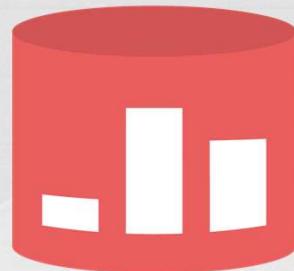


www.sqlbi.com



sqlbi

Microsoft Partner

Gold Business Intelligence
Gold Data Platform

SSAS
MAESTRO
by Microsoft





MASTERING
DAX
Workshop



We write
Books



We teach
Courses



We provide
Consulting

- Remote Consulting
- Power BI/SSAS Optimization
- BI Architectural Review
- On-Site Consulting
- Custom Training & Mentoring

We are recognized
BI Experts



The DAX language

- Language of
 - Power BI
 - Analysis Services Tabular
 - Power Pivot
- DAX is simple, but it is not easy
- New programming concepts and patterns

Content

- Introduction to DAX
- Table functions
- Evaluation contexts
- CALCULATE
- Advanced evaluation context
- Working with iterators
- Building a date table
- Time intelligence
- Hierarchies
- Working with tables and queries
- Advanced filter context
- Advanced relationships

Introduction to the DAX language

Introduction to DAX



What is DAX?

- Programming language
 - Power BI
 - Analysis Services Tabular
 - Power Pivot
- Resembles Excel
 - Because it was born with PowerPivot
 - Important differences
 - No concept of «row» and «column»
 - Different type system
- Many new functions
- Designed for data models and business

Functional language

DAX is a functional language, the execution flows with function calls, here is an example of a DAX formula.

```
=SUMX (
    FILTER (
        VALUES ( 'Date'[Year] ),
        'Date'[Year] < 2005
    ),
    IF (
        'Date'[Year] >= 2000,
        [Sales Amount] * 100,
        [Sales Amount] * 90
    )
)
```

If it is not formatted, it is not DAX

Code formatting is of paramount importance in DAX.

```
=SUMX(FILTER(VALUES('Date'[Year]), 'Date'[Year]<2005), IF('Date'[Year]>=2000, [Sales Amount]*100, [Sales Amount]*90))
```

```
=SUMX (
    FILTER (
        VALUES ( 'Date'[Year] ),
        'Date'[Year] < 2005
    ),
    IF (
        'Date'[Year] >= 2000,
        [Sales Amount] * 100,
        [Sales Amount] * 90
    )
)
```

www.daxformatter.com



DAX data types

- Numeric types
 - Integer (64 bit)
 - Decimal (floating point)
 - Currency (money)
 - Date (DateTime)
 - TRUE / FALSE (Boolean)
- Other types
 - String
 - Binary Objects

Format strings are not
data types!

DAX type handling

- Operator Overloading
 - Operators are not strongly typed
 - The result depends on the inputs
- Example:
 - "5" + "4" = 9
 - 5 & 4 = "54"
- Conversion happens when needed
 - Pay attention to undesired conversions

DateTime****

- Floating point value
- Integer part
 - Number of days after December, 30, 1899
- Decimal part
 - Seconds: $1 / (24 * 60 * 60)$
- DateTime Expressions
 - Date + 1 = The day after
 - Date - 1 = The day before

Calculated columns

- Columns computed using DAX
- Always computed row by row
- Product[Price] means
 - The value of the Price column (explicit)
 - In the Product table (explicit, optional)
 - For the current row (implicit)
 - Different for each row

Column references

- The general format to reference a column
 - 'TableName'[ColumnName]
- Quotes can be omitted
 - If TableName does not contain spaces
 - Do it: omit quotes if there are no spaces in table name
- TableName can be omitted
 - Current table is searched for ColumnName
 - **Don't do it, harder to understand**

Calculated columns and measures

- GrossMargin = SalesAmount - ProductCost
 - Calculated column
- GrossMargin% = GrossMargin / SalesAmount
 - Cannot be computed row by row
- Measures needed

$$\sum_{k=0}^n \left(\frac{\text{GrossMargin}}{\text{SalesAmount}} \right) \neq \frac{\sum(\text{GrossMargin})}{\sum(\text{SalesAmount})}$$

Measures

- Written using DAX
- Do not work row by row
- Instead, use tables and aggregators
- Do not have the «current row» concept
- Examples
 - GrossMargin
 - is a calculated column
 - but can be a measure, too
 - GrossMargin %
 - must be a measure

Naming convention

- Measures should not belong to a table
 - Avoid table name
 - [Margin%] instead of Sales[Margin%]
 - Easier to move to another table
 - Easier to identify as a measure
- Use this syntax when to reference:
 - Calculated columns → Table[Column]
 - Measures → [Measure]

Measures vs calculated columns

- Use a column when
 - Need to slice or filter on the value
- Use a measure
 - Calculate percentages
 - Calculate ratios
 - Need complex aggregations
- Space and CPU usage
 - Columns consume memory in the model
 - Measures consume CPU at query time

Aggregation functions

- Useful to aggregate values
 - SUM
 - AVERAGE
 - MIN
 - MAX
- Aggregate only one column
 - SUM (Orders[Price])
 - **SUM (Orders[Price] * Orders[Quantity])**

The «X» aggregation functions

- Iterators: useful to aggregate formulas
 - SUMX
 - AVERAGEX
 - MINX
 - MAXX
- Iterate over the table and evaluate the expression for each row
- Always receive two parameters
 - Table to iterate
 - Formula to evaluate for each row

Example of SUMX

For each row in the Sales table, evaluates the formula, then sum up all the results.
Inside the formula, there is a «current row».

```
SUMX (Sales,  
      Sales[Price] * Sales[Quantity])  
)
```

City	Channel	Color	Size	Quantity	Price
Paris	Store	Red	Large	1	15
Paris	Store	Red	Small	2	13
Torino	Store	Green	Large	4	11
New York	Store	Green	Small	8	9
	Internet	Red	Large	16	7
	Internet	Red	Small	32	5
	Internet	Green	Large	64	3
	Internet	Green	Small	128	1

$$\begin{aligned}1 \times 15 &= 15 \\2 \times 13 &= 26 \\4 \times 11 &= 44 \\8 \times 9 &= 72 \\16 \times 7 &= 112 \\32 \times 5 &= 160 \\64 \times 3 &= 192 \\128 \times 1 &= 128\end{aligned}$$

Result = 749

SUM or SUMX?

Actually, SUM is nothing but syntax sugar for SUMX

```
--  
-- This is the compact format for a SUM  
--  
SUM ( Sales[Quantity] )  
  
--  
-- Internally, this is translated into  
--  
SUMX (  
    Sales,  
    Sales[Quantity]  
)
```

Counting values

- Useful to count values
 - COUNT/COUNTA (counts anything but blanks)
 - COUNTBLANK (counts blanks and empty strings)
 - COUNTROWS (rows in a table)
 - DISTINCTCOUNT (performs distinct count)

Conditional functions

- Provide Boolean logic
 - AND
 - OR
 - NOT
 - IF
- AND / OR can be expressed with operators:

AND (A, B) = A && B

OR (A, B) = A || B

IN operator

- Verify if the result of an expression is included in a list of values:

```
Customer[State] IN { "WA", "NY", "CA" }
```

- It would require multiple OR conditions otherwise:

```
Customer[State] = "WA"  
| | Customer[State] = "NY"  
| | Customer[State] = "CA"
```

The SWITCH function

Makes it easier to perform nested IF.
Internally, it is converted into a set of nested IF.

```
SizeDesc =  
  
SWITCH (  
    Product[Size],  
    "S", "Small",  
    "M", "Medium",  
    "L", "Large",  
    "XL", "Extra Large",  
    "Other"  
)
```

Switch to perform CASE WHEN

Creative usage of SWITCH might be very useful.

```
DiscountPct =  
  
SWITCH (  
    TRUE (),  
    Product[Size] = "S", 0.5,  
    AND ( Product[Size] = "L", Product[Price] < 100 ), 0.2,  
    Product[Size] = "L", 0.35,  
    0  
)
```

MAX and MIN

Used both as aggregators and to compare values.

```
--  
-- Computes the maximum of sales amount  
--  
MAX ( Sales[SalesAmount] ) MAXX ( Sales, Sales[SalesAmount] )
```

```
--  
-- Computes the maximum between amount and listprice  
--  
MAX ( Sales[Amount], Sales[ListPrice] )
```

```
IF (  
    Sales[Amount] > Sales[ListPrice],  
    Sales[Amount],  
    Sales[ListPrice]  
)
```

Handling errors

- SUMX (Sales, Sales[Quantity])
- Fails if Sales[Quantity] is a string and cannot be converted
- Causes of errors
 - Conversion errors
 - Arithmetical operations
 - Empty or missing values

ISERROR (Expression)

Evaluates Expression and returns TRUE or FALSE, depending on the presence of an error during evaluation.

```
ISERROR (
    Sales[GrossMargin] / Sales[Amount]
)
```

IFERROR (Expression, Alternative)

In case of an error evaluating Expression it returns Alternative.

Useful to avoid writing the same expression twice.

```
IFERROR (
    Sales[GrossMargin] / Sales[Amount],
    BLANK ()
)
```

The DIVIDE function

Divide is useful to avoid using IF inside an expression to check for zero denominators.
It is also faster than using IF.

```
IF (
    Sales[SalesAmount] <> 0,
    Sales[GrossMargin] / Sales[SalesAmount],
    0
)
```

You can write it better with DIVIDE

```
DIVIDE (
    Sales[GrossMargin],
    Sales[SalesAmount],
    0
)
```

Using variables

Very useful to avoid repeating subexpressions in your code.

```
VAR  
    TotalQuantity = SUM ( Sales[Quantity] )  
  
RETURN  
  
IF (  
    TotalQuantity > 1000,  
    TotalQuantity * 0.95,  
    TotalQuantity * 1.25  
)
```

Mathematical functions

- Provide... math
 - ABS, EXP
 - FACT, LN
 - LOG, LOG10
 - MOD, PI
 - POWER, QUOTIENT
 - SIGN, SQRT
- Work exactly as you would expect ☺

Rounding functions

- Many different rounding functions:
 - FLOOR (Value, 0.01), TRUNC (Value, 2)
 - ROUNDDOWN (Value, 2)
 - MROUND (Value, 0.01), ROUND (Value, 2)
 - CEILING (Value, 0.01), ISO.CEILING (Value, 2)
 - ROUNDUP (Value, 2)
- Cast functions – they return a specific data type
 - INT (Value)
 - CURRENCY (Value)

Text functions

- Very similar to Excel ones
 - CONCATENATE,
 - FIND, FIXED, FORMAT,
 - LEFT, LEN, LOWER, MID,
 - REPLACE, REPT, RIGHT,
 - SEARCH, SUBSTITUTE, TRIM,
 - UPPER, VALUE, EXACT
 - CONCATENATE, CONCATENATEX

Date functions

- Many useful functions
 - DATE, DATEVALUE, DAY, EDATE,
 - EOMONTH, HOUR, MINUTE,
 - MONTH, NOW, SECOND, TIME,
 - TIMEVALUE, TODAY, WEEKDAY,
 - WEEKNUM, YEAR, YEARFRAC
- Time Intelligence functions will be covered later

Relational functions

- RELATED
 - Follows relationships and returns the value of a column
- RELATEDTABLE
 - Follows relationships and returns all the rows in relationship with the current one
- It doesn't matter how long the chain of relationships is
 - But all the relationships must be in the same direction

First steps with DAX



The goal of this lab is to gain some confidence with the basics of DAX.

Start with a simple model from the Contoso database and author your first DAX formulas.

Refer to [lab number 1](#) on the hands-on manual.

Some functions return tables instead of values

Table Functions



Table functions

- Basic functions that work on full tables
 - FILTER
 - ALL
 - VALUES
 - DISTINCT
 - RELATEDTABLE
- They are used very often in DAX – their knowledge is required
- Their result is often used in other functions
- They can be combined together to form complex expressions
- We will discover many other table functions later in the course

Filtering a table

City	Channel	Color	Size	Quantity	Price
Paris	Store	Red	Large	1	15
Paris	Store	Red	Small	2	13
Torino	Store	Green	Large	4	11
New York	Store	Green	Small	8	9
	Internet	Red	Large	16	7
	Internet	Red	Small	32	5
	Internet	Green	Large	64	3
	Internet	Green	Small	120	1

```
SUMX(   
    FILTER(   
        Orders,   
        Orders[Price] > 1   
    ),   
    Orders[Quantity] * Orders[Price]   
)
```

The screenshot shows a Microsoft Excel spreadsheet with a PivotTable and a PowerPivot Field List.

PivotTable:

	A	B	C	D	E	F	G
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							
11							
12							
13							

PowerPivot Field List:

- Choose fields to add to report:
- Search:
- Orders
 - City
 - Channel
 - Color
 - Size
 - Quantity
 - Price
 - Amount
 - CalcAmount

PivotTable Data:

	CalcAmount	Column Labels				
Row Labels		Large	Small	Grand Total		
Green			192		192	
Red			112	160	272	
Grand Total			304	160	464	

The FILTER function

- FILTER
 - Adds a new condition
 - Restricts the number of rows of a table
 - Returns a table
 - Can be iterated by an «X» function
- Needs a table as input
- The input can be another FILTER

Ignoring filters

City	Channel	Color	Size	Quantity	Price
Paris	Store	Red	Large	1	15
Paris	Store	Red	Small	2	13
Torino	Store	Green	Large	4	11
New York	Store	Green	Small	8	9
	Internet	Red	Large	16	7
	Internet	Red	Small	32	5
	Internet	Green	Large	64	3
	Internet	Green	Small	128	1

```
SUMX (  
    ALL ( Orders ),  
    Orders[Quantity] * Orders[Price]  
)
```

The screenshot shows a Microsoft Excel spreadsheet with a PivotTable and a PowerPivot Field List.

PivotTable Data:

Column Labels	Large	Small	Total	All Amount	Total Sum of Amount
Row Labels	All Amount	Sum of Amount	All Amount	Sum of Amount	
Green	749	192	749	128	320
Red	749	112	749	160	272
Grand Total	749	304	749	288	592

PowerPivot Field List:

- Orders
 - City
 - Channel
 - Color
 - Size
 - Quantity
 - Price
 - Amount
 - All Amount

The ALL function

- ALL
 - Returns all the rows of a table
 - Ignores any filter
 - Returns a table
 - That can be iterated by an «X» function
- Needs a table as input
- Can be used with a single column
 - **ALL (Customers[CustomerName])**
 - The result contains a table with one column

ALL with many columns

Returns a table with all the values of all the columns passed as parameters.

```
COUNTROWS (
    ALL (
        Orders[Channel],
        Orders[Color],
        Orders[Size]
    )
)
```

Columns of the same table

ALLEXCEPT

Returns a table with all existing combinations of the given columns.

```
ALL (
    Orders[Channel],
    Orders[Color],
    Orders[Size],
    Orders[Quantity],
    Orders[Price],
    Orders[Amount]
)
```

Orders[City] not listed here

```
ALLEXCEPT ( Orders, Orders[City] )
```

Mixing filters

- Table functions can be mixed
- Each one requires a table
- Each one returns a table
- **FILTER (ALL (Table), Condition)**
 - Puts a filter over the entire table
 - Ignores the current filter context

Mixing filters

	City	Channel	Color	Size	Quantity	Price
1	Paris	Store	Red	Large	1	15
2	Paris	Store	Red	Small	2	13
3	Torino	Store	Green	Large	4	11
4	New York	Store	Green	Small	8	9
5		Internet	Red	Large	16	7
6		Internet	Red	Small	32	5
7		Internet	Green	Large	64	3
8		Internet	Green	Small	128	1

```
SUMX (
    FILTER (
        ALL( Orders ),
        Orders[Channel] = "Internet"
    ),
    Orders[Quantity] * Orders[Price]
)
```

PivotTable Data:

Row Labels	Column 1		Total		All	Sum of A
	Large	Small	All	Internet		
Green	592	192	592	128	592	320
Red	592	112	592	160	592	272
Grand Total	592	304	592	288	592	592

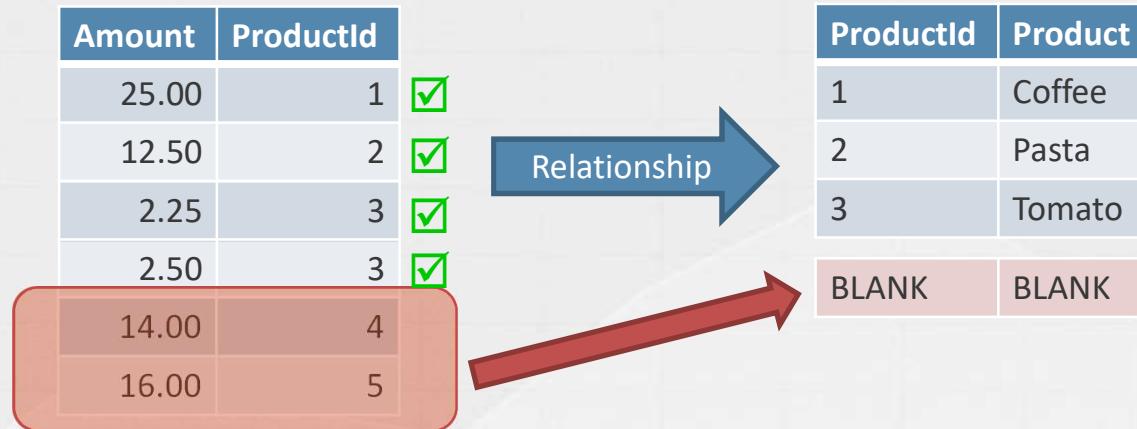
DISTINCT

Returns the unique values of a column, only the ones visible in the current filter context.

NumOfProducts :=

```
COUNTROWS (
    DISTINCT ( Product[ProductCode] )
)
```

How many values for a column?



Tables targets of a relationship might contain an additional blank row, created by DAX to guarantee referential integrity.

VALUES

Returns the unique values of a column, only the ones visible in the current filter context, including the additional blank row if it is visible in the filter context.

```
NumOfProducts :=  
    COUNTROWS (  
        VALUES ( Product[ProductCode] )  
    )
```

ALLNOBLANKROW

ALL returns the additional blank row, if it exists. ALLNOBLANKROW omits it.

```
--  
-- Returns only the existing products  
--  
  
= COUNTROWS (  
    ALLNOBLANKROW ( Products[ProductKey] )  
)
```

Counting different values

Row Labels	CountRowsDistinct	CountRowsValues	CountRowsAll	CountRowsAllNoBlankRow
Coffee	1	1	4	3
Pasta	1	1	4	3
Tomato	1	1	4	3
(blank)		1	4	3
Grand Total	3	4	4	3

Note the difference among

- o DISTINCT
- o VALUES
- o ALL
- o ALLNOBLANKROW

ALLSELECTED

ALLSELECTED returns the elements of a table as they are visible outside of the current visual, be either a pivot table in Excel or a visual in Power BI.

```
Pct All =
```

```
DIVIDE (
    [Sales Amount],
    SUMX (
        ALL ( Sales ),
        Sales[Quantity] * Sales[Net Price]
    )
)
```

Category
Audio
Cameras and camcorders
Cell phones
Computers
Games and Toys
Home Appliances
Music, Movies and Audio...
TV and Video

```
Pct AllSel =
```

```
DIVIDE (
    [Sales Amount],
    SUMX (
        ALLSELECTED ( Sales ),
        Sales[Quantity] * Sales[Net Price]
    )
)
```

Category	Sales Amount	Pct All	Pct AllSel
Cameras and camcorders	842,849,210.26	22.15%	47.25%
Computers	842,569,053.80	22.14%	47.24%
Audio	51,553,689.31	1.35%	2.89%
Games and Toys	46,769,456.76	1.23%	2.62%
Total	1,783,741,410.13	46.87%	100.00%

sqlbi

RELATEDTABLE

Returns a table with all the rows related with the current one.

```
NumOfProducts =  
COUNTRROWS (  
    RELATEDTABLE ( Product )  
)
```

Tables and relationships

The result of table function inherits the relationships of their columns.

```
=SUMX (  
    FILTER (   
        ProductCategory,  
        COUNTROWS (   
            RELATEDTABLE ( Product )  
        ) > 10  
    ),  
    SUMX (   
        RELATEDTABLE ( Sales ),  
        Sales[SalesAmount]  
    )  
)
```

Tables with one row and one column

When a table contains ONE row and ONE column, you can treat it as a scalar value.

```
Sel Category :=
```

```
"You selected: " &
IF (
    HASONEVALUE ( 'Product Category'[Category] ),
    VALUES ( 'Product Category'[Category] ),
    "Multiple values"
)
```

Category
<input checked="" type="checkbox"/> Audio
<input type="checkbox"/> Cameras and camcorders
<input type="checkbox"/> Cell phones
<input type="checkbox"/> Computers
<input type="checkbox"/> Games and Toys
<input type="checkbox"/> Home Appliances
<input type="checkbox"/> Music, Movies and Audio Books
<input type="checkbox"/> TV and Video

You selected: Audio

Sel Category

SELECTEDVALUE

SELECTEDVALUE is a convenient function that simplifies retrieving the value of a column, when only one value is visible.

```
SELECTEDVALUE (
    'Product Category'[Category],
    "Multiple values"
)
```

Equivalent to:

```
IF (
    HASONEVALUE ( 'Product Category'[Category] ),
    VALUES ( 'Product Category'[Category] ),
    "Multiple values"
)
```

ISEMPTY

Checks if a table is empty, faster than an equivalent expression using COUNTROWS.

```
ISEMPTY ( VALUES ( Product[Unit Price] ) )
```

is equivalent to

```
COUNTROWS ( VALUES ( Product[Unit Price] ) ) = 0
```

Table variables

A variable can contain either a scalar value or a table.

