX (
        ALL ( 'Product'[Category] ),
        [Sales Amount]
    )
)
```

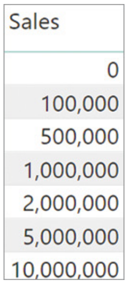
This code produces a blank whenever there are multiple categories in the current filter context, removing the total row. Whenever one uses *RANKX* or, in more general terms, whenever the measure computed depends on specific characteristics of the filter context, one should protect the measure with a conditional expression that ensures that the calculation only happens when it should, providing a blank or an error message in any other case. This is exactly what the previous measure does.

As we mentioned earlier *RANKX* accepts many arguments, not only the first two. There are three remaining arguments, which we introduce here. We describe them later in this section.

- The third parameter is the value expression, which might be useful when different expressions are being used to evaluate respectively the lookup table and the value to use for the ranking.
- The fourth parameter is the sort order of the lookup table. It can be *ASC* or *DESC*. The default is *DESC*, with the highest values on top—that is, higher value results in lower ranking.
- The fifth parameter defines how to compute values in case of ties. It can be *DENSE* or *SKIP*. If it is *DENSE*, then ties are removed from the lookup table; otherwise they are kept.

Let us describe the remaining parameters with some examples.

The third parameter is useful whenever one needs to use a different expression respectively to build the lookup table and to compute the value to rank. For example, consider the requirement of a custom table for the ranking, like the one depicted in Figure 7-13.



Sales
0
100,000
500,000
1,000,000
2,000,000
5,000,000
10,000,000

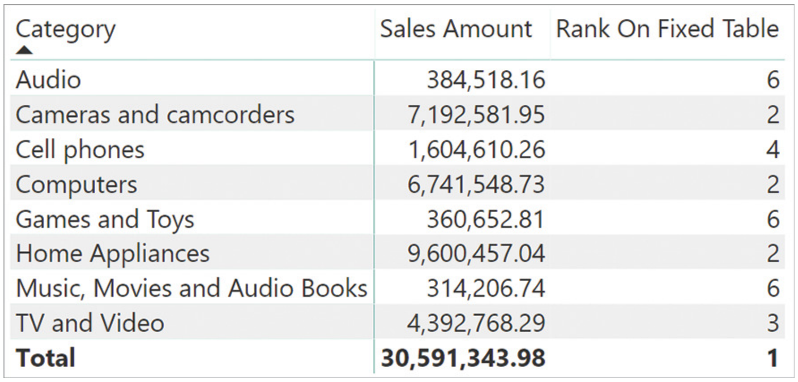
FIGURE 7-13 Instead of building a dynamic lookup table, one might need to use a fixed lookup table.

If one wants to use this table to compute the lookup table, then the expression used to build it should be different from the *Sales Amount* measure. In such a case, the third parameter becomes useful. To rank the sales amount against this specific lookup table—which is named *Sales Ranking*—the code is the following:

```
Rank On Fixed Table :=
RANKX (
    'Sales Ranking',
    'Sales Ranking'[Sales],
    [Sales Amount]
)
```

In this case, the lookup table is built by getting the value of '*Sales Ranking'[Sale]* in the row context of *Sales Ranking*. Once the lookup table is built, *RANKX* evaluates *[Sales Amount]* in the original evaluation context.

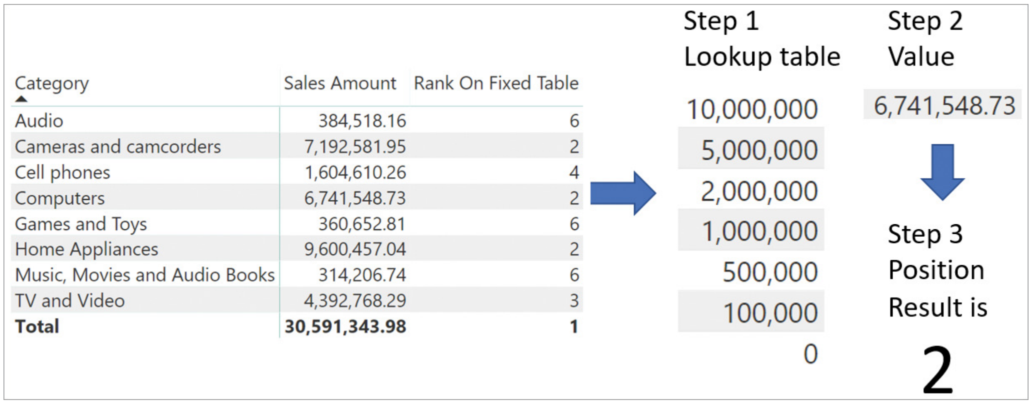
The result of this calculation is visible in Figure 7-14.



Category	Sales Amount	Rank On Fixed Table
Audio	384,518.16	6
Cameras and camcorders	7,192,581.95	2
Cell phones	1,604,610.26	4
Computers	6,741,548.73	2
Games and Toys	360,652.81	6
Home Appliances	9,600,457.04	2
Music, Movies and Audio Books	314,206.74	6
TV and Video	4,392,768.29	3
Total	**30,591,343.98**	**1**

FIGURE 7-14 *Rank On Fixed Table* ranks *Sales Amount* against the fixed *Sales Ranking* table.

The full process is depicted in Figure 7-15, where you can also appreciate that the lookup table is sorted before being used.



Category	Sales Amount	Rank On Fixed Table
Audio	384,518.16	6
Cameras and camcorders	7,192,581.95	2
Cell phones	1,604,610.26	4
Computers	6,741,548.73	2
Games and Toys	360,652.81	6
Home Appliances	9,600,457.04	2
Music, Movies and Audio Books	314,206.74	6
TV and Video	4,392,768.29	3
Total	**30,591,343.98**	**1**

FIGURE 7-15 When using a fixed lookup table, the expression used to build the lookup table is different from the expression used for step 2.

The fourth parameter can be *ASC* or *DESC*. It changes the sort order of the lookup table. By default it is *DESC*, meaning that a lower ranking is assigned to the highest value. If one uses *ASC*, then the lower value will be assigned the lower ranking because the lookup table is sorted the opposite way.

The fifth parameter, on the other hand, is useful in the presence of ties. To introduce ties in the calculation, we use a different measure—*Rounded Sales*. *Rounded Sales* rounds values to the nearest multiple of one million, and we will slice it by brand:

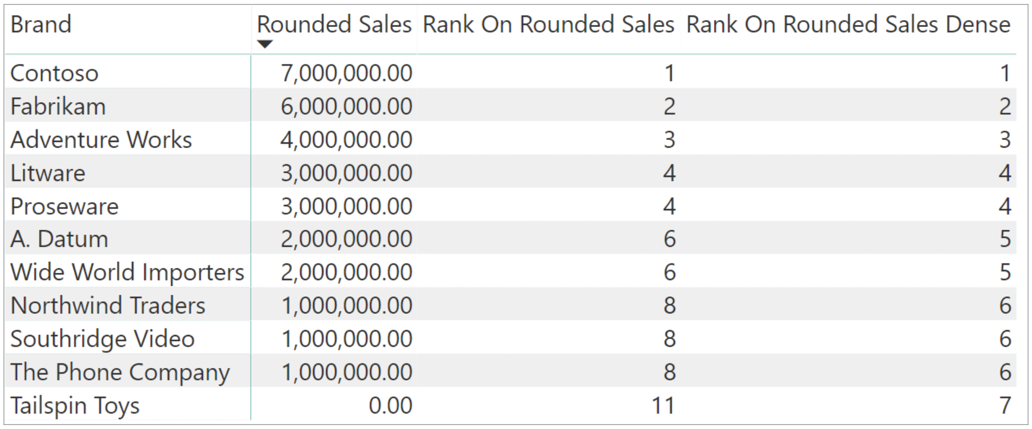
```
Rounded Sales := MROUND ( [Sales Amount], 1000000 )
```

Then, we define two different rankings: One uses the default ranking (which is *SKIP*), whereas the other one uses *DENSE* for the ranking:

```
Rank On Rounded Sales :=
RANKX (
    ALL ( 'Product'[Brand] ),
    [Rounded Sales]
)