Using table variables greatly helps in splitting complex expressions.

```
VAR
```

```
    SalesGreaterThan10 = FILTER ( Sales, Sales[Quantity] > 10 )
```

```
RETURN
```

```
    SUMX (  
        FILTER (   
            SalesGreaterThan10,  
            RELATED ( Product[Color] ) = "Red"  
        ),  
        Sales[Amount]  
    )
```

Table functions



The goal of this lab is to gain some confidence with the handling of table functions, and use their result in more complex expressions.

Please refer to **lab number 2** on the hands-on manual.

Let us take a look at how DAX works

Evaluation Contexts

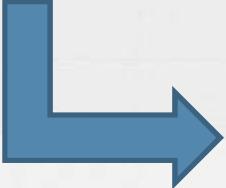


Evaluation contexts

- Evaluation contexts are the pillars of DAX
- Simple concepts, hard to learn
- At the beginning, they looks very easy
- Using them, complexity arises
- The devil is in the details

What is an evaluation context?

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



TotalSales	Row Labels ▾ TotalSales
\$29,358,677.22	
	Black \$8,838,411.96
	Blue \$2,279,096.28
	Multi \$106,470.74
	NA \$435,116.69
	Red \$7,724,330.52
	Silver \$5,113,389.08
	White \$5,106.32
	Yellow \$4,856,755.63
Grand Total	\$29,358,677.22

Numbers are sliced by color, i.e.
the formula is NOT computing
sum of sales, it is computing it
for only a subset of the data
model

The value of a formula
depends on its context

Filter context in a pivot table

The diagram illustrates the components of a PivotTable and their relationship to filter context:

- PivotTable Filter:** A blue box pointing to the "Gender" filter dropdown at the top left of the PivotTable, which is set to "FEMALE".
- Columns:** A blue box pointing to the column headers "Graduate Degree", "High School", "Partial College", "Partial High School", and "Grand Total".
- Rows:** A blue box pointing to the row labels "Black", "Blue", "Multi", and "NA".
- Slicers:** A blue box pointing to the "ProductModel" slicer on the left side, which lists items like "All-Purpose Bike St...", "Bike Wash", "Classic Vest", etc.

ProductModel	TotalSales	Column Labels	Bachelors	Graduate Degree	High School	Partial College	Partial High School	Grand Total
Row Labels								
All-Purpose Bike St...	\$5,142.90			\$3,012.27	\$3,012.27	\$4,481.67	\$1,297.97	\$16,947.08
Bike Wash	\$5,143.50			\$3,238.50	\$2,984.50	\$4,826.00	\$1,270.00	\$17,462.50
Classic Vest	\$2,876.80			\$1,717.09	\$1,672.14	\$2,813.87	\$737.18	\$9,817.08
Cycling Cap	\$8,090.95			\$5,444.01	\$3,913.36	\$6,996.16	\$1,984.74	\$26,429.22
Fender Set - Mount...			Grand Total	\$21,254.15	\$13,411.87	\$11,582.27	\$19,117.70	\$5,289.89
Half-Finger Gloves								\$70,655.88
Hitch Rack - 4 Bike								
HL Mountain Tire								

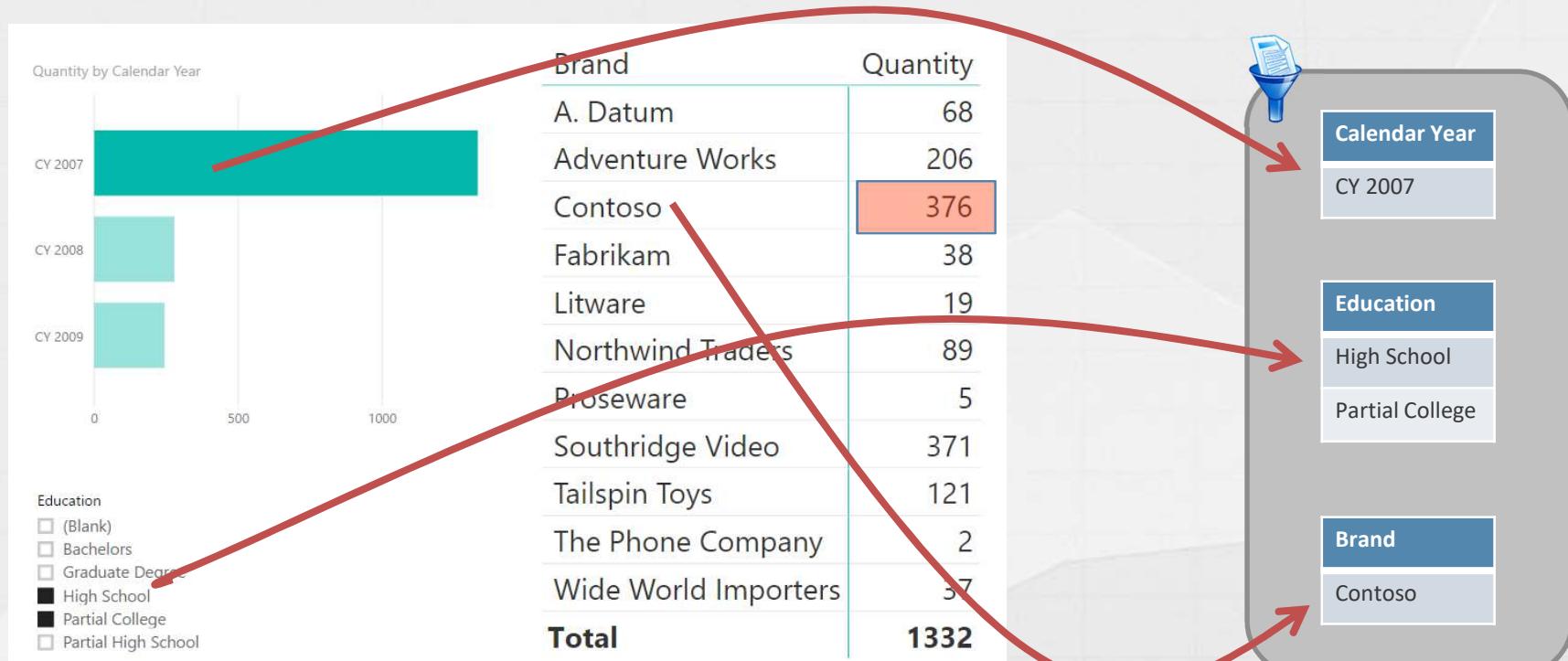
Example of a filter context

The screenshot illustrates a data analysis interface with the following components:

- Data Grid:** A main table showing product details across four categories: City, Channel, Color, Size, Quantity, and Price. The data includes rows for Paris (Store, Red, Large, 1, 15), Paris (Store, Red, Small, 2, 13), Torino (Store, Green, Large, 4, 11), New York (Store, Green, Small, 8, 9), Internet (Red, Large, 16, 7), Internet (Red, Small, 32, 5), Internet (Green, Large, 64, 3), and Internet (Green, Small, 128, 1).
- Filter Panel:** On the left, there are two filter panels:
 - Channel Filter:** Shows options for "Internet" (selected) and "Store".
 - City Filter:** Shows options for "New York" (selected), "Paris", and "Torino".
- Summary Table:** A detailed breakdown of quantities by color and size, with a "Grand Total" at the bottom.

	Sum of Quantity	Column Labels	Row Labels	Large	Small	Grand Total
Green	64				128	192
Red	16				32	48
Grand Total	80	160				240

Filter context in Power BI



Filter context

- Defined by
 - Row Selection
 - Column Selection
 - Report Filters
 - Slicers Selection
- Rows outside of the filter context
 - Are not considered for the computation
- Defined automatically by the client, tool
- Can also be created with specific functions

Row context

- Defined by
 - Calculated column definition
 - Defined automatically for each row
 - Row Iteration functions
 - SUMX, AVERAGEX ...
 - All «X» functions and iterators
 - Defined by the user formulas
- Needed to evaluate column values, it is the concept of "current row"

SUMX (Orders, Orders[Quantity]*Orders[Price])

The diagram illustrates the calculation of the formula `SUMX (Orders, Orders[Quantity]*Orders[Price])` for the Internet channel. It shows a table of order data and the breakdown of the calculation.

City	Channel	Color	Size	Quantity	Price
Paris	Store	Red	Large	1	15
Paris	Store	Red	Small	2	13
Torino	Store	Green	Large	4	11
New York	Store	Green	Small	8	9
	Internet	Red	Large	16	7
	Internet	Red	Small	32	5
	Internet	Green	Large	64	3
	Internet	Green	Small	128	1

The calculation for the Internet channel is as follows:

$$16 \times 7 = 112$$

$$32 \times 5 = 160$$

$$64 \times 3 = 192$$

$$128 \times 1 = 128$$

SUM = 592

The diagram shows the Power BI interface with the following components:

- Filter Panes:** A Channel filter pane with Internet selected and a Store option, and a City filter pane with New York, Paris, and Torino selected.
- Calculated Column:** A table titled "CalcAmount" with "Column Labels" set to "Large". It has three columns: Large, Small, and Grand Total. The data is:

	Large	Small	Grand Total
Green	192	128	320
Red	112	160	272
Grand Total	304	288	592
- Output:** A box labeled "Channel" containing "Internet".

Context errors

- Orders[Quantity] * Orders[Price]
- In a calculated column it works fine
- In a measure it does not work
 - *A single value for column 'Quantity' in table 'Sales' cannot be determined...*
 - A better error message would be:
“you need a row context”

There are always two contexts

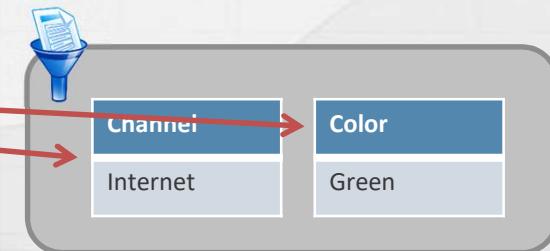
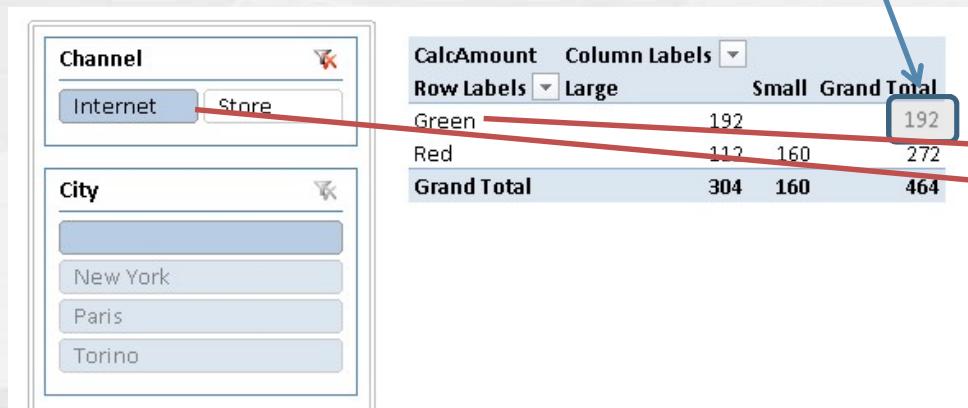
- Filter context
 - Filters tables
 - Might be empty
 - All the tables are visible
 - But this never happens in the real world
- Row context
 - Iterates rows
 - For the rows active in the filter context
 - Might be empty
 - There is no iteration running
- Both are «evaluation contexts»

Filtering a table

City	Channel	Color	Size	Quantity	Price
Paris	Store	Red	Large	1	15
Paris	Store	Red	Small	2	13
Torino	Store	Green	Large	4	11
New York	Store	Green	Small	8	9
	Internet	Red	Large	16	7
	Internet	Red	Small	32	5
	Internet	Green	Large	64	3
	Internet	Green	Small	120	1

```

SUMX(
  FILTER(
    Orders,
    Orders[Price] > 1
  ),
  Orders[Quantity] * Orders[Price]
)
  
```



Ignoring filters

City	Channel	Color	Size	Quantity	Price
Paris	Store	Red	Large	1	15
Paris	Store	Red	Small	2	13
Torino	Store	Green	Large	4	11
New York	Store	Green	Small	8	9
	Internet	Red	Large	16	7
	Internet	Red	Small	32	5
	Internet	Green	Large	64	3
	Internet	Green	Small	128	1

```
SUMX (
    ALL ( Orders ),
    Orders[Quantity] * Orders[Price]
)
```

The diagram illustrates the data flow from a Power BI report to a calculated column in a table.

Power BI Report:

- Filter Panel:** Shows filters for Channel (Internet) and City (New York).
- Data View:** A PivotTable showing the sum of amount for different combinations of Channel, Color, and Size.

Calculated Column:

A calculated column named "Total All Amoun" is defined in a table:

Row Lab	AllAmount	Sum of Amount	AllAmount	Sum of Amount	Total All Amoun	Total Sum of Amoun
Green	749	192	749	128	749	320
Red	749	112	749	160	749	272
Grand Total	749	304	749	288	749	592

Annotations:

- Red arrows point from the selected filters in the Power BI report to the corresponding cells in the PivotTable.
- A red arrow points from the "Total All Amoun" cell in the PivotTable to the calculated column in the table.



Using RELATED in a row context

Starting from a row context, you can use RELATED to access columns in related tables.

```
SUMX (  
    Sales,  
    Sales[Quantity]  
        * RELATED ( Products[ListPrice] )  
        * RELATED ( Categories[Discount] )  
)
```

You need RELATED because
the row context is iterating
the Sales table.

Nesting row contexts

Row contexts can be nested, on the same or on different tables.

```
SUMX (
    Categories,
    SUMX (
        RELATEDTABLE ( Products ),
        SUMX (
            RELATEDTABLE ( Sales )
            ( Sales[Quantity] * Products[ListPrice] ) * Categories[Discount]
        )
    )
)
```

-
- Three row contexts:
- Categories
 - Products of category
 - Sales of product

Ranking by price

- Create a calculated column
- Ranks products by list price
- Most expensive product is ranked 1

ProductKey	ProductName	ListPrice	ListPriceRank
314	Road-150 Red, 56	\$3,578.27	1
313	Road-150 Red, 52	\$3,578.27	1
312	Road-150 Red, 48	\$3,578.27	1
311	Road-150 Red, 44	\$3,578.27	1
310	Road-150 Red, 62	\$3,578.27	1
347	Mountain-100 Si...	\$3,399.99	2
346	Mountain-100 Si...	\$3,399.99	2
345	Mountain-100 Si...	\$3,399.99	2
344	Mountain-100 Si...	\$3,399.99	2
351	Mountain-100 B...	\$3,374.99	3
350	Mountain-100 B...	\$3,374.99	3
349	Mountain-100 B...	\$3,374.99	3
348	Mountain-100 B...	\$3,374.99	3
380	Road-250 Black, ...	\$2,443.35	4
378	Road-250 Black, ...	\$2,443.35	4

Nesting row contexts

When you nest row contexts on the same table, you can use a variable to save the value of the outer row context

```
Products[RankOnPrice] =  
VAR CurrentListPrice = Products[ListPrice]  
RETURN  
COUNTROWS (  
    FILTER (  
        Products,  
        Products[ListPrice] > CurrentListPrice  
    )  
) + 1
```

Row context of the calculated column

Get ListPrice from the outer row context

Row context of the FILTER function

Nesting row contexts

As an alternative, in versions of DAX that do not support variables, you can use the EARLIER function

```
Products[RankOnPrice] =  
  
COUNTRows (  
    FILTER (  
        Products,  
        Products[ListPrice] > EARLIER ( Products[ListPrice] )  
    )  
) + 1
```

Row context of the
calculated column

Row context of the
FILTER function

Get ListPrice from the
outer row context

EARLIER

- Useful when two or more row contexts exist for the same table, if you use an older version of DAX
- Its original name (much better) was **OUTER**
- EARLIER
 - Returns values from the previous row context
 - 2nd parameter: number of steps to jump out
- EARLIEST
 - Returns values from the first row context

Computing the correct rank

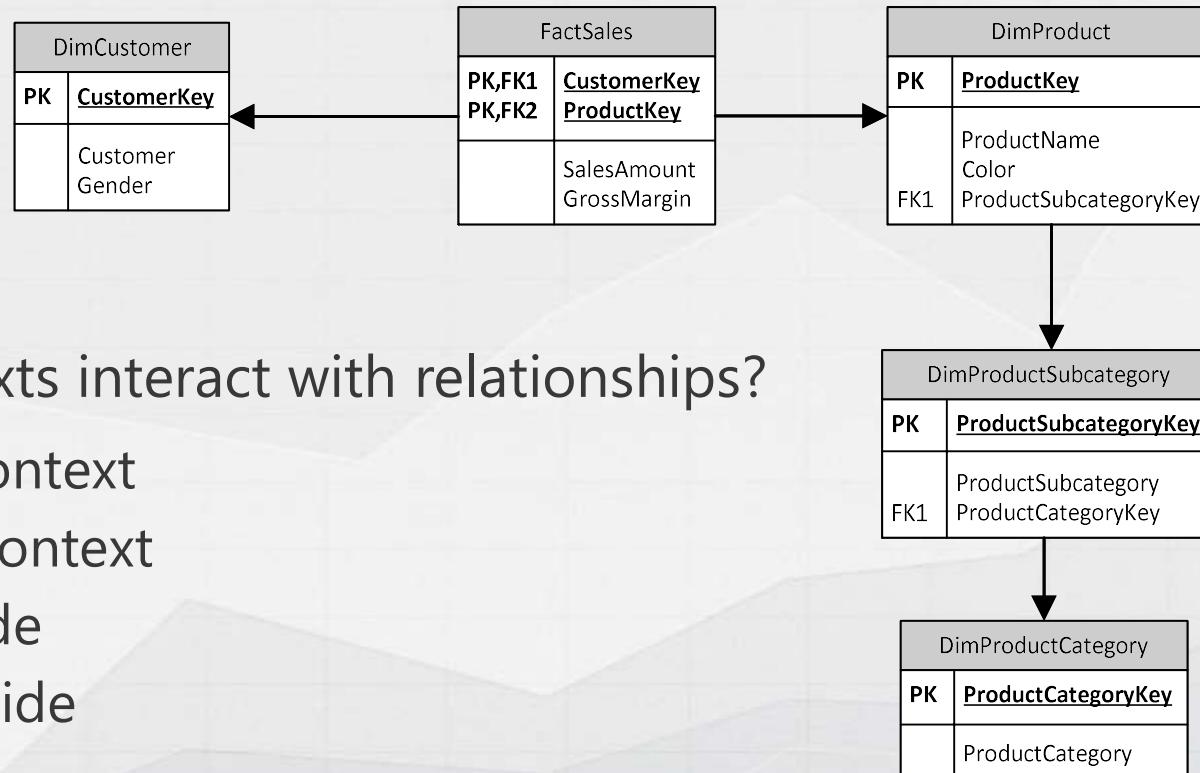
The correct solution requires to rank over the different prices, not the different products. ALL with a column becomes very handy here.

```
Products[RankOnPrice] =  
  
VAR CurrentListPrice = Products[ListPrice]  
VAR AllPrices = ALL ( Products[ListPrice] )  
  
RETURN  
  
COUNTROWS (  
    FILTER (  
        AllPrices,  
        Products[ListPrice] > CurrentListPrice  
    )  
) + 1
```

Evaluation contexts and relationships



Filters and relationships



- Do contexts interact with relationships?
 - Row Context
 - Filter Context
 - One side
 - Many side

Row context – multiple tables

Row Context does not propagate over relationships

		fx =Orders[Amount] * (1 - Channels[Discount])						
Channel	Color	Size	Quantity	Price	Amount	DiscountedAmount	Action	
Core	Red	Large	1	15	15	#ERROR		
Core	Red	Large	1	15	15	#ERROR		
Core	Green	Large	1	15	15	#ERROR		
Core	Green	Large	1	15	15	#ERROR		
Internet	Red	Large	1	15	15	#ERROR		
Internet	Red	Large	1	15	15	#ERROR		
Internet	Green	Large	1	15	15	#ERROR		
Internet	Green	Large	1	15	15	#ERROR		

A single value for column 'Discount' in table 'Channels' cannot be determined. This can happen when a measure formula refers to a column that contains many values without specifying an aggregation such as min, max, count, or sum to get a single result.

RELATED

- RELATED (table[column])
 - Opens a new row context on the target table
 - Following relationships

	City	Chann...	Color	Size	Quantity	Price	Amount	DiscountedAmount
	Paris	Store	Red	Large	1	15	15	14.25
	Paris	Store	Red	Small	2	13	26	24.7
	Torino	Store	Green	Large	4	11	44	41.8
	New York	Store	Green	Small	8	9	72	68.4
		Internet	Red	Large	16	7	112	100.8
		Internet	Red	Small	32	5	160	144
		Internet	Green	Large	64	3	192	172.8
		Internet	Green	Small	128	1	128	115.2

RELATEDTABLE

- RELATEDTABLE (table)
 - Filters the parameter table
 - Returns only rows related with the current one
- It is the companion of RELATED

[OrdersCount]	f x	=COUNTROWS(RELATEDTABLE(Orders))
Cha...	Discount	OrdersCount
Internet	0.1	4
Store	0.05	4

Filter context – many tables

The screenshot illustrates the concept of filter context in PowerPivot, showing how filters applied in one table affect data in other related tables.

PowerPivot Field List:

- Orders:** City, Channel, Color (checked), Size (checked), Quantity, Price, Amount, DiscountedAmount, CalcAmount
- Channels:** Channel, Discount
- Cities:** City, Country, Continent (checked)

Sum of Discounted/ Column Labels:

Row Labels	Large	Small	Grand Total
Green	41.8		41.8
Red	14.25	24.7	38.95
Grand Total	56.05	24.7	80.75

Continent Table:

	A	B	C	D	E	F	G
1							
2							
3							
4							
5			Continent				
6			Europe				
7			North America				
8							
9							
10							
11							
12							
13							
14							
15			Country	Continent			
16			Paris France Europe	Europe			
17			New York USA North America	North America			
18			Torino Italy Europe	Europe			
19			Madrid Spain Europe	Europe			

Channel Table:

	A	B	C	D	E	F	G
1							
2							
3			Channel				
4			Store	Internet			
5							
6							
7							
8							
9							
10							
11							
12							
13							
14							
15							
16							
17							
18							
19							

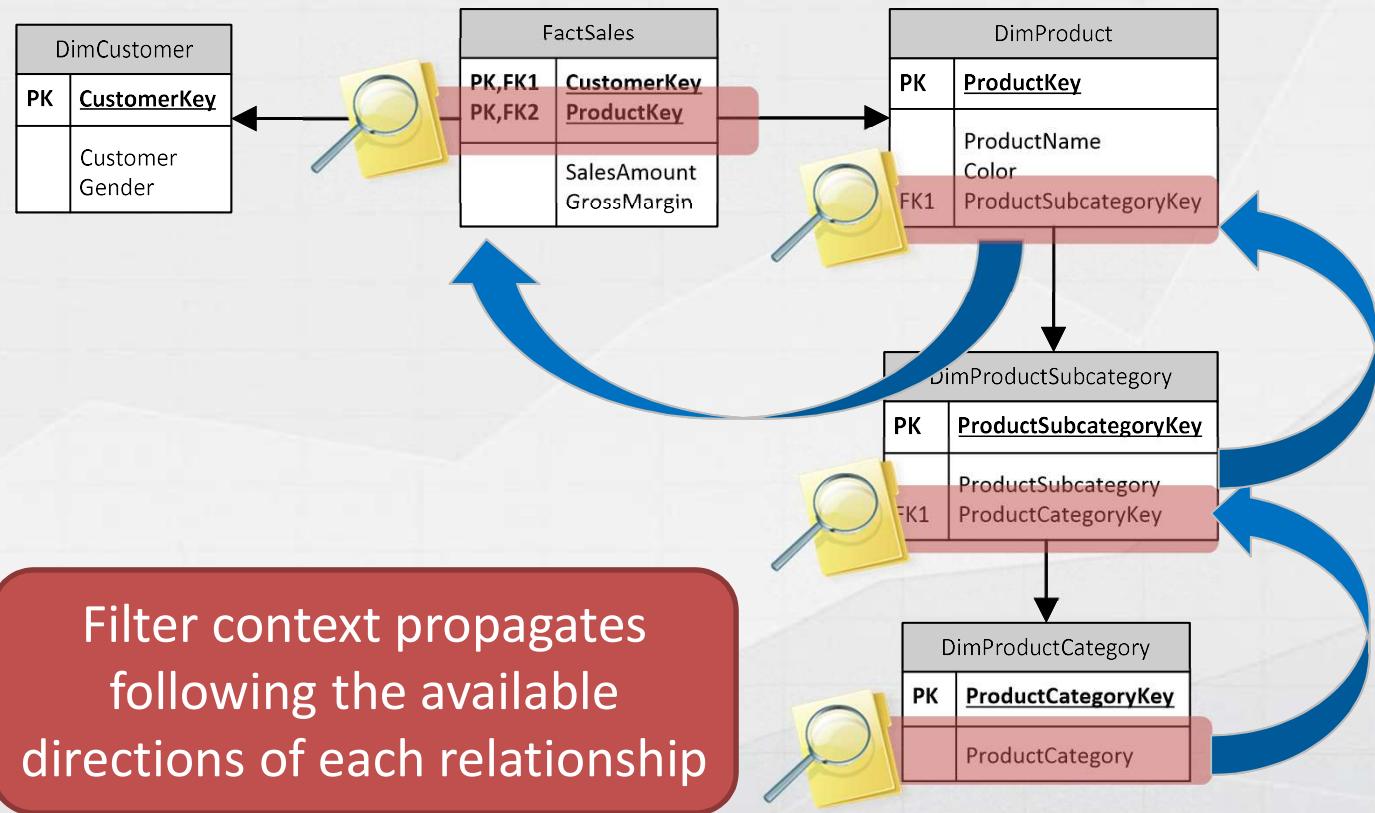
Country Table:

	A	B	C	D	E	F	G
1							
2							
3			Country	Continent			
4			Paris France Europe	Europe			
5			New York USA North America	North America			
6			Torino Italy Europe	Europe			
7			Madrid Spain Europe	Europe			
8							
9							
10							
11							
12							
13							
14							
15							
16							
17							
18							
19							

Orders Table:

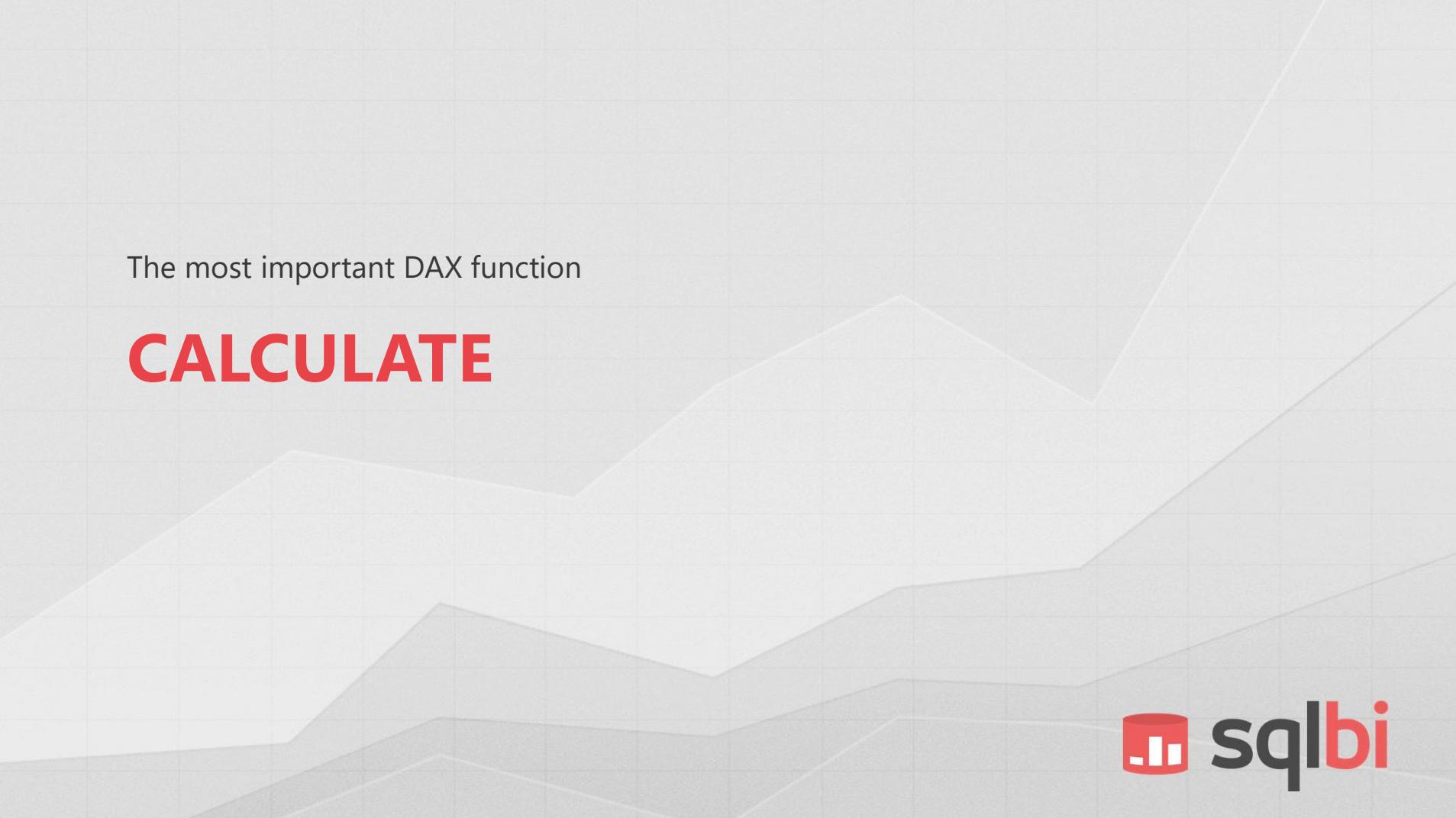
	A	B	C	D	E	F	G
1							
2							
3			Color	Size	Quantity	Price	Amount
4			Red Large	1	15	15	14.25
5			Red Small	2	13	26	24.7
6			Green Large	4	11	44	41.8
7			Green Small	8	9	72	68.4
8			Internet Red Large	16	7	112	100.8
9			Internet Red Small	32	5	160	144
10			Internet Green Large	64	3	192	172.8
11			Internet Green Small	128	1	128	115.2
12							
13							
14							
15							
16							
17							
18							
19							

Filters and relationships



Bidirectional cross-filter

- Choose the propagation of the filter context
 - Single: one to many propagation
 - Both: filter context propagates both ways
- Not available in Excel, only Analysis Services and Power BI
- Beware of several details
 - Performance degradation
 - Filtering is active when the “many” side is cross-filtered, numbers might be hard to read for certain measures
 - Ambiguity might appear in the model



The most important DAX function

CALCULATE



CALCULATE syntax

Filters are evaluated in the outer filter context, then combined together in AND, and finally used to build a new filter context into which DAX evaluates the expression.

```
CALCULATE (
    Expression,
    Filter1,
    ...
    Filtern
)
```

Repeated many times, as needed

The filter parameters are used to modify the existing filter context. They can add, remove or change existing filter

CALCULATE examples

Compute the sum of sales where the price is greater than \$100.00.

```
NumOfBigSales :=
```

```
CALCULATE (
    SUMX ( Sales, Sales[Quantity] * Sales[Net Price] ),
    Sales[Net Price] > 100
)
```

Filter and SUM are
on the same table.
You can obtain the same
result using FILTER

Filters are tables

Each filter is a table.

Boolean expressions are nothing but shortcuts for table expressions.

```
CALCULATE (
    [Sales Amount],
    Sales[Net Price] > 100
)
```

Is equivalent to

```
CALCULATE (
    [Sales Amount],
    FILTER (
        ALL ( Sales[Net Price] ),
        Sales[Net Price] > 100
    )
)
```

Let's learn CALCULATE by using some demo

CALCULATE Examples



CALCULATE examples

Compute the sales amount for all of the product colors, regardless of the user selection.

```
SalesAllColors :=  
  
CALCULATE (  
    [Sales Amount],  
    ALL ( Product[Color] )  
)
```

The condition is the list
of acceptable values

CALCULATE examples

Compute the sales amount for red products, regardless of user selection for color.
The filter context is applied on the entire model, products filter the Sales table, too.

```
SalesRedProducts :=  
  
CALCULATE (  
    [Sales Amount],  
    Product[Color] = "Red"  
)
```

Filter and SUM are
on different tables.
Filter happens because of
filter context propagation

CALCULATE examples

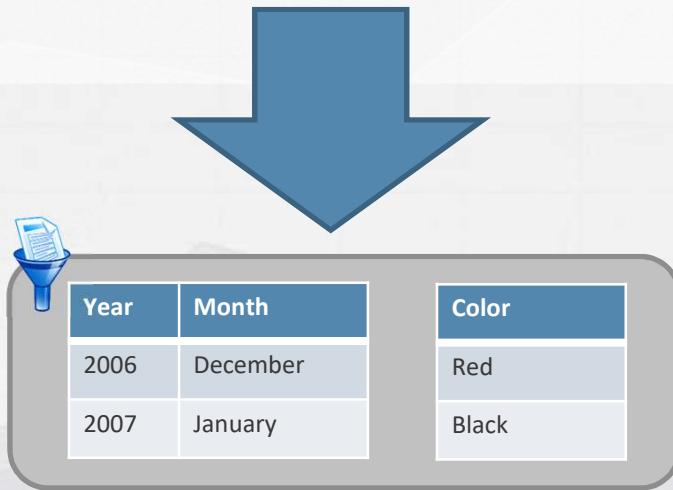
Compute the sales amount for red and blue products, within the user selection for color.

```
SalesTrendyColors :=  
CALCULATE (  
    [Sales Amount],  
    KEEPFILTERS ( Product[Color] IN { "Red", "Blue" } )  
)
```

KEEPFILTERS keeps the previous filter, so that the new filter does not override the previous one

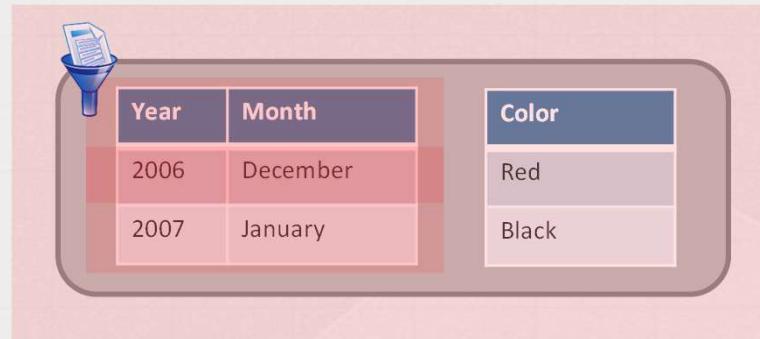
What is a filter context?

```
CALCULATE (
    ...,
    Product[Color] IN { "Red", "Black" },
    FILTER (
        ALL ( Date[Year], Date[Month] ),
        OR (
            AND ( Date[Year] = 2006, Date[Month] = "December" ),
            AND ( Date[Year] = 2007, Date[Month] = "January" )
        )
    )
)
```



Filters are tables.
You can use any table
function to create a
filter in CALCULATE

Filter context definition



Tuple: value for a set of columns

Filter: table of tuples

Filter context: set of filters

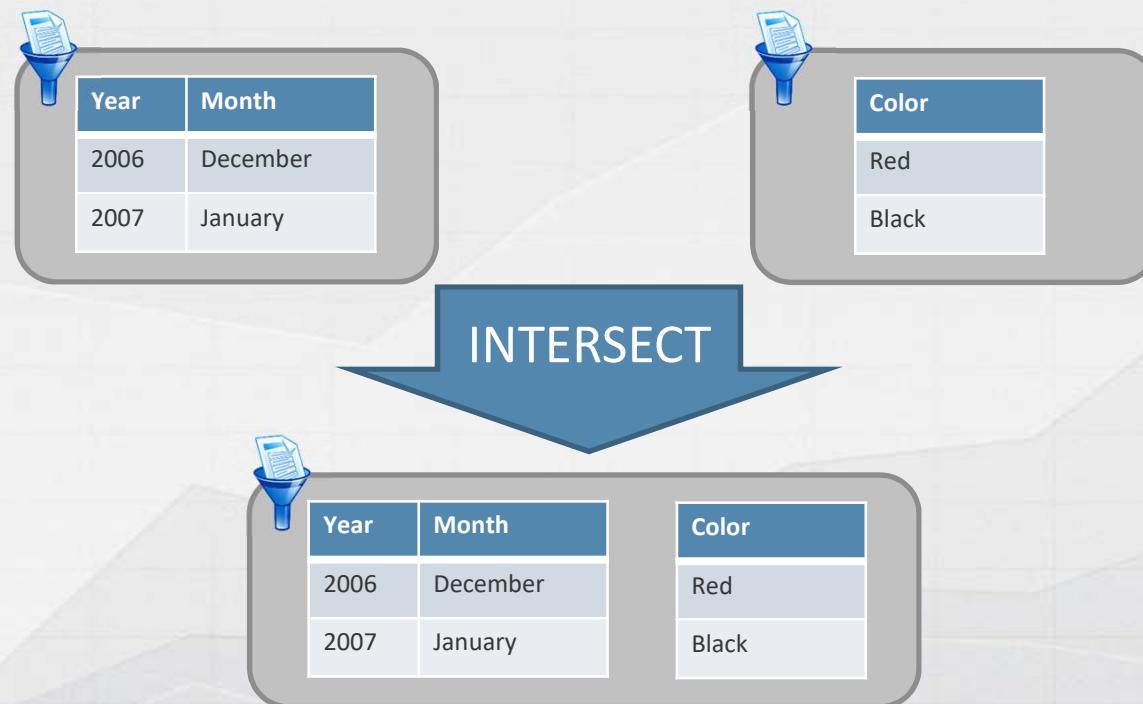
Multiple conditions in CALCULATE

Multiple filter parameters in CALCULATE are intersected, generating a new filter context that uses both filters at the same time.

```
CALCULATE (
    ...,
    Product[Color] IN { "Red", "Black" },
    FILTER (
        ALL ( Date[Year], Date[Month] ),
        OR (
            AND ( Date[Year] = 2006, Date[Month] = "December" ),
            AND ( Date[Year] = 2007, Date[Month] = "January" )
        )
    )
)
```

Intersection of filter context

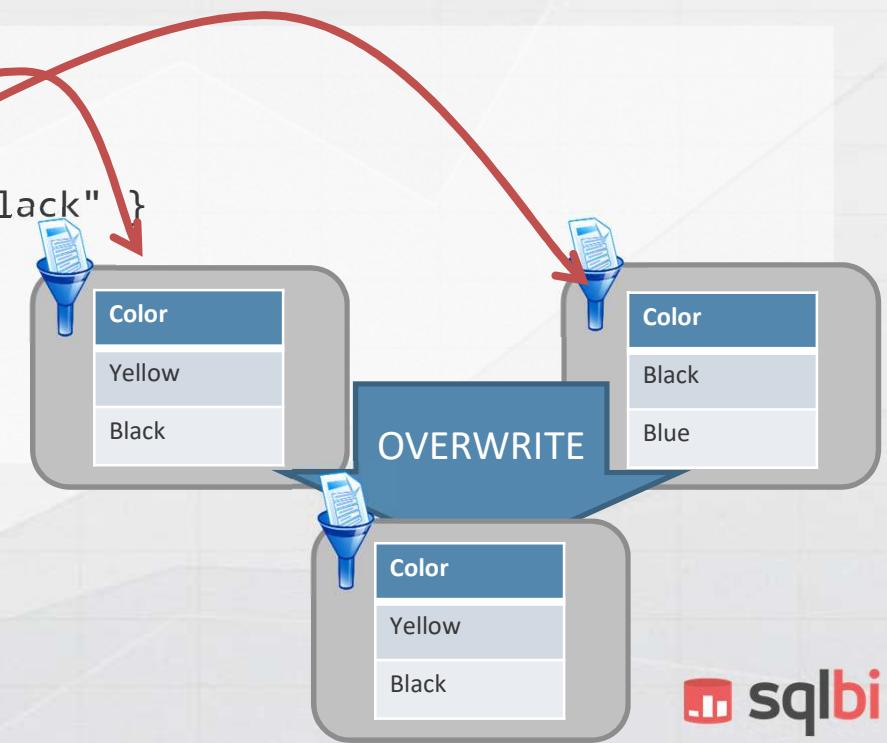
Used by CALCULATE to put filters in AND



Overwriting filter contexts

Nested CALCULATE do not intersect filters, they use another operator, called OVERWRITE.
In fact, the Yellow/Black filter wins against the Black/Blue one, being the innermost.
Yellow/Black overwrites Black/Blue.

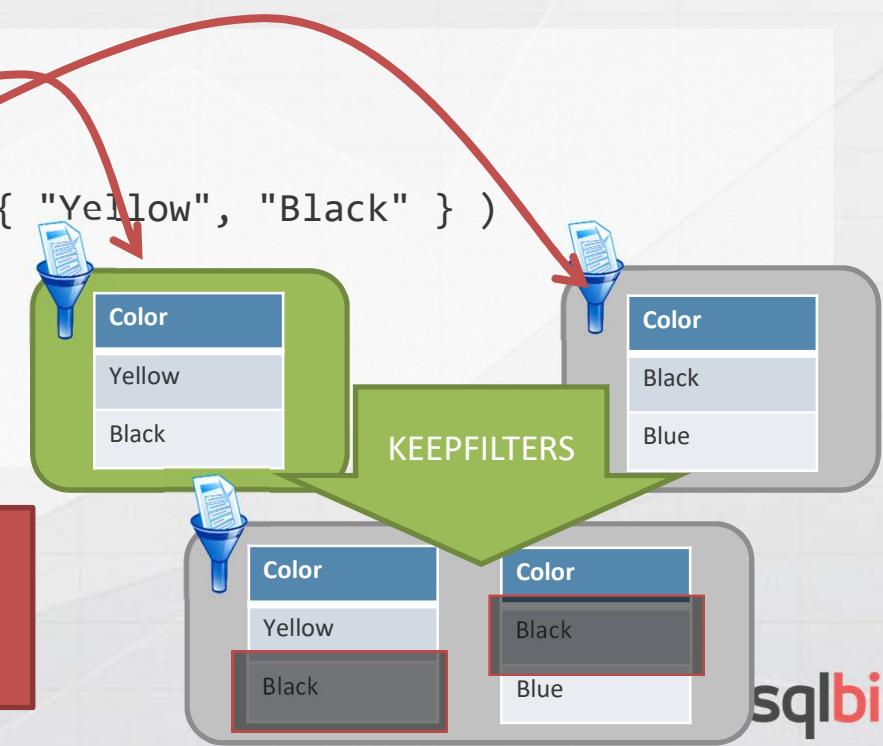
```
CALCULATE (
    CALCULATE (
        ...,
        Product[Color] IN { "Yellow", "Black" }
    ),
    Product[Color] { "Black", "Blue" }
)
```



KEEPFILTERS

KEEPFILTERS retains the previous filters, instead of replacing them.

```
CALCULATE (
    CALCULATE (
        ...,
        KEEPFILTERS ( Product[Color] IN { "Yellow", "Black" } )
    ),
    Product[Color] { "Black", "Blue" }
)
```



CALCULATE operators

- Overwrite a filter context at the individual columns level
- Remove previously existing filters (ALL)
- Add filters (KEEPFILTERS)
- In DAX you work by manipulating filters with the following internal operators:
 - INTERSECT (multiple filters in CALCULATE)
 - OVERWRITE (nested CALCULATE)
 - REMOVEFILTERS (using ALL)
 - ADDFILTER (using KEEPFILTERS)

Use one column only in compact syntax

- Boolean filters can use only one column
 - You cannot mix multiple columns in the same boolean expression

```
CALCULATE (
    [Sales Amount],
    Sales[Quantity] * Sales[Net Price] > 1000
)
```

Filters on tables are bad.
Not only bad... REALLY BAD!

Later we will see the details, for now
just remember: BAD BAD BAD!

```
CALCULATE (
    [Sales Amount]
    FILTER (
        Sales,
        Sales[Quantity] * Sales[Net Price] > 1000
    )
)
```

Overwriting filters on multiple columns

If you want to overwrite a filter on multiple columns, ALL with several columns is your best friend.

```
CALCULATE (
    [Sales Amount],
    FILTER (
        ALL ( Sales[Quantity], Sales[Price] ),
        Sales[Quantity] * Sales[Net Price] > 1000
    )
)
```

KEEPFILTERS might be required

KEEPFILTERS is required to keep the same semantics of the table filter.

```
CALCULATE (
    [Sales Amount],
    KEEPFILTERS (
        FILTER (
            ALL ( Sales[Quantity], Sales[Price] ),
            Sales[Quantity] * Sales[Net Price] > 1000
        )
    )
)
```

```
CALCULATE (
    [Sales Amount],
    FILTER (
        Sales,
        Sales[Quantity] * Sales[Net Price] > 1000
    )
)
```

NEVER filter a
table!

Cannot use aggregators in compact syntax

Boolean filters cannot use aggregators or measures.

```
CALCULATE (
    [Sales Amount],
    Sales[Quantity] < SUM ( Sales[Quantity] ) / 100
)
```

```
VAR TotalQuantity = SUM ( Sales[Quantity] )
RETURN
CALCULATE (
    [Sales Amount],
    Sales[Quantity] < TotalQuantity / 100
)
```

Variables and evaluation contexts

Variables are computed in the evaluation where they are defined, not in the one where they are used. CALCULATE cannot modify the value of a variable because it is already computed.

```
WrongRatio :=  
  
VAR  
    Amt = [Sales Amount]  
RETURN  
    DIVIDE (  
        Amt,  
        CALCULATE ( Amt, ALL ( Sales ) )  
    )
```

Result is always 1

Basic evaluation contexts



- Time to write some DAX code by yourself.
- Next exercise session is focuses on some DAX code that uses evaluation contexts.
- Please refer to **lab number 3** on the hands-on manual

We're not done with evaluation contexts yet!

Advanced evaluation context



CALCULATE MODIFIERS

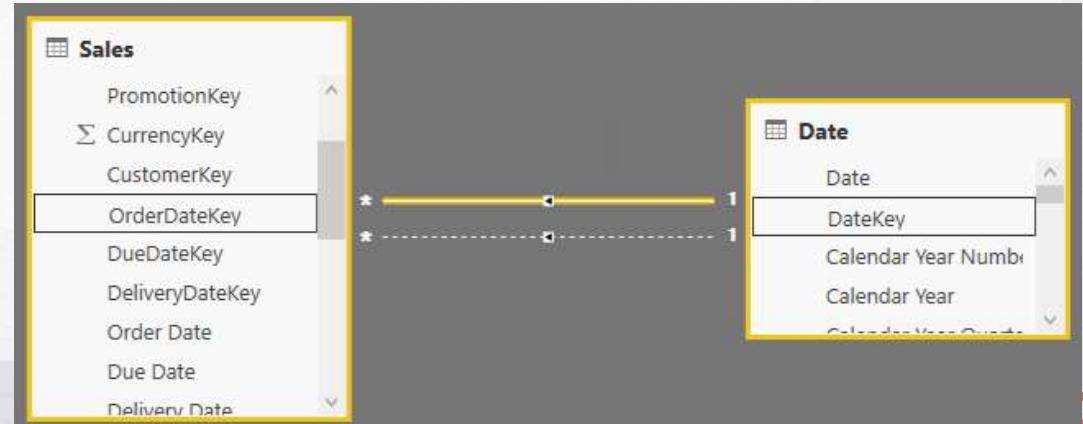
- CALCULATE accepts filters and *modifiers*
- A modifier changes CALCULATE behavior
 - ALL* functions behave as REMOVEFILTER
 - ALL, ALLSELECTED, ALLNOBLANKROW, ALLEXCEPT
 - USERELATIONSHIP activates a relationship
 - CROSSFILTER changes filter propagation of relationships
- There are other filter modifiers:
 - KEEPFILTERS keeps previous filters, avoids OVERWRITE

USERELATIONSHIP

Temporarily activates an inactive relationship. It lasts for the duration of CALCULATE, then the previously active relationship becomes active again.

We will discuss it later in the Time Intelligence section.

```
CALCULATE (
    [Sales Amount],
    USERELATIONSHIP (
        Sales[DueDateKey],
        'Date'[DateKey]
    )
)
```



CROSSFILTER

Changes the direction of a relationship cross-filter just for the evaluation made by a CALCULATE function.

```
Num Of Customers :=
```

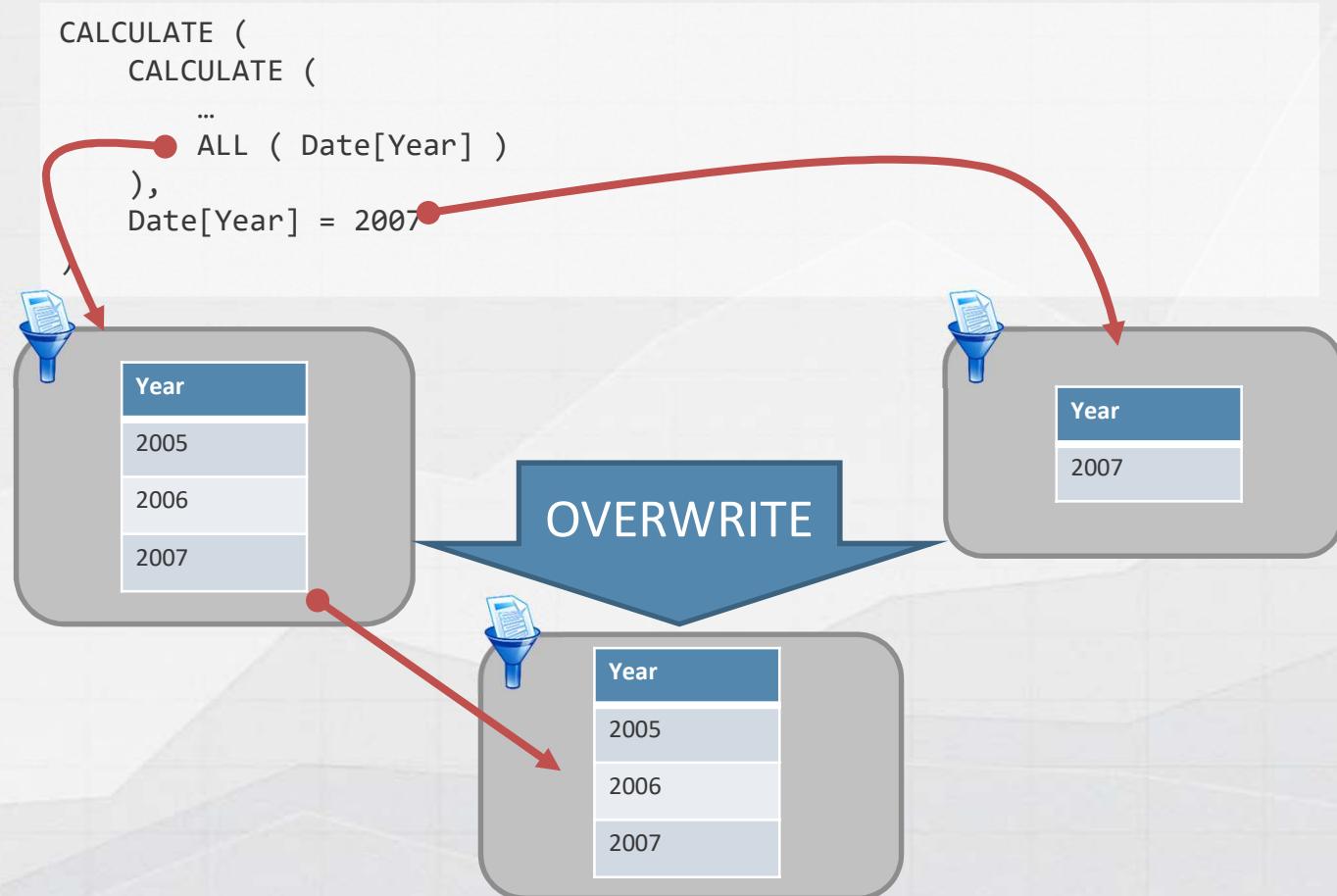
```
CALCULATE (
    DISTINCTCOUNT ( Customers[Customer Name] ),
    CROSSFILTER (
        Sales[CustomerKey],
        Customer[CustomerKey],
        BOTH
    )
)
```

Options:
NONE, ONEWAY, BOTH

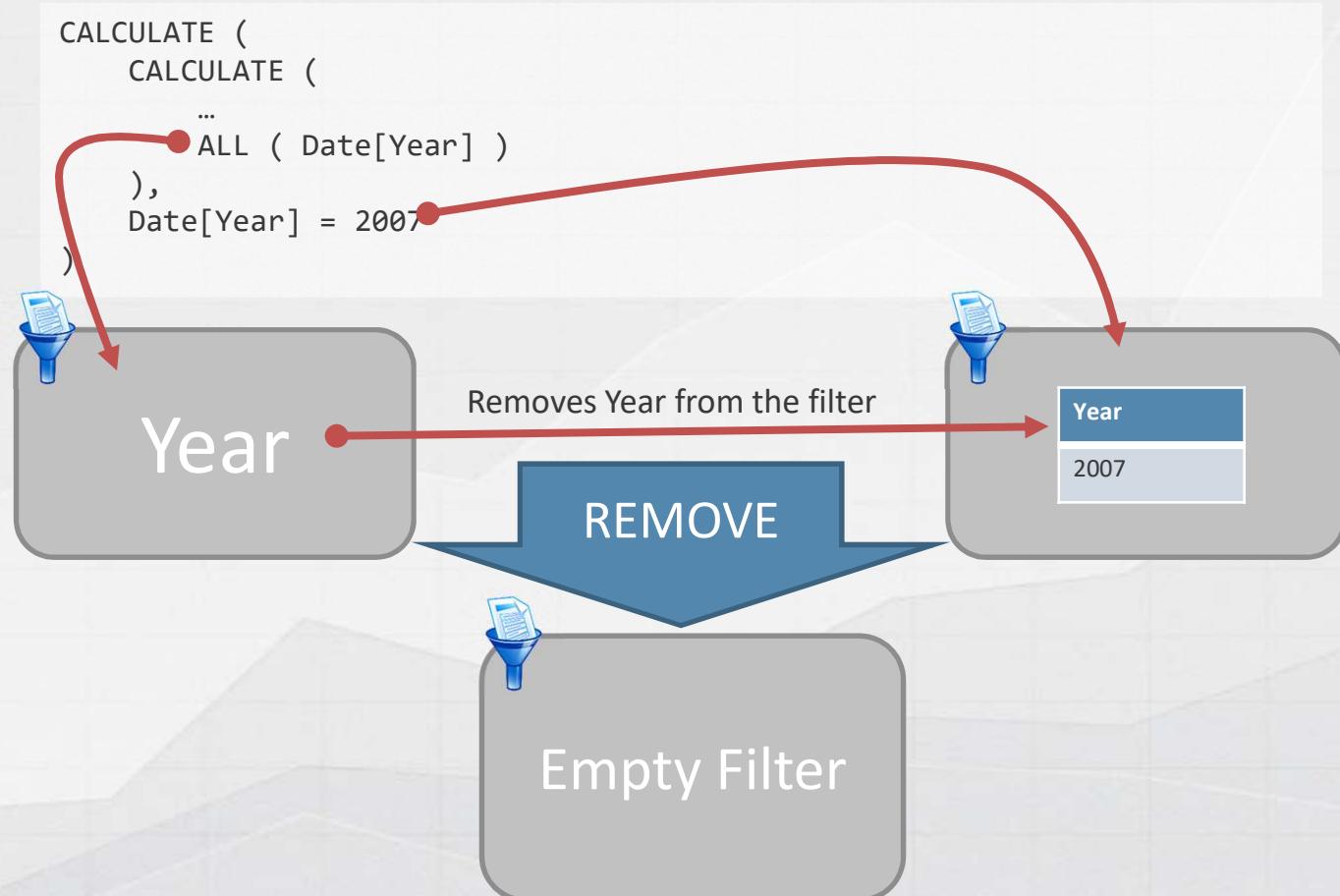
ALL performs removal of filters

- ALL is a table function
- Nevertheless, it is also a CALCULATE modifier
- Should be named REMOVEFILTER
 - Removes the filters from all the columns contained in the table it returns
- Only when used in CALCULATE it is a filter modifier – otherwise, it is a table function
- Same for all the functions in the ALL* family

What seems to be happening



What really happens



ALLSELECTED

ALLSELECTED is both a table function and a CALCULATE modifier.

Intuitively it restores the original filter context (slicer and filter, not rows and columns).
The internal behavior is much different, use it with care.

More information here: <https://www.sqlbi.com/articles/the-definitive-guide-to-allselected/>

ALLSELECTED (table[column])

ALLSELECTED (table)

ALLSELECTED ()

ALLSELECTED: visual totals

Occupation
Clerical
Management
Manual
Professional
Skilled Manual

Row Labels	Sum of SalesAmount		Column Labels	Grand Total
	F	M	Grand Total	
Clerical	£2,263,459.25	£2,421,327.40	£4,684,786.64	
Management	£2,674,334.07	£2,793,527.47	£5,467,861.54	
Manual	£1,394,911.13	£1,463,059.76	£2,857,970.89	
Professional	£5,134,484.19	£4,773,493.09	£9,907,977.28	
Grand Total	£11,467,188.64	£11,451,407.72	£22,918,596.36	

ALL would remove the filter from Occupation.

Sometimes, you need to retrieve the total of the selection (visual total), not the total removing the filter completely.

ALLSELECTED allows you to get the “visual total”.

ALLSELECTED (column)

Occupation
Clerical
Management
Manual
Professional
Skilled Manual

Row Labels	Sum of SalesAmount Column Labels		
	F	M	Grand Total
Clerical	£2,263,459.25	£2,421,327.40	£4,684,786.64
Management	£2,674,334.07	£2,793,527.47	£5,467,861.54
Manual	£1,394,911.13	£1,463,059.76	£2,857,970.89
Professional	£5,134,484.19	£4,773,493.09	£9,907,977.28
Grand Total	£11,467,188.64	£11,451,407.72	£22,918,596.36

```
CALCULATE (
    SUM ( [SalesAmount] ),
    ALLSELECTED ( DimCustomer[Occupation] )
)
= 11,467,188.64
```

ALLSELECTED ()

Occupation
Clerical
Management
Manual
Professional
Skilled Manual

Row Labels	Sum of SalesAmount Column Labels ▾		
	F	M	Grand Total
Clerical	£2,263,459.25	£2,421,327.40	£4,684,786.64
Management	£2,674,334.07	£2,793,527.47	£5,467,861.54
Manual	£1,394,911.13	£1,463,059.76	£2,857,970.89
Professional	£5,134,484.19	£4,773,493.09	£9,907,977.28
Grand Total	£11,467,188.64	£11,451,407.72	£22,918,596.36

```
CALCULATE (
    SUM ( [SalesAmount] ),
    ALLSELECTED ( )
)
= 22,918,596,36
```

KEEPFILTERS

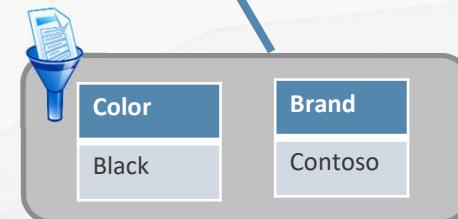
KEEPFILTERS is a filter modifier, not a CALCULATE modifier.

It changes the behavior of one filter only, it is applied when the filter is applied.

```
CALCULATE (
    CALCULATE (
        [Sales Amount],
        KEEPFILTERS <-- Product[Color] IN { "Yellow", "Black" } ),
        Product[Brand] = "Contoso"
    ),
    Product[Color] { "Black", "Blue" },
    Product[Brand] = "Northwind Traders"
)
```

Outer filter on
Brand is ignored

Outer filter on
Color is kept



Context Transition

- Calculate performs another task
- If executed inside a row context
 - It takes the row context
 - Transforms it into an equivalent filter context
 - Applies it to the data model
 - Before computing its expression
- Very important and useful feature
 - Better to learn it writing some code...

Context transition

Sales

Product	Quantity	Net Price
A	1	11.00
B	2	25.00
A	2	10.99

Row Context

```
Test :=  
SUMX (  
    Sales,  
    CALCULATE ( SUM ( Sales[Quantity] ) )  
)
```

Filter Context

SUMX Iteration

Row Iterated	Sales[Quantity] Value	Row Result
1	1	1
2	2	2
3	2	2

The result of
SUMX is 5



Product	Quantity	Net Price
A	1	11.00

Unexpected results if there are duplicated rows

Sales

Product	Quantity	Net Price
A	1	11.00
B	2	25.00
B	2	25.00

Row Context

```
Test :=  
SUMX (  
    Sales,  
CALCULATE ( SUM ( Sales[Quantity] ) )  
)
```

Filter Context

Product	Quantity	Net Price
B	2	25.00

SUMX Iteration

Row Iterated	Sales[Quantity] Value	Row Result
1	1	1
2	2	4
3	2	4

The result of
SUMX is 9

Some notes on context transition

- It is invoked by CALCULATE
- It is expensive: don't use it iterating large tables
- It does not filter one row, it filters all the identical rows
- It creates a filter context out of a row context
- It happens whenever there is a row context
- It transforms all the row contexts, not only the last one
- Row contexts are no longer available in CALCULATE

Automatic CALCULATE

Whenever a measure is invoked, an automatic CALCULATE is added around the measure.

This is the reason why using [Measure] and Table[Column] as a standard is a best practice.

```
SUMX (
    Orders,
    [Sales Amount]
)  

      ↓  

SUMX (
    Orders,
    CALCULATE ( [Sales Amount] )
)
```

Circular dependency

Circular dependency happens when two columns (or measures) depend one each other, in a way that DAX cannot perform the calculation.

```
Product[MarginPct] :=
```

```
    Product[Margin] / Product[Unit Cost]
```

```
Product[Margin] :=
```

```
    Product[MarginPct] * Product[Unit Cost]
```

Circular dependency

When using CALCULATE, context transition makes the expression depend on all the columns of the table, because the new filter context filters all the columns. Dependency is much more complex than expected.

```
Sales[Sum of Sales Amount] =  
CALCULATE (  
    SUM ( Sales[Sales Amount] )  
)
```

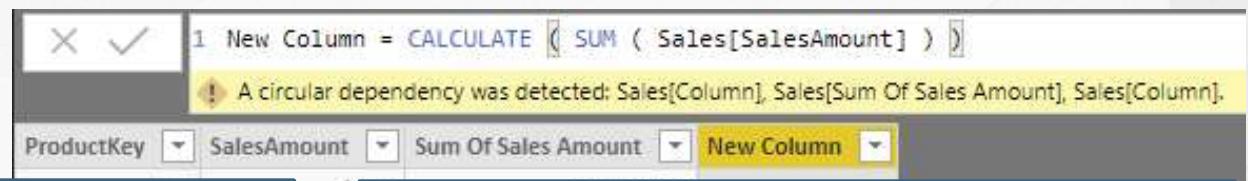
The column depends on

- Sales Amount
- Product Key

Circular dependency

When using CALCULATE, context transition makes the expression depend on all the columns of the table, because the new filter context filters all the columns. Dependency is much more complex than expected.

```
Sales[New Column] =  
  
CALCULATE (  
    SUM ( Sales[Sales Amount] )  
)
```



New column depends from

- Sales Amount
- Product Key
- Sum of Sales Amount

The old column depends from

- Sales Amount
- Product Key
- New Column

Circular dependency solution

- Either:
 - Mark a column as the row identifier
 - Create an incoming relationship
 - Use ALLEXCEPT to remove dependencies
- With a row identifier
 - CALCULATE columns depend on the row id
 - All of them, no circular dependency
- Row Identifiers are expensive for fact tables
 - Maximum number of distinct values
 - Use with care, avoid if possible

CALCULATE execution order

CALCULATE starts from the current context

1. It *evaluates* filter arguments, still in the original evaluation contexts (both row context and filter context)
2. It executes context transition
 - Adds all the column filter to the new context
3. It evaluates the modifiers
 - USERELATIONSHIP, CROSSFILTER, ALL
4. It applies the filter arguments evaluated in (1)

Modifiers executed before filters

CALCULATE modifiers are applied BEFORE the filter arguments.

Therefore, filter arguments use the model after the modifiers have been applied.

Delivered Amount in 2007 :=

```
CALCULATE (
    [Sales Amount],
    'Date'[Calendar Year] = "CY 2007",
    USERELATIONSHIP (
        Sales[DeliveryDateKey] ,
        'Date'[DateKey]
    )
)
```

First, the relationship is activated
Second, the year is filtered

Therefore, it computes orders
delivered in 2007

Modifiers executed before filters

ALL is a CALCULATE modifier. Therefore, it is executed before the CALCULATE filters.
In this case, ALL vanishes the effect of KEEPFILTERS.

```
Sales Red :=  
CALCULATE (  
    [Sales Amount],  
    'Product'[Color] = "Red"  
)
```

```
Sales Red :=  
CALCULATE (  
    [Sales Amount],  
    KEEPFILTERS ( 'Product'[Color] = "Red" ),  
    ALL ( 'Product'[Color] )  
)
```

The two measures are equivalent,
because ALL – as a modifier – is
executed before KEEPFILTERS

Advanced evaluation contexts



Time to write some DAX code by yourself.

Next exercise session is focuses on some DAX code that uses evaluation contexts.

Please refer to **lab number 4** on the hands-on manual

Time to start thinking in DAX

Working with iterators



Computing max daily sales

- What is the maximum amount of sales in one day?
- MAX is not enough
- Consolidate the daily amount and then find the maximum value

Year	Sales Amount	DailyMax
CY 2007	\$1,617,722.61	\$37,485.00
January 2007	\$117,679.75	\$34,487.70
1/2/2007	\$34,487.70	\$34,487.70
1/3/2007	\$12,760.35	\$12,760.35
1/4/2007	\$646.82	\$646.82
1/5/2007	\$5,248.59	\$5,248.59
1/7/2007	\$3,563.05	\$3,563.05
1/9/2007	\$1,935.00	\$1,935.00
1/10/2007	\$1,199.85	\$1,199.85
1/11/2007	\$3,454.32	\$3,454.32
1/12/2007	\$763.86	\$763.86
1/13/2007	\$606.13	\$606.13

Year	Sales Amount	DailyMax
CY 2007	\$1,617,722.61	\$37,485.00
January 2007	\$117,679.75	\$34,487.70
1/2/2007	\$34,487.70	\$34,487.70
1/3/2007	\$12,760.35	\$12,760.35
1/4/2007	\$646.82	\$646.82
1/5/2007	\$5,248.59	\$5,248.59
1/7/2007	\$3,563.05	\$3,563.05
1/9/2007	\$1,935.00	\$1,935.00
1/10/2007	\$1,199.85	\$1,199.85
1/11/2007	\$3,454.32	\$3,454.32
1/12/2007	\$763.86	\$763.86
1/13/2007	\$606.13	\$606.13

MIN-MAX sales per customer

Iterators can be used to compute values at a different granularity than the one set in the report.

```
MinSalesPerCustomer :=
```

```
    MINX ( Customer, [Sales Amount] )
```

```
MaxSalesPerCustomer :=
```

```
    MAXX ( Customer, [Sales Amount] )
```

Useful iterators

There are many useful iterators, they all behave the same way: iterate on a table, compute an expression and aggregate its value.

- MAXX
- MINX
- AVERAGEX
- SUMX
- PRODUCTX
- CONCATENATEX
- VARX.P | .S
- STDEVX.P | .S
- MEDIANX
- PERCENTILEX.EXC | .INC
- GEOMEANX

The RANKX function

Useful to compute ranking, not so easy to master.

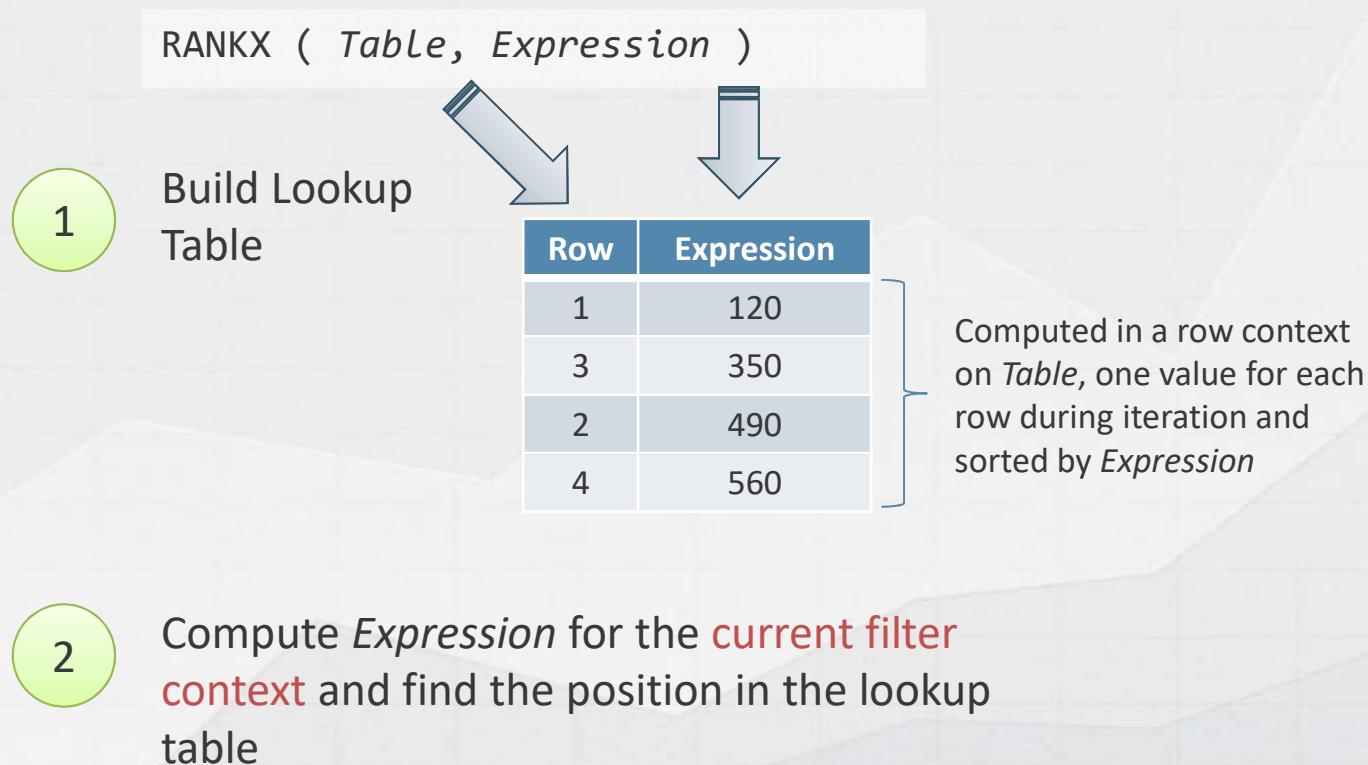
Good for making practice with row context and filter context.

```
--  
-- Syntax is easy...  
--
```

```
Ranking :=
```

```
RANKX (  
    Products,  
    Products[ListPrice]  
)
```

How RANKX works



RANKX (Table, Expression)

- Table
 - Evaluated in the external filter context
- Expression
 - Evaluated in the row context of Table during the iteration
 - Evaluated in the external filter context for the rank calculation
- These features make it a hard function, at first sight

RANKX most common pitfall

The Products table is evaluated in the filter context of the cell.

Thus, the lookup table contains a single row in each cell of the report other than Total.

```
RankOnSales :=
```

```
RANKX (
    Products,
    [SumOfSales]
)
```

Product	SumOfSales	RankOnSale
Bike	130.00	1
Helmet	70.00	1
Shirt	110.00	1
Shoes	240.00	1
Total	550.00	1

ALL is most likely the right table to iterate

Using ALL, the lookup table contains all of the products and RANKX works as expected.
ALLSELECTED is another good candidate (depending on requirements).

```
RankOnSales :=
```

```
RANKX (
    ALL ( Products ),
    [SumOfSales]
)
```

Product	SumOfSales	RankOnSale
Bike	130.00	2
Helmet	70.00	4
Shirt	110.00	3
Shoes	240.00	1
Total	550.00	1

RANKX and context transition

RANKX somewhat relies on context transition.

If the context transition is missing, then wrong values will be computed.

```
RankOnSales :=
```

```
RANKX (
    ALL ( Products ),
    SUM ( Sales[Sales] )
)
```

Product	SumOfSales	RankOnSale
Bike	130.00	1
Helmet	70.00	1
Shirt	110.00	1
Shoes	240.00	1
Total	550.00	1

CALCULATE to force context transition

You need to force context transition, when needed, using CALCULATE.
Using a measure reference does not have this requirement because it is implicit.

```
RankOnSales :=
```

```
RANKX (
    ALL ( Products ),
    CALCULATE ( SUM ( Sales[Sales] ) )
)
```

Product	SumOfSales	RankOnSale
Bike	130.00	2
Helmet	70.00	4
Shirt	110.00	3
Shoes	240.00	1
Total	550.00	1

ISINSCOPE to detect the hierarchy level

ISINSCOPE lets you detect when a column is currently being grouped by.

It is useful when you want to control the calculation in different levels of a hierarchy.

```
RankOnSale =
```

```
IF (
    ISINSCOPE ( Products[Product] ),
    RANKX (
        ALLSELECTED ( Products ),
        [SumOfSales]
    )
)
```

Category	Product	SumOfSales	RankOnSale
Sport	Bike	130.00	2
	Helmet	70.00	4
	Total	200.00	
Wearings	Shirt	110.00	3
	Shoes	240.00	1
	Total	350.00	
	Total	550.00	

ALLEXCEPT, ALLSELECTED, ISINSCOPE

Mixing different table functions and ISINSCOPE you can obtain a powerful report like the one shown here, with ranking of categories and of products within each category.

```
RankOnSale =  
  
IF (  
    ISINSCOPE ( Products[Product] ),  
    RANKX (  
        ALLEXCEPT (  
            Products,  
            Products[Category]  
        ),  
        [SumOfSales]  
    ),  
    IF (  
        ISINSCOPE ( Products[Category] ),  
        RANKX (  
            ALLSELECTED ( Products[Category] ),  
            [SumOfSales]  
        )  
    )  
)
```

Category	Product	SumOfSales	RankOnSale
Sport	Bike	130.00	1
	Helmet	70.00	2
	Total	200.00	2
Wearings	Shirt	110.00	2
	Shoes	240.00	1
	Total	350.00	1
Total		550.00	

RANKX 3° parameter

- Expression
 - Evaluated in the row context of <table>
- Value
 - Evaluated in the row context of the caller
 - Defaults to <Expression>
 - But can be a different one

```
RANKX (
    Table,
    Expression, → Row Context of table
    Value → Context of the caller
)
```

RANKX 3° parameter need

Sometimes the expression can be invalid in one of the contexts.

```
RankOnPrice:=  
IF (  
    HASONEVALUE ( Products[Product] ),  
    RANKX (  
        ALLSELECTED ( Products ),  
        Products[Price],  
        VALUES ( Products[Price] )  
    )  
)
```

Row Labels	Price	RankOnPrice
Bike	100	4
Helmet	200	3
Shirt	300	2
Shoe	400	1
Grand Total	1000	

Working with iterators



Time to write some DAX code by yourself.

Next exercise session is focuses on some simple and not-so-simple DAX code. Choose the exercise that best fit your skills.

Please refer to **lab number 5** on the hands-on manual.

Probably the most important table in your model

Building a date table



Date table

- Time intelligence needs a date table
 - Built in DAX, Power Query, SQL
 - DAX example: <https://www.sqlbi.com/tools/dax-date-template/>
- Date table properties
 - All dates should be present – no gaps
 - From 1° of January, to 31° of December
 - Or for the fiscal calendar including full months
 - Otherwise time intelligence DAX functions do not work

Auto date/time and column variations

In Power BI the auto date/time setting automatically creates one date table for each date column in the model.

It is not a best practice, unless you are using a very simple model.

Columns in these auto-created tables can be accessed through “column variations”.

```
Sales Amount YTD :=  
TOTALYTD (  
    'Sales'[Sales Amount],  
    'Sales'[Order Date].[Date]  
)  
Column variation
```

Year ▼	Quarter	Month	Day	Sales Amount	Sales Amount YTD
2009	Qtr 1	January	1	2,198.95	2,198.95
			2	1,325.89	3,524.84
			3	1,775.52	5,300.35
			4	2,167.90	7,468.25
			5	511.70	7,979.95
			6	907.89	8,887.84
			7	332.37	9,220.21
			8	4,605.52	13,825.73
			9	4,442.14	18,267.87
			10	82.70	18,350.58
			11	60.44	18,411.01
			12	1,771.18	20,182.19

CALENDARAUTO

Automatically creates a calendar table based on the database content.
Optionally you can specify the last month (for fiscal years).

```
--  
-- The parameter is the last month  
-- of the fiscal year  
--  
= CALENDARAUTO (  
    6  
)
```

Beware: CALENDARAUTO uses
all the dates in your model,
excluding only calculated
columns and tables

CALENDAR

Returns a table with a single column named “Date” containing a contiguous set of dates in the given range, inclusive.

```
CALENDAR (
    DATE ( 2005, 1, 1 ),
    DATE ( 2015, 12, 31 )
)
```

```
CALENDAR (
    MIN ( Sales[Order Date] ),
    MAX ( Sales[Order Date] )
)
```

CALENDAR

If you have multiple fact tables, you need to compute the correct values.

```
=CALENDAR (
    MIN (
        MIN ( Sales[Order Date] ),
        MIN ( Purchases[Purchase Date] )
    ),
    MAX (
        MAX ( Sales[Order Date] ),
        MAX ( Purchases[Purchase Date] )
    )
)
```

Mark as date table

- Need to mark the calendar as date table
- Set the column containing the date
- Needed to make time intelligence work if the relationship does not use a Date column
- Multiple tables can be marked as date table
- Used by client tools as metadata information

Set sorting options

- Month names do not sort alphabetically
 - April is not the first month of the year
- Use Sort By Column
- Set all sorting options in the proper way
- Beware of sorting granularity
 - 1:1 between names and sort keys

Using multiple dates

- Date is often a role dimension
 - Many roles for a date
 - Many date tables
- How many date tables?
 - Try to use only one table
 - Use many, only if needed by the model
 - Many date tables lead to confusion
 - And issues when slicing
- Use proper naming convention

Time intelligence functions

Time intelligence in DAX



What is time intelligence?

- Many different topics in one name
 - Year To Date
 - Quarter To Date
 - Running Total
 - Same period previous year
 - Working days computation
 - Fiscal Year

Aggregations over time

- Many useful aggregations
 - YTD: Year To Date
 - QTD: Quarter To Date
 - MTD: Month To Date
- They all need a date table
- And some understanding of CALCULATE

Sales 2015 up to 05-15 (v1)

Using CALCULATE you can filter the dates of the period to summarize.

```
SalesAmount20150515 :=  
  
CALCULATE (  
    SUM ( Sales[SalesAmount] ),  
    FILTER (  
        ALL ( 'Date'[Date] ),  
        AND (  
            'Date'[Date] >= DATE ( 2015, 1, 1 ),  
            'Date'[Date] <= DATE ( 2015, 5, 15 )  
        )  
    )  
)
```

Sales 2015 up to 05-15 (v2)

You can replace FILTER with DATESBETWEEN.

The result is always a table with a column.

```
SalesAmount20150515 :=
```

```
CALCULATE (
    SUM ( Sales[SalesAmount] ),
    DATESBETWEEN (
        'Date'[Date],
        DATE ( 2015, 1, 1 ),
        DATE ( 2015, 5, 15 )
    )
)
```

Sales Year-To-Date (v1)

Replace the static dates using DAX expressions that retrieve the last day in the current filter.

```
SalesAmountYTD :=  
  
CALCULATE (  
    SUM ( Sales[SalesAmount] ),  
    DATESBETWEEN (  
        'Date'[Date],  
        DATE ( YEAR ( MAX ( 'Date'[Date] ) ), 1, 1 ),  
        MAX ( 'Date'[Date] )  
    )  
)
```

Year to date (Time Intelligence)

DATESYTD makes filtering much easier.

```
SalesAmountYTD :=  
  
CALCULATE (  
    SUM ( Sales[SalesAmount] ),  
    DATESYTD ( 'Date'[Date] )  
)
```

Year to date: the easy way

TOTALYTD: the “DAX for dummies” version.

It hides the presence of CALCULATE, so we suggest not to use it.

```
SalesAmountYTD :=
```

```
TOTALYTD (
    SUM ( Sales[SalesAmount] ),
    'Date'[Date]
)
```

Handling fiscal year

The last, optional, parameter is the end of the fiscal year.

Default: 12-31 (or 31/12 – you can use any format regardless of locale settings).

```
SalesAmountYTD :=  
TOTALYTD (  
    SUM ( Sales[SalesAmount] ),  
    'Date'[Date],  
    "06-30"  
)  
  
SalesAmountYTD :=  
CALCULATE (  
    SUM ( Sales[SalesAmount] ),  
    DATESYTD ( 'Date'[Date], "06-30" )  
)
```

DATEADD

Shifts a table back and forth over time, using parameters to define the shift period.
The time period can be DAY, MONTH, QUARTER, YEAR.

```
Sales SPLY :=  
  
CALCULATE (  
    SUM( Sales[SalesAmount] ),  
    DATEADD ( 'Date'[Date] , -1, YEAR )  
)
```

Same period last year

Specialized version of DATEADD that always goes back one year.

```
Sales SPLY :=  
  
CALCULATE (  
    SUM ( Sales[SalesAmount] ),  
    SAMEPERIODLASTYEAR ( 'Date'[Date] )  
)
```

PARALLELPERIOD

Returns a set of dates (a table) shifted in time.

The entire period is returned, regardless of dates not selected in the initial filter context.

```
Sales PPLY :=  
CALCULATE (  
    SUM ( Sales[SalesAmount] ),  
    PARALLELPERIOD ( 'Date'[Date] , -1, YEAR )  
)  
  
-- very similar to
```

```
Sales PPLY :=  
CALCULATE (  
    SUM ( Sales[SalesAmount] ),  
    PREVIOUSYEAR ( 'Date'[Date] )  
)
```

Moving annual total (v1)

DATESINPERIOD returns all the dates in a given number of periods, starting from a reference date. Negative offsets go back in time.

```
CALCULATE (
    SUM ( Sales[SalesAmount] ),
    DATESINPERIOD (
        'Date'[Date],
        MAX ( 'Date'[Date] ),
        -1,
        YEAR
    )
)
```

Running total

Running total requires an explicit filter.

We use a variable to store the last visible date in the current filter context.

```
SalesAmountRT :=  
  
VAR LastVisibleDate = MAX ( 'Date'[Date] )  
  
RETURN  
  
CALCULATE (  
    SUM ( Sales[SalesAmount] ),  
    FILTER (  
        ALL ( 'Date' ),  
        'Date'[Date] <= LastVisibleDate  
    )  
)
```

Mixing time intelligence functions

Parameters of time intelligence functions are tables.
Using a column – as we did so far – is only syntax sugar.

```
Sales YTDLY :=
```

```
    CALCULATE (
        SUM ( Sales[SalesAmount] ),
        DATESYTD (
            SAMEPERIODLASTYEAR ( 'Date'[Date] )
        )
    )
```

DATESYTD ('Date'[Date])

is similar to

DATESYTD (VALUES ('Date'[Date]))

Moving annual total (v2)

Moving window from the current date back one year, one can build it using basic functions. It could be slower, but it provides more flexibility over DATESINPERIOD if needed.

```
CALCULATE (
    SUM ( Sales[SalesAmount] ),
    DATESBETWEEN (
        'Date'[Date],
        NEXTDAY (
            SAMEPERIODLASTYEAR (
                LASTDATE ( 'Date'[Date] )
            )
        ),
        LASTDATE ( 'Date'[Date] )
    )
)
```

Beware of function order!

Time intelligence functions return sets of dates, and the set of dates need to exist. A non-existing date results in BLANK, generating wrong or unexpected results.

```
CALCULATE (
    SUM ( Sales[SalesAmount] ),
    DATESBETWEEN (
        'Date'[Date],
        SAMEPERIODLASTYEAR (
            NEXTDAY (
                LASTDATE ( 'Date'[Date] )
            )
        ),
        LASTDATE ( 'Date'[Date] )
    )
)
```

Semi-additive measures



Semi-additive measures

- Additive Measure
 - SUM over all dimensions
- Semi-additive Measure
 - SUM over some dimensions
 - Different function over other dimensions
 - Time is the standard exception for aggregations
 - Examples
 - Warehouse stocking
 - Current account balance

Current account balance

Name	Country	Occupation	Date	Balance
Katie Jordan	USA	Farmer	1/31/2010	1,687.00
Luis Bonifaz	Argentina	IT Consultant	1/31/2010	1,470.00
Maurizio Macagno	Italy	IT Consultant	1/31/2010	1,500.00
Katie Jordan	USA	Farmer	2/28/2010	2,812.00
Luis Bonifaz	Argentina	IT Consultant	2/28/2010	2,450.00
Maurizio Macagno	Italy	IT Consultant	2/28/2010	
Katie Jordan	USA	Farmer	3/31/2010	
Luis Bonifaz	Argentina	IT Consultant	3/31/2010	
Maurizio Macagno	Italy	IT Consultant	3/31/2010	

Year	Quarter	Month	Katie Jordan	Luis Bonifaz	Maurizio Macagno	Total
CY 2010	Q1	January	1,687.00	1,470.00	1,500.00	4,657.00
		February	2,812.00	2,450.00	2,500.00	7,762.00
		March	3,737.00	3,430.00	3,500.00	10,667.00
		Total	8,236.00	7,350.00	7,500.00	23,086.00
	Q2	April	2,250.00	1,960.00	2,000.00	6,210.00
		May	2,025.00	1,764.00	1,800.00	5,589.00
		June	2,700.00	2,352.00	2,400.00	7,452.00
		Total	6,975.00	6,076.00	6,200.00	19,251.00
	Q3	July	3,600.00	3,136.00	3,200.00	9,936.00
		August	5,062.00	4,410.00	4,500.00	13,972.00
		September	2,812.00	2,450.00	2,500.00	7,762.00
		Total	11,474.00	9,996.00	10,200.00	31,670.00
Year	Quarter	Month	Katie Jordan	Luis Bonifaz	Maurizio Macagno	Total
CY 2010	Q1	January	1,687.00	1,470.00	1,500.00	4,657.00
		February	2,812.00	2,450.00	2,500.00	7,762.00
		March	3,737.00	3,430.00	3,500.00	10,667.00
		Total	8,236.00	7,350.00	7,500.00	23,086.00
	Q2	April	2,250.00	1,960.00	2,000.00	6,210.00
		May	2,025.00	1,764.00	1,800.00	5,589.00
		June	2,700.00	2,352.00	2,400.00	7,452.00
		Total	6,975.00	6,076.00	6,200.00	19,251.00
	Q3	July	3,600.00	3,136.00	3,200.00	9,936.00
		August	5,062.00	4,410.00	4,500.00	13,972.00
		September	2,812.00	2,450.00	2,500.00	7,762.00
		Total	11,474.00	9,996.00	10,200.00	31,670.00

- Month level **correct**
- Quarter level **wrong**
- Year level **wrong**

Semi-additive measures

- Aggregation depends on the filter
 - Last date, over time
 - SUM for the other dimensions

Year	Quarter	Month	Katie	Jordan	Luis Bonifaz	Maurizio Macagno	Total
CY 2010	Q1	+ January	1,687.00	1,470.00	1,500.00	4,657.00	
		+ February	2,812.00	2,450.00	2,500.00	7,762.00	
		+ March	3,737.00	3,430.00	3,500.00	10,667.00	
	Total		3,737.00	3,430.00	3,500.00	10,667.00	
	Q2	+ April	2,250.00	1,960.00	2,000.00	6,210.00	
		+ May	2,025.00	1,764.00	1,800.00	5,589.00	
		+ June	2,700.00	2,352.00	2,400.00	7,452.00	
Total			2,700.00	2,352.00	2,400.00	7,452.00	

Semi-additive measures

LASTDATE searches for the last visible date in the current filter context; you can obtain the same result by using a variable to store the last visible day in the filter context.

```
LastBalance :=  
CALCULATE (  
    SUM ( Balances[Balance] ),  
    LASTDATE ( 'Date'[Date] )  
)  
  
LastBalance :=  
VAR LastDayVisible = MAX ( Date[Date] )  
RETURN  
    CALCULATE (  
        SUM ( Balances[Balance] ),  
        'Date'[Date] = LastDayVisible  
)
```

LASTNONBLANK

LASTNONBLANK iterates Date searching the last value for which its second parameter is not a BLANK. Thus, it searches for the last date with any row in the fact table.

```
LastBalanceNonBlank :=  
  
CALCULATE (  
    SUM ( Balances[Balance] ),  
    LASTNONBLANK (  
        'Date'[Date],  
        CALCULATE ( COUNTROWS ( Balances ) )  
    )  
)
```

```
CALCULATE (  
    COUNTROWS ( Balances ),  
    ALL ( Balances[Name] )  
)
```

Semi-additive measures

Most of the times, it is better to search for the last date with transactions, avoiding blank results when there is no data at the end of the period.

```
LastBalance :=  
  
VAR LastDayWithTransactions =  
    CALCULATE (  
        MAX ( Balances[Date] ),  
        ALL ( Balances[Name] )  
    )  
  
RETURN  
    CALCULATE (  
        SUM ( Balances[Balance] ),  
        'Date'[Date] = LastDayWithTransactions  
    )
```

LASTNONBLANK by customer (1)

Iterating over the customers, it is possible to compute the LASTNONBLANK for each customer, summing at the end the partial results

```
LastBalanceNonBlank :=  
  
SUMX (   
    VALUES ( Balances[Name] ),  
    CALCULATE (   
        SUM ( Balances[Balance] ),  
        LASTNONBLANK (   
            'Date'[Date],  
            CALCULATE ( COUNTROWS ( Balances ) )  
        )  
    )  
)
```

LASTNONBLANK by customer (2)

Using a filter context with multiple columns, you can compute an equivalent LASTNONBLANK filter on a customer-by-customer basis.

```
LastBalanceNonBlankPerCustomer :=  
  
CALCULATE (  
    SUM ( Balances[Balance] ),  
    TREATAS (  
        ADDCOLUMNS (  
            VALUES ( Balances[Name] ),  
            "LastAvailableDate", CALCULATE ( MAX ( Balances[Date] ) )  
        ),  
        Balances[Name],  
        'Date'[Date]  
    )  
)
```

This version uses TREATAS
and data lineage, we will
discuss them later

There are many week scenarios, depending on what you mean by «week»...

Calculations over weeks



Weeks do not aggregate to months and years

Date	DateKey	Calendar Year	Month Number	Month	Day of Week Number	Day of Week
1/1/2005	20050101	CY 2005		1 January		7 Saturday
1/2/2005	20050102	CY 2005		1 January		1 Sunday
1/3/2005	20050103	CY 2005		1 January		2 Monday
1/4/2005	20050104	CY 2005		1 January		3 Tuesday
1/5/2005	20050105	CY 2005		1 January		4 Wednesday
1/6/2005	20050106	CY 2005		1 January		5 Thursday
1/7/2005	20050107	CY 2005		1 January		6 Friday
1/8/2005	20050108	CY 2005		1 January		7 Saturday
1/9/2005	20050109	CY 2005		1 January		1 Sunday
1/10/2005	20050110	CY 2005		1 January		2 Monday
1/11/2005	20050111	CY 2005		1 January		3 Tuesday

Custom calendars

- Time Intelligence functions
 - One day belong to the same quarter every day
 - Not true if you use week calculations
- 4-4-5, 4-5-4, 5-4-4
 - One quarter is made of three months
 - Two of 4 weeks
 - One of 5 weeks
 - Difference only in position of the 5 weeks month
- No support in DAX for these calendars

Create week numbers

- Usually stored in the database
- If not, use DAX or Power Query
 - DAX example: <https://www.sqlbi.com/tools/dax-date-template/>

Calculate ISO year-to-date

You cannot leverage standard time intelligence calculations, you need to write the code by yourself. Nevertheless, this turns out to be much easier than expected.

Sales ISO YTD :=

```
VAR LastDayVisible =
    MAX ( 'Date'[FW DayOfYearNumber] )
RETURN CALCULATE (
    [Sales],
    ALL ( 'Date' ),
    'Date'[FW DayOfYearNumber] <= LastDayVisible,
    VALUES ( 'Date'[FW YearNumber] )
)
```

Year	Sales	YTD Sales
FW 2007	11,243,758.33	11,243,758.33
FW P01	715,084.77	715,084.77
FW P02	862,603.82	1,577,688.60
FW P03	1,068,984.79	2,646,673.39
FW P04	1,115,232.77	3,761,906.16
FW P05	802,523.65	4,564,429.81
FW P06	1,128,845.60	5,693,275.41
FW P07	809,198.05	6,502,473.46
FW P08	864,392.71	7,366,866.17
FW P09	1,159,085.17	8,525,951.33
FW P10	898,015.56	9,423,966.89
FW P11	773,291.37	10,197,258.26
FW P12	1,046,500.07	11,243,758.33

For more calculations see www.daxpatterns.com and
www.sqlbi.com/articles/week-based-time-intelligence-in-dax/



Time intelligence



Next exercise session is focused on some common scenarios where time intelligence is required, along with some evaluation context skills.

Please refer to **lab number 6** on the exercise book.

Let us discover how DAX handles hierarchies

Hierarchies in DAX



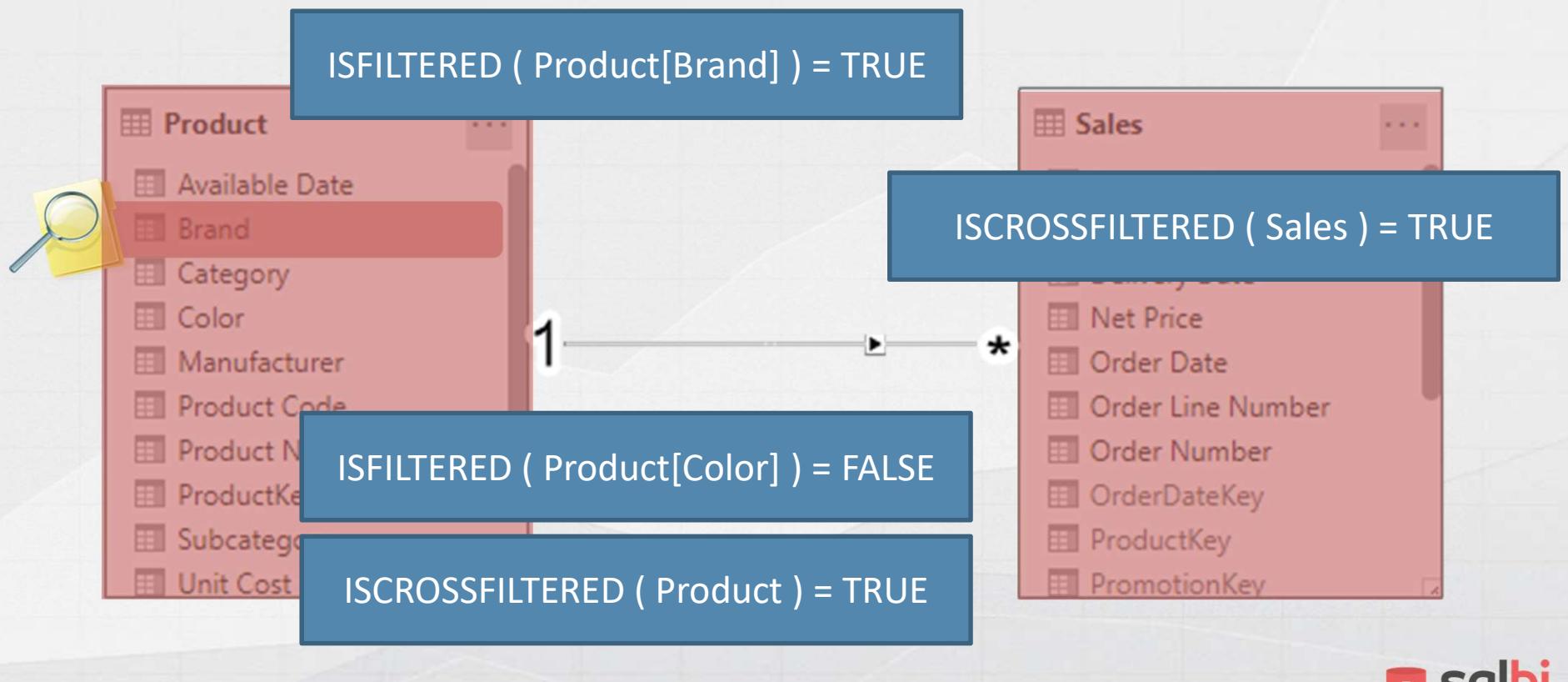
What are hierarchies?

- Predefined exploration paths
 - Year – Month – Day
 - Category – Subcategory – Product
- Make browsing a model easier
- Natural: there exists a 1:M relationship between parent and children
- Unnatural: data is shuffled
- With MDX, it works better with natural hierarchies
 - Detection happens during processing

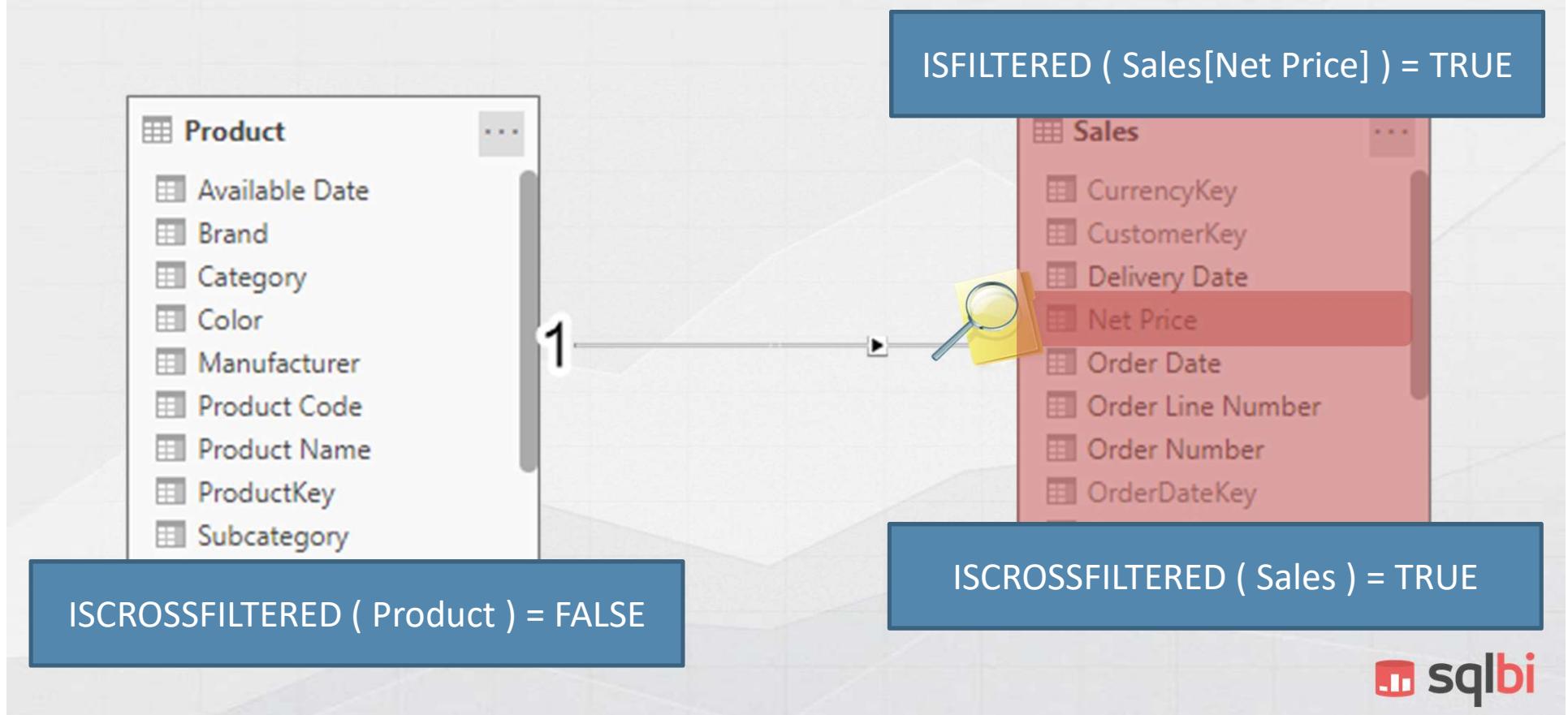
Hierarchies across tables

- All columns in a hierarchy need to belong to the same table
- Use RELATED to move columns in the right place
- Optionally hide original tables

FILTER and CROSSFILTER, from the 1-side



FILTER and CROSSFILTER, from the many-side



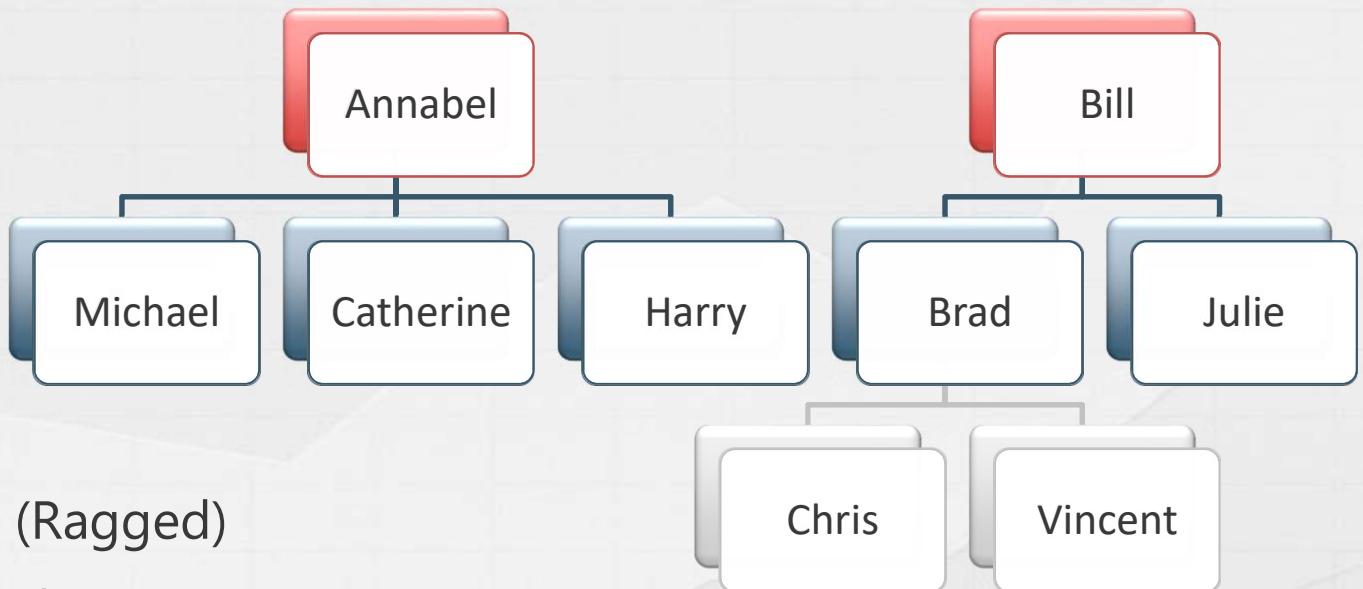
ISINSCOPE

- ISINSCOPE is similar to ISFILTERED
- It also checks that the column is currently used as a group-by column
- Useful to avoid detecting the slicers as filters

Percentages over hierarchies

- For a given hierarchy
 - Percentage over grand total
 - Percentage over selection
 - Percentage over Parent
- Plus: show everything in a single measure

Parent / child hierarchies



- Unbalanced (Ragged)
- Variable Depth
- Data associated to both leaves and nodes
- Handled in AS, not handled in Power BI and Power Pivot

The final result

Level1	PC Amount
Annabel	3,200
Catherine	1,200
Harry	800
Michael	600
Bill	1,600
Brad	1,300
Chris	400
Vincent	500
Julie	300
Total	4,800

Harry has no children

Brad has children

- Leaf nodes should not be expanded
- Each node contains
 - The sum of all of its children
 - Plus its personal data

The PATH function

Path performs P/C navigation recursively.

```
HierarchyPath =  
  
PATH (  
    Hierarchy[NodeId],  
    Hierarchy[ParentNodeID]  
)
```

Level columns

Create one calculated column for each level in the hierarchy.

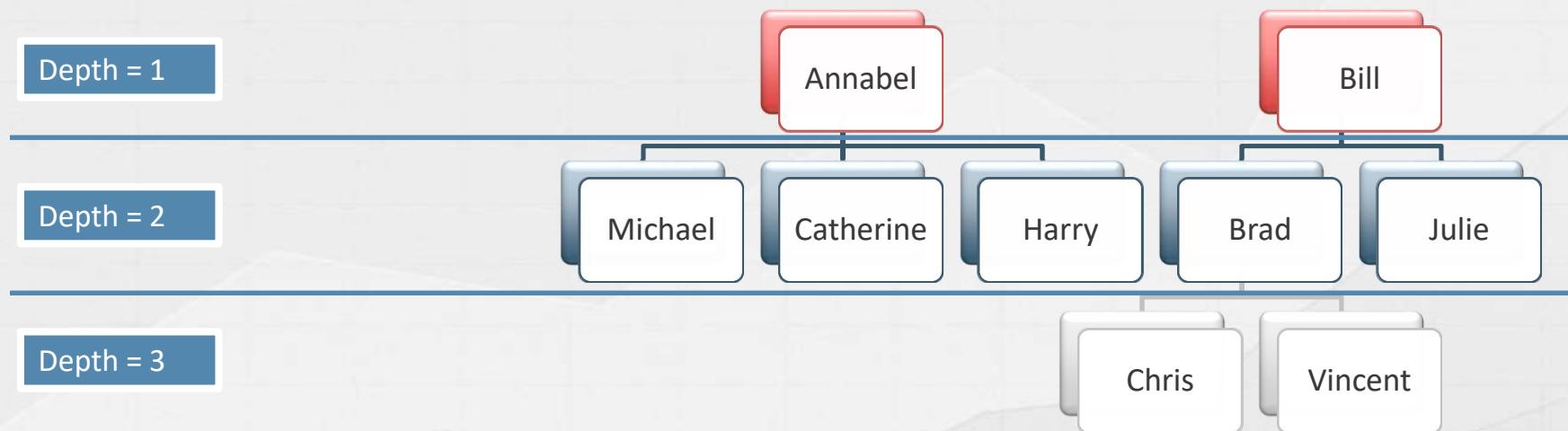
```
Level1 =  
  
LOOKUPVALUE (  
    Hierarchy[Node],  
    Hierarchy[NodeId],  
    PATHITEM (  
        Hierarchy[HierarchyPath],  
        1  
    )  
)
```

The «HideMemberIf» issue

- P/C are often ragged, i.e. unbalanced
- AS handles ragged hierarchies, Power Pivot and Power BI do not
- Some DAX acrobatics needed to obtain the desired result

NodeDepth column

Add a calculated column to the hierarchy which computes the node depth



NodeDepth = PATHLENGTH ([HierarchyPath])

Create a BrowseDepth measure

BrowseDepth computes the maximum filtered level in the hierarchy, i.e. the depth reached by the user browsing data.

```
[BrowseDepth] :=  
  
ISINSCOPE ( 'Hierarchy'[Level1] ) +  
ISINSCOPE ( 'Hierarchy'[Level2] ) +  
ISINSCOPE ( 'Hierarchy'[Level3] ) +  
...  
ISINSCOPE ( 'Hierarchy'[LevelN] ) +
```

Hide unwanted rows

When the MIN of the NodeDepth is less than the BrowseDepth, it means that we are below the current node NodeDepth and the measure should be BLANKed.

```
SumOfAmount :=  
  
IF (  
    [BrowseDepth] <= MIN ( 'Hierarchy'[NodeDepth] ),  
    SUM ( Invoices[Amount] ),  
    BLANK ()  
)
```

Parent/Child: Conclusions

- Parent/Child are challenging
 - Yet they can be solved
 - Complexity is in the formulas
- No DAX function to handle hierarchies
 - They are only UI items, no DAX available
 - Very different from MDX

How to create queries and manipulate tables with DAX

Working with tables and queries



Tools to query tabular

- SQL Server Management Studio
 - Offers a basic DAX query editing experience
 - IntelliSense not optimal
- Excel, Power View, Power BI Desktop
 - Remember: Excel uses MDX

DAX studio

- <http://daxstudio.org>
 - Free add-in for Excel
 - Standalone executable for AS and Power BI
- It can query
 - Tabular
 - Power Pivot data models
 - Power BI Desktop
 - Visual Studio integrated workspace
- Runs and measures both DAX and MDX queries



EVALUATE syntax

EVALUATE is the statement to use to write DAX queries.

DEFINE

 MEASURE <table>[<col>] = <expression>

EVALUATE <Table Expression>

 ORDER BY <expression> [ASC | DESC], ...

 START AT <value>, ...

EVALUATE example

The easiest EVALUATE: query a full table.

```
EVALUATE  
    Store  
  
ORDER BY  
    Store[Continent],  
    Store[Country],  
    Store[City]  
  
START AT "North America"
```

CALCULATETABLE

CALCULATETABLE should be used to apply filters on **existing columns** to a table expression.

```
EVALUATE  
CALCULATETABLE (  
    Product,                      -- This could be any table expression  
    Product[Color] = "Red"        -- This is like a slicer  
)  
  
EVALUATE  
CALCULATETABLE (  
    Product,  
    Product[Color] IN { "Red", "Blue", "White" } -- Like a slicer  
                                -- with multiple selections  
)
```

FILTER

FILTER should be used to take advantage of the row context.

For example, when a context transition is used by a measure or by CALCULATE.

```
EVALUATE  
FILTER (  
    Product,  
    [Sales Amount] > 1000000  
)
```

```
EVALUATE  
FILTER (  
    Product,  
    CALCULATE ( SUM ( Sales[Quantity] ) ) > 1000  
)
```

Evaluation order of CALCULATE

CALCULATE and CALCULATETABLE evaluate the first parameter only after the latter ones have been computed.

EVALUATE

```
CALCULATETABLE (
    CALCULATETABLE (
        Product,
        ALL ( Product[Color] ) ) ,
    Product[Color] = "Red"
)
```

The result is Product rows of any color

SELECTCOLUMNS

Projection in DAX, simple and useful.
Each expression is executed in a row context.

EVALUATE

```
SELECTCOLUMNS (
    'Product',
    "Category", RELATED ( 'Product Category'[Category] ),
    "Color", 'Product'[Color],
    "Name", 'Product'[Product Name],
    "Sales Amount", [Sales Amount]
)
```

ADDCOLUMNS

Adds one or more columns to a table expression, keeping all existing columns.
Each column is computed in a row context.

EVALUATE

```
ADDCOLUMNS (
    'Product Category',
    "Sales", [Sales Amount]
)
```

Remember the automatic
CALCULATE which surrounds
any measure calculation

SUMMARIZE

Performs GROUP BY in DAX and optionally computes subtotals.

WARNING: function almost deprecated for aggregating data (more on this later).

EVALUATE

SUMMARIZE (

Sales,
'Product Category' [Category],
"Sales", ~~[Sales Amount]~~

)

Source Table

GROUP BY Column(s)

Result Expression(s)
added to the source table

Beware of SUMMARIZE

Never use SUMMARIZE to compute calculated columns, always use a mix of SUMMARIZE and ADDCOLUMNS. SUMMARIZE semantics is very complex, leading to unexpected results.

EVALUATE

```
ADDCOLUMNS (
    SUMMARIZE (
        Sales,
        'Product Category'[Category],
        'Product Subcategory'[Subcategory]
    ),
    "Sales", [Sales Amount]
)
```

SUMMARIZECOLUMNNS

New version of SUMMARIZE, with a different semantic.

```
EVALUATE  
  
SUMMARIZECOLUMNNS (  
    'Date'[Calendar Year],  
    Product[Color],  
    "Sales", [Sales Amount]  
)
```

```
EVALUATE  
FILTER (  
    SUMMARIZE (  
        CROSSJOIN (  
            VALUES ( 'Date'[Calendar Year] ),  
            VALUES ( Product[Color] )  
        ),  
        'Date'[Calendar Year],  
        Product[Color],  
        "Sales", [Sales Amount]  
    ),  
    NOT ( ISBLANK ( [Sales] ) )  
)
```

SUMMARIZECOLUMNS

- Simpler syntax
- Empty rows are automatically removed
- Multiple fact tables scanned at once
- Some limits when used in a measure:
 - <http://www.sqlbi.com/articles/introducing-summarizecolumns/>
- Few additional features
 - IGNORE
 - ROLLUPGROUP
 - ROLLUPADDISSUBTOTAL
 - VISUAL

SUMMARIZECOLUMNS

Most of the queries can be expressed with a single function call.

```
SUMMARIZECOLUMNS (
    Customer[Company Name],
    ROLLUPADDISSUBTOTAL (
        ROLLUPGROUP ( Customer[City] ),
        "CityTotal"
    ),
    FILTER (
        ALL ( Customer[Country] ),
        Customer[Country] = "France"
    ),
    "Sales", [Sales Amount]
)
```

Using CROSSJOIN

CROSSJOIN does what its name suggests: performs a cartesian product between two tables.

EVALUATE

```
ADDCOLUMNS (
    CROSSJOIN (
        DISTINCT ( 'Product'[Color] ),
        DISTINCT ( 'Product'[Size] )
    ),
    "Sales", [Sales Amount]
)
```

Using TOPN

Returns the TOP N products sorting the table by Sales Quantity.

```
EVALUATE
```

```
CALCULATETABLE (
    TOPN (
        10,
        ADDCOLUMNS (
            VALUES ( Product[Product Name] ),
            "Sales", CALCULATE ( SUM ( Sales[Quantity] ) )
        ),
        [Sales]
    ),
    Product[Color] = "Red"
)
```

Using GENERATE

GENERATE is the equivalent of APPLY in SQL.

EVALUATE

```
GENERATE (
    VALUES ( 'Product Category'[Category] ),
    SELECTCOLUMNS (
        RELATEDTABLE ( 'Product Subcategory' ),
        "Subcategory", 'Product Subcategory'[Subcategory]
    )
)
ORDER BY [Category], [Subcategory]
```

TOPN and GENERATE

Returns the TOP 3 products for each category sorting by Sales Quantity.

```
EVALUATE
```

```
GENERATE (
    VALUES ( 'Product Category'[Category] ),
    CALCULATETABLE (
        TOPN (
            3,
            ADDCOLUMNS (
                VALUES ( Product[Product Name] ),
                "Sales", CALCULATE ( SUM ( Sales[Quantity] ) )
            ),
            [Sales]
        )
    )
)
```

Using ROW

Creates a one-row table, often used to get results, more rarely used inside calculations and complex queries.

EVALUATE

```
ROW (
    "Jan Sales", CALCULATE (
        SUM ( Sales[Quantity] ),
        'Date'[Month] = "January"
    ),
    "Feb Sales", CALCULATE (
        SUM ( Sales[Quantity] ),
        'Date'[Month] = "February"
    )
)
```

Using table constructor

A table constructor is more common than the ROW function, even though it does not control column names (Value1, Value2, ...).

```
EVALUATE
{ (
    CALCULATE (
        SUM ( Sales[Quantity] ),
        'Date'[Month] = "January"
    ),
    CALCULATE (
        SUM ( Sales[Quantity] ),
        'Date'[Month] = "February"
    )
)}
```

Using table constructor

A table constructor is the easiest way to display in DAX Studio the result of a measure or any other scalar expression.

```
EVALUATE
```

```
{ [Sales Amount] }
```

Using DATABASE

Creates a full table with a single function call, useful to build temporary tables.
Dynamic expressions not allowed in the list of values (only constants are allowed).

```
EVALUATE  
DATABASE (  
    "Price Range", STRING,  
    "Min Price", CURRENCY,  
    "Max Price", CURRENCY,  
    {  
        { "Low", 0, 10 },  
        { "Medium", 10, 100 },  
        { "High", 100, 999999 }  
    }  
)
```

Tables and relationships

Tables resulting from table functions inherits relationships from the columns they use.

```
EVALUATE  
ADDCOLUMNS (  
    CROSSJOIN (  
        VALUES ('Date'[Calendar Year]),  
        VALUES ( Product[Color] )  
    ),  
    "Sales Amount",  
    CALCULATE (  
        SUM ( Sales[Quantity] )  
    )  
)
```

Which sales?

Sales of the given year and color

Columns and expressions

Expressions are not columns.

Column references keep the same data lineage, expressions have a different one.

```
EVALUATE  
CALCULATETABLE (  
    ADDCOLUMNS (  
        SELECTCOLUMNS (  
            'Product',  
            "Product Name", Product[Product Name] & ""  
        ),  
        "Sales Amount", [Sales Amount]  
    ),  
    'Product'[Color] = "Red"  
)
```

Becoming an expression,
Product Name changes
the data lineage

UNION

Performs the UNION ALL of the inputs.

EVALUATE

```
VAR DatesWithSales = SUMMARIZE ( Sales, 'Date'[Date] )
```

```
VAR MondayDates =
```

```
    CALCULATETABLE (
```

```
        VALUES ( 'Date'[Date] ),
```

```
        'Date'[Day of Week] = "Monday"
```

```
    )
```

```
VAR Result =
```

```
    UNION ( DatesWithSales, MondayDates )
```

```
RETURN
```

```
    Result
```

Columns in UNION

Columns can originate from different sources. The first table defines the column names. The datatype is chosen to accommodate all the different datatypes of the column.

EVALUATE

```
UNION (
    ROW ( "DAX", 1 ),
    ROW ( "is a", 1 ),
    ROW ( "Language", 2 )
)
```

INTERSECT

Performs the intersection of the inputs.

```
EVALUATE  
CALCULATETABLE (  
    VAR FirstDay = MIN ( 'Date'[Date] )  
    VAR CurrentCustomers = VALUES ( Sales[CustomerKey] )  
    VAR PreviousCustomers =  
        CALCULATETABLE (  
            VALUES(Sales[CustomerKey]),  
            'Date'[Date] < FirstDay  
        )  
    VAR ReturningCustomers =  
        INTERSECT ( CurrentCustomers, PreviousCustomers )  
RETURN  
    ReturningCustomers,  
    'Date'[Calendar Year Number] = 2007,  
    'Date'[Month Number] = 2  
)
```

INTERSECT / UNION data lineage

INTERSECT gets column names and data lineage from the first table, whereas UNION only keep the data lineage if all the tables have the same one.

```
EVALUATE  
CALCULATETABLE (  
    VAR FirstDay =  
        MIN ( 'Date'[Date] )  
    VAR CurrentCustomers =  
        VALUES ( Sales[CustomerKey] )  
    VAR PreviousCustomers =  
        CALCULATETABLE ( VALUES(Sales[CustomerKey]), 'Date'[Date] < FirstDay )  
    VAR ReturningCustomers =  
        INTERSECT ( CurrentCustomers, PreviousCustomers )  
    RETURN  
        {([Sales Amount], CALCULATE ( [Sales Amount], ReturningCustomers ) )},  
    'Date'[Calendar Year Number] = 2007,  
    'Date'[Month Number] = 2  
)
```

EXCEPT

Subtract one set from another, returning items in Set1 not present in Set2.

```
EVALUATE  
CALCULATETABLE (  
    VAR FirstDay = MIN ( 'Date'[Date] )  
    VAR CurrentCustomers = VALUES ( Sales[CustomerKey] )  
    VAR PreviousCustomers =  
        CALCULATETABLE (  
            VALUES (Sales[CustomerKey] ),  
            'Date'[Date] < FirstDay  
        )  
    VAR NewCustomers =  
        EXCEPT ( CurrentCustomers, PreviousCustomers )  
    RETURN  
        { ( [Sales Amount], CALCULATE ( [Sales Amount], NewCustomers ) ) },  
        'Date'[Calendar Year Number] = 2007,  
        'Date'[Month Number] = 2  
)
```

GROUPBY

Similar to SUMMARIZE, but it has the additional feature of CURRENTGROUP to iterate over the subset of rows and can group the result of dynamic expressions.

```
EVALUATE  
VAR SalesPriceLevels =  
    ADDCOLUMNS (  
        ALL ( Sales[Unit Price] ),  
        "Sales", [Sales Amount],  
        "Price Level", SWITCH ( TRUE,  
            Sales[Unit Price] < 100, "LOW",  
            Sales[Unit Price] < 1000, "MEDIUM",  
            "HIGH"  
        )  
    )  
RETURN  
    GROUPBY (  
        SalesPriceLevels,  
        [Price Level],  
        "Sales", SUMX ( CURRENTGROUP (), [Sales] )  
    )
```

Query measures

Measures defined locally in the query make the authoring of complex queries much easier.

```
DEFINE
    MEASURE Sales[NumOfSubcategories] =
        COUNTROWS ( RELATEDTABLE ( 'Product Subcategory' ) )
    MEASURE Sales[NumOfProducts] =
        COUNTROWS ( RELATEDTABLE ( 'Product' ) )

EVALUATE

ADDCOLUMNS (
    'Product Category',
    "SubCategories", [NumOfSubcategories],
    "Products Count", [NumOfProducts]
)
```

DAX measures in MDX

An interesting option is the ability to define DAX measures inside an MDX query.

WITH

```
MEASURE 'Internet Sales'[Ship Sales Amount] =
    CALCULATE(
        'Internet Sales'[Internet Total Sales],
        USERELATIONSHIP( 'Internet Sales'[Ship Date Id], 'Date'[Date Id] )
    )

MEMBER Measures.[Ship Sales 2003]
    AS ([Measures].[Ship Sales Amount], [Date].[Calendar Year].&[2003] )
```

SELECT

```
{ Measures.[Ship Sales 2003] }
ON COLUMNS,
NON EMPTY
    [Product Category].[Product Category Name].[Product Category Name]
ON ROWS
FROM [Model]
```

Querying in DAX



Querying is not very complex, but requires a good DAX mood.

Next set of exercises is all about authoring some DAX queries, from simple to more complex ones.

Please refer to **lab number 7** on the exercise book.

Let us go deeper in the analysis of filter contexts

Advanced Filter Context



Advanced filter context

- Data lineage and TREATAS
- Expanded tables
- Arbitrarily shaped filters
- ALLSELECTED and sha

Understanding how DAX remembers about column lineage

Data lineage and TREATAS



Table results in DAX

The result of a table function is a table with results.

The table does not contain only the values, it also maintains the data lineage.

EVALUATE

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

«Audio» is not only a string,
it is the name of a valid
value for Product[Category]

Category
Audio
TV and Video
Computers
Cameras and camcorders
Cell phones
Music, Movies and Audio Books
Games and Toys
Home Appliances

Tables have relationships

A table result has relationships with the model, depending on the data lineage of the columns.

```
EVALUATE  
FILTER (  
    ADDCOLUMNS (  
        CROSSJOIN (  
            VALUES ('Product'[Category]),  
            VALUES ('Date'[Calendar Year])  
        ),  
        "Amt", [Sales Amount]  
    ),  
    [Amt] > 0  
)
```

Category	Calendar Year	Amt
Audio	CY 2007	102,722.07
TV and Video	CY 2007	2,269,589.88
Computers	CY 2007	2,660,318.87
Cameras and camcorders	CY 2007	3,274,847.26
Cell phones	CY 2007	477,451.74
Music, Movies and Audio Books	CY 2007	87,874.44
Games and Toys	CY 2007	89,860.07
Home Appliances	CY 2007	2,347,281.80
Audio	CY 2008	105,363.42
TV and Video	CY 2008	919,946.50
Computers	CY 2008	2,066,341.75

Anonymous tables do not have relationships

An anonymous table does not have relationships with the data model.

```
EVALUATE  
VAR Categories = {  
    "Category",  
    "Audio",  
    "TV and Video",  
    "Computers",  
    "Cameras and camcorders",  
    "Cell phones",  
    "Music, Movies and Audio Books",  
    "Games and Toys",  
    "Home Appliances"  
}  
RETURN  
    ADDCOLUMNS ( Categories, "Amt", [Sales Amount] )
```

Category	Amt
Audio	30,591,343.98
TV and Video	30,591,343.98
Computers	30,591,343.98
Cameras and camcorders	30,591,343.98
Cell phones	30,591,343.98
Music, Movies and Audio Books	30,591,343.98
Games and Toys	30,591,343.98
Home Appliances	30,591,343.98

Column names are not relevant

The column name does not matter. What matters is only the data lineage.

```
EVALUATE  
ADDCOLUMNS (  
    SELECTCOLUMNS (  
        VALUES ( 'Product'[Category] ),  
        "New name for Category", 'Product'[Category]  
    ),  
    "Amt", [Sales Amount]  
)
```

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

Columns have lineage, expressions do not

A reference to a column maintains the data lineage but, as soon as you use an expression, the data lineage is lost and the column becomes anonymous.

```
EVALUATE  
ADDCOLUMNS (  
    SELECTCOLUMNS (  
        VALUES ( 'Product'[Category] ),  
        "New name for Category", 'Product'[Category] & ""  
    ),  
    "Amt", [Sales Amount]  
)
```

New name for Category	Amt
Audio	30,591,343.98
TV and Video	30,591,343.98
Computers	30,591,343.98
Cameras and camcorders	30,591,343.98
Cell phones	30,591,343.98
Music, Movies and Audio Books	30,591,343.98
Games and Toys	30,591,343.98
Home Appliances	30,591,343.98

TREATAS

TREATAS receives a table as its first argument, and one or more column references next. It changes the data lineage of the input table to match the following column references.

```
EVALUATE
VAR Categories = {
    "Category",
    "Audio",
    "TV and Video",
    "Computers and geeky stuff",
    "Cameras and camcorders",
    "Cell phones",
    "Music, Movies and Audio Books",
    "Games and Toys",
    "Home Appliances"
}
RETURN
    ADDCOLUMNS (
        TREATAS (
            Categories,
            'Product'[Category]
        ),
        "Amt", [Sales Amount]
    )
```

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

LASTNONBLANK by customer

TREATAS became useful in the LastBalanceNonBlankPerCustomer measure.
Without it, the expression MAX (Balances[Date]) would not filter the model.

```
LastBalanceNonBlankPerCustomer :=
```

```
CALCULATE (
    SUM ( Balances[Balance] ),
    TREATAS (
        ADDCOLUMNS (
            VALUES ( Balances[Name] ),
            "LastAvailableDate", CALCULATE ( MAX ( Balances[Date] ) )
        ),
        Balances[Name],
        Date[Date]
    )
)
```

Filtering a table filters more than a table

Expanded tables



Filter propagation

Filter is on Products. Subcategory is not filtered and the result is 44.

NumOfSubcategories =

COUNTROWS ('Subcategory')

- Category
- Audio
 - Cameras and camcorders
 - Cell phones
 - Computers
 - Games and Toys
 - Home Appliances
 - Music, Movies and Audio Books
 - TV and Video

44

NumOfSubcategories

- Product Name
- A. Datum Advanced Digital Camera M300 Azure
 - A. Datum Advanced Digital Camera M300 Black
 - A. Datum Advanced Digital Camera M300 Green
 - A. Datum Advanced Digital Camera M300 Grey
 - A. Datum Advanced Digital Camera M300 Orange
 - A. Datum Advanced Digital Camera M300 Pink
 - A. Datum Advanced Digital Camera M300 Silver
 - A. Datum All in One Digital Camera M200 Azure
 - A. Datum All in One Digital Camera M200 Black
 - A. Datum All in One Digital Camera M200 Green
 - A. Datum All in One Digital Camera M200 Grey
 - A. Datum All in One Digital Camera M200 Orange
 - A. Datum All in One Digital Camera M200 Pink
 - A. Datum All in One Digital Camera M200 Silver

Filters are tables

Each filter is a table.

Boolean expressions are just shortcuts for table expressions.

```
CALCULATE (
    SUM ( Sales[SalesAmount] ),
    Sales[SalesAmount] > 100
)
```

Is equivalent to

```
CALCULATE (
    SUM ( Sales[SalesAmount] ),
    FILTER (
        ALL ( Sales[SalesAmount] ),
        Sales[SalesAmount] > 100
    )
)
```

What happens if we
use a full table in a
CALCULATE
expression?

Strange results!

What is happening here?
Cross table filtering is in action.

Filtered By Product =

```
CALCULATE(  
    [NumOfSubcategories],  
    'Product'  
)
```

Category
Audio
Cameras and camcorders
Cell phones
Computers
Games and Toys
Home Appliances
Music, Movies and Audio Books
TV and Video

44

NumOfSubcategories

1

Filtered By Product

Product Name
A. Datum Advanced Digital Camera M300 Azure
A. Datum Advanced Digital Camera M300 Black
A. Datum Advanced Digital Camera M300 Green
A. Datum Advanced Digital Camera M300 Grey
A. Datum Advanced Digital Camera M300 Orange
A. Datum Advanced Digital Camera M300 Pink
A. Datum Advanced Digital Camera M300 Silver
A. Datum All in One Digital Camera M200 Azure
A. Datum All in One Digital Camera M200 Black
A. Datum All in One Digital Camera M200 Green
A. Datum All in One Digital Camera M200 Grey
A. Datum All in One Digital Camera M200 Orange
A. Datum All in One Digital Camera M200 Pink
A. Datum All in One Digital Camera M200 Silver

Key topics

The key to really understand filter context is to understand

- Base Tables
- Expanded Tables

Base tables

Label	Product Name	SubCat
0101001	512MB MP3 Player E51 Silver	1
0101002	512MB MP3 Player E51 Blue	1
0101003	1G MP3 Player E100 White	2
0101004	2G MP3 Player E200 Silver	4
0101005	2G MP3 Player E200 Red	2
0101006	2G MP3 Player E200 Black	3
0101007	2G MP3 Player E200 Blue	3

Left Outer Join

SubCategory Name	
1	MP4&MP3
2	Recorder
3	Radio
4	Recording Pen
5	Headphones
6	Bluetooth Headphones

Expanded tables

Label	Product Name	SubCat
0101001	512MB MP3 Player E51 Silver	1
0101002	512MB MP3 Player E51 Blue	1
0101003	1G MP3 Player E100 White	2
0101004	2G MP3 Player E200 Silver	4
0101005	2G MP3 Player E200 Red	2
0101006	2G MP3 Player E200 Black	3
0101007	2G MP3 Player E200 Blue	3

	SubCategory Name	Cat
1	MP4&MP3	1
1	MP4&MP3	1
2	Recorder	1
4	Recording Pen	1
2	Recorder	1
3	Radio	1
3	Radio	1

Each base table is «expanded» with the columns of related tables following a LEFT OUTER JOIN, as indicated by the existing relationship

Expanded tables

- Each Table
 - Contains all of its columns
 - «*Native Columns*»
 - Plus all of the columns in all related tables
 - «*Related Columns*»
- This does not happen physically, but greatly helps in understanding how filters work
- Hint: RELATED does not follow relationships: it simply grants the access to the related columns.

Sample model

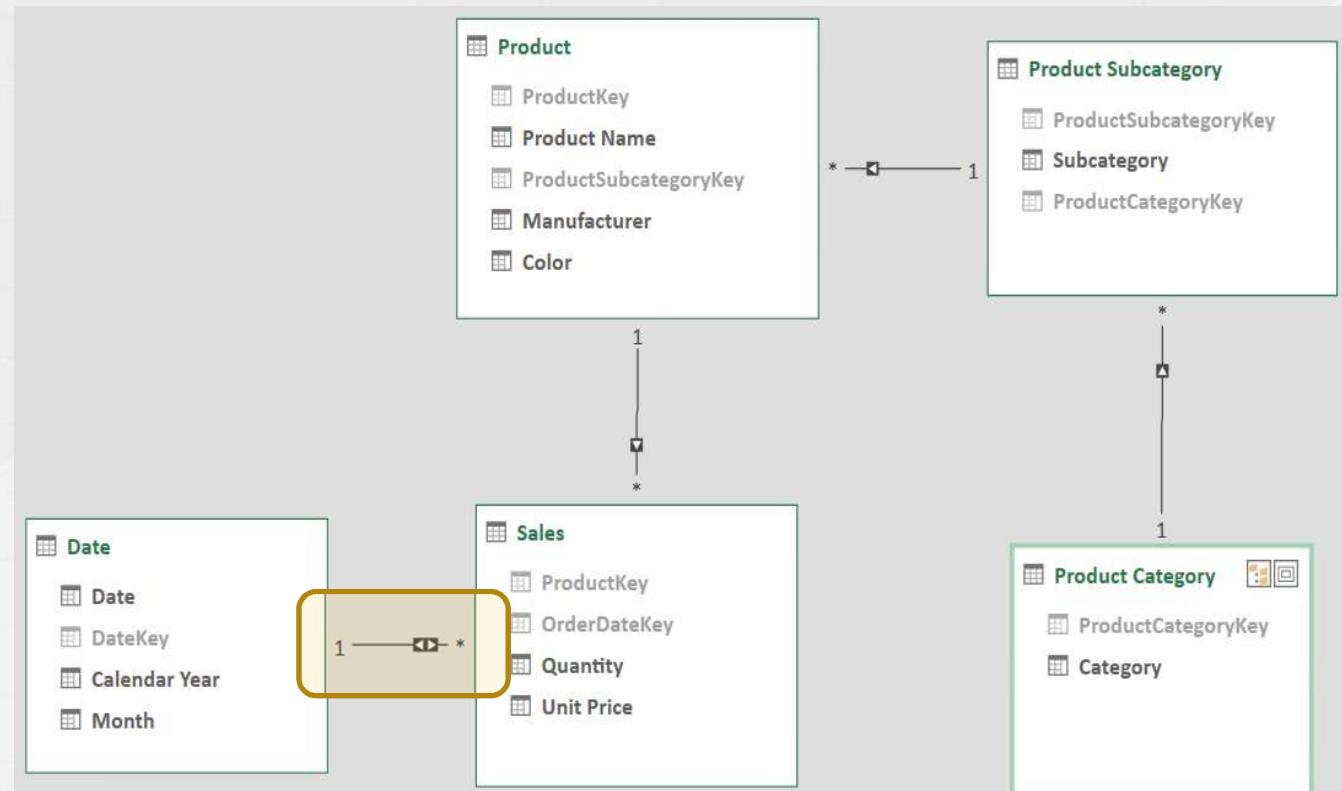


Table expansion

	Product Category	Product Subcategory	Product	Sales	Date
Category					
ProductCategoryKey					
ProductCategoryKey					
Subcategory					
ProductSubcategoryKey					
ProductSubcategoryKey					
Product Name					
Manufacturer					
Color					
ProductKey					
ProductKey					
Unit Price					
Quantity					
OrderDateKey					
DateKey					
Date					
Calendar Year					
Month					

Legend

Native Columns

Derived Columns

Filtering Columns

Context propagation

Thinking at expanded tables, the filter context propagation becomes extremely simple to understand.

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

	Product Category	Product Subcategory	Product	Sales	Date
Category					
ProductCategoryKey					
ProductCategoryKey					
Subcategory					
ProductSubcategoryKey					
Product Name					
Manufacturer					
Color					
ProductKey					
ProductKey					
Unit Price					
Quantity					
OrderDateKey					
DateKey					
Date					
Calendar Year					
Month					

Legend Native Columns Derived Columns Filtering Columns

Back to our formula

Product is applied to the filter context.

It includes columns from both category and subcategory, it has been expanded!

```
CALCULATE (
    COUNTROWS ( 'Product Subcategory' ),
    'Product'
)
```

	Product Category	Product Subcategory	Product
Category			
ProductCategoryKey			
ProductCategoryKey			
Subcategory			
ProductSubcategoryKey			
ProductSubcategoryKey			
Product Name			
Manufacturer			
Color			
ProductKey			

Calculate expanded tables

Tables used by CALCULATE are always expanded when used in the filter context.

Counts the subcategories of the selection.

Filtered By Product =

```
CALCULATE(  
    [NumOfSubcategories],  
    'Product'  
)
```

- Category
- Audio
 - Cameras and camcorders
 - Cell phones
 - Computers
 - Games and Toys
 - Home Appliances
 - Music, Movies and Audio Books
 - TV and Video

44

NumOfSubcategories

1

Filtered By Product

- Product Name
- A. Datum Advanced Digital Camera M300 Azure
 - A. Datum Advanced Digital Camera M300 Black
 - A. Datum Advanced Digital Camera M300 Green
 - A. Datum Advanced Digital Camera M300 Grey
 - A. Datum Advanced Digital Camera M300 Orange
 - A. Datum Advanced Digital Camera M300 Pink
 - A. Datum Advanced Digital Camera M300 Silver
 - A. Datum All in One Digital Camera M200 Azure
 - A. Datum All in One Digital Camera M200 Black
 - A. Datum All in One Digital Camera M200 Green
 - A. Datum All in One Digital Camera M200 Grey
 - A. Datum All in One Digital Camera M200 Orange
 - A. Datum All in One Digital Camera M200 Pink
 - A. Datum All in One Digital Camera M200 Silver

What is happening here?

If the filter is working on the Product table, why is it showing 44?

We learned that filtering Product results in a filter on the related tables, including Category and Subcategory.

What is the slicer filtering?

NumOfSubcategories =

COUNTROWS ('Subcategory')

- Category
- Audio
 - Cameras and camcorders
 - Cell phones
 - Computers
 - Games and Toys
 - Home Appliances
 - Music, Movies and Audio Books
 - TV and Video

44

NumOfSubcategories

- Product Name
- A. Datum Advanced Digital Camera M300 Azure
 - A. Datum Advanced Digital Camera M300 Black
 - A. Datum Advanced Digital Camera M300 Green
 - A. Datum Advanced Digital Camera M300 Grey
 - A. Datum Advanced Digital Camera M300 Orange
 - A. Datum Advanced Digital Camera M300 Pink
 - A. Datum Advanced Digital Camera M300 Silver
 - A. Datum All in One Digital Camera M200 Azure
 - A. Datum All in One Digital Camera M200 Black
 - A. Datum All in One Digital Camera M200 Green
 - A. Datum All in One Digital Camera M200 Grey
 - A. Datum All in One Digital Camera M200 Orange
 - A. Datum All in One Digital Camera M200 Pink
 - A. Datum All in One Digital Camera M200 Silver

Slicers filter columns, not tables

A slicer filters a column, not a table.

By filtering columns, the filter propagates only one-to-many in a standard relationship.

Filtered By Column =

```
CALCULATE (  
    [NumOfSubcategories],  
    VALUES ( 'Product'[Product Name] )  
)
```

- Category
- Audio
 - Cameras and camcorders
 - Cell phones
 - Computers
 - Games and Toys
 - Home Appliances
 - Music, Movies and Audio Books
 - TV and Video

- Product Name
- A. Datum Advanced Digital Camera M300 Azure
 - A. Datum Advanced Digital Camera M300 Black
 - A. Datum Advanced Digital Camera M300 Green
 - A. Datum Advanced Digital Camera M300 Grey
 - A. Datum Advanced Digital Camera M300 Orange
 - A. Datum Advanced Digital Camera M300 Pink
 - A. Datum Advanced Digital Camera M300 Silver
 - A. Datum All in One Digital Camera M200 Azure
 - A. Datum All in One Digital Camera M200 Black
 - A. Datum All in One Digital Camera M200 Green
 - A. Datum All in One Digital Camera M200 Grey
 - A. Datum All in One Digital Camera M200 Orange
 - A. Datum All in One Digital Camera M200 Pink
 - A. Datum All in One Digital Camera M200 Silver

44

NumOfSubcategories

1

Filtered By Product

44

Filtered By Column

Filtering Columns

Filtering a column of DimProduct does not affect the other tables Category and Subcategory, because its columns are not part of other expanded tables.

Filtering a table is very different than filtering a column!

	Product Category	Product Subcategory	Product
Category			
ProductCategoryKey			
ProductCategoryKey			
Subcategory			
ProductSubcategoryKey			
ProductSubcategoryKey			
Product Name			
Manufacturer			
Color			
ProductKey			

Compute a simple percentage

Category

- Audio
- Cameras and camcorders
- Cell phones
- Computers
- Games and Toys
- Home Appliances
- Music, Movies and Audio Books
- TV and Video

Filter on Category

Subcategory	Sales Amount	Pct Correct
Computers Accessories	\$43,756.54	5.34%
Desktops	\$146,317.82	17.85%
Laptops	\$219,571.94	26.79%
Monitors	\$63,507.55	7.75%
Printers, Scanners & Fax	\$55,913.37	6.82%
Projectors & Screens	\$290,528.85	35.45%
Total	\$819,596.07	100.00%

Percentage of
subcategory against
the total of category

The first trial is wrong...

Category	Subcategory	Sales Amount	Pct
□ Audio	Computers Accessories	\$43,756.54	1.14%
□ Cameras and camcorders	Desktops	\$146,317.82	3.83%
□ Cell phones	Laptops	\$219,571.94	5.74%
■ Computers	Monitors	\$63,507.55	1.66%
□ Games and Toys	Printers, Scanners & Fax	\$55,913.37	1.46%
□ Home Appliances	Projectors & Screens	\$290,528.85	7.60%
□ Music, Movies and Audio Books			
□ TV and Video	Total	\$819,596.07	21.43%

```
DIVIDE (
    [Sales Amount],
    CALCULATE (
        [Sales Amount],
        ALL ( ProductSubcategory )
    )
)
```

Wrong formula in action

Category
□ Audio
□ Cameras and camcorders
□ Cell phones
■ Computers
□ Games and Toys
□ Home Appliances
□ Music, Movies and Books
□ TV and Video

Subcategory	Sales	Amount	Pct
Computers Accessories	\$43,756.54	1.14%	
Desktops	\$146,317.82	3.83%	
Laptops	\$219,571.94	5.74%	
Monitors	\$63,507.55	1.66%	
Printers, Scanners & Fax	\$55,913.37	1.46%	
Projectors & Screens	\$290,528.85	7.60%	
Total	\$819,596.07	21.43%	

```
DIVIDE (
    [Sales Amount],
    CALCULATE (
        [Sales Amount],
        ALL ( ProductSubcategory )
    )
)
```

Category	Product Category	Product Subcategory	Product
ProductCategoryKey			
ProductCategoryKey			
Subcategory			
ProductSubcategoryKey			
ProductSubcategoryKey			
Product Name			
Manufacturer			
Color			
ProductKey			

Correct formula

Category
□ Audio
□ Cameras and camcorders
□ Cell phones
■ Computers
□ Games and Toys
□ Home Appliances
□ Music, Movies and Audio Books
□ TV and Video

Subcategory	Sales Amount	Pct Correct
Computers Accessories	\$43,756.54	5.34%
Desktops	\$146,317.82	17.85%
Laptops	\$219,571.94	26.79%
Monitors	\$63,507.55	7.75%
Printers, Scanners & Fax	\$55,913.37	6.82%
Projectors & Screens	\$290,528.85	35.45%
Total	\$819,596.07	100.00%

```
DIVIDE (
    [Sales Amount],
    CALCULATE (
        [Sales Amount],
        ALL ( ProductSubcategory ),
        ProductCategory
    )
)
```

Correct formula in action

Category
□ Audio
□ Cameras and camcorders
□ Cell phones