Rank On Rounded Sales Dense :=
RANKX (
    ALL ( 'Product'[Brand] ),
    [Rounded Sales],
    ,
    ,
    DENSE
)
```

The result of the two measures is different. In fact, the default behavior considers the number of ties and it increases the ranking accordingly. When using *DENSE*, the ranking increases by one regardless of ties. You can appreciate the different result in Figure 7-16.



Brand	Rounded Sales	Rank On Rounded Sales	Rank On Rounded Sales Dense
Contoso	7,000,000.00	1	1
Fabrikam	6,000,000.00	2	2
Adventure Works	4,000,000.00	3	3
Litware	3,000,000.00	4	4
Proseware	3,000,000.00	4	4
A. Datum	2,000,000.00	6	5
Wide World Importers	2,000,000.00	6	5
Northwind Traders	1,000,000.00	8	6
Southridge Video	1,000,000.00	8	6
The Phone Company	1,000,000.00	8	6
Tailspin Toys	0.00	11	7

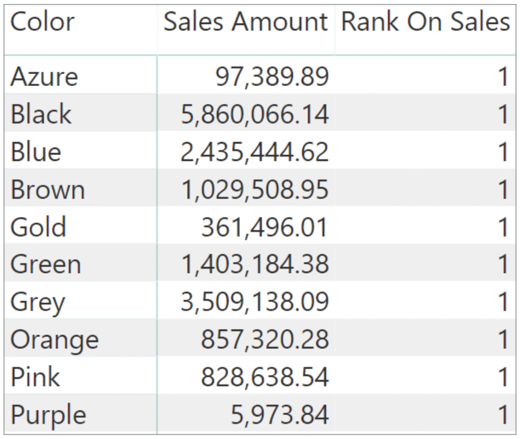
FIGURE 7-16 Using *DENSE* or *SKIP* produces different ranking values in the presence of ties in the lookup table.

Basically, *DENSE* performs a *DISTINCT* on the lookup table before using it. *SKIP* does not, and it uses the lookup table as it is generated during the iteration.

When using *RANKX,* it is important to consider which table to use as the first parameter to obtain the desired result. In the previous queries, it was necessary to specify *ALL (Product[Brand])* because we wanted to obtain the ranking of each brand. For brevity, we omitted the usual test with *HASONEVALUE*. In practice you should never skip it; otherwise the measure is at risk of computing unexpected results. For example, a measure like the following one produces an error if not used in a report that slices by Brand:

```
Rank On Sales :=
RANKX (
    ALL ( 'Product'[Brand] ),
    [Sales Amount]
)
```

In Figure 7-17 we slice the measure by product color and the result is always 1.



Color	Sales Amount	Rank On Sales
Azure	97,389.89	1
Black	5,860,066.14	1
Blue	2,435,444.62	1
Brown	1,029,508.95	1
Gold	361,496.01	1
Green	1,403,184.38	1
Grey	3,509,138.09	1
Orange	857,320.28	1
Pink	828,638.54	1
Purple	5,973.84	1

FIGURE 7-17 A ranking by brand produces unexpected results if sliced by *Color*.

The reason is that the lookup table contains the sales amount sliced by brand and by color, whereas the values to search in the lookup table contain the total only by color. As such, the total by color will always be larger than any of its subsets by brand, resulting in a ranking of 1. Adding the protection code with *IF HASONEVALUE* ensures that—if the evaluation context does not filter a single brand—the result will be blank.

Finally, *ALLSELECTED* is oftentimes used with *RANKX*. If a user performs a selection of some brands out of the entire set of brands, ranking over *ALL* might produce gaps in the ranking. This is because *ALL* returns all the brands, regardless of the filter coming from the slicer. For example, consider the following measures:

```
Rank On Selected Brands :=
RANKX (
    ALLSELECTED ( 'Product'[Brand] ),
    [Sales Amount]
)

Rank On All Brands :=
RANKX (
    ALL ( 'Product'[Brand] ),
    [Sales Amount]
)
```

In Figure 7-18, you can see the comparison between the two measures in the presence of a slicer filtering certain brands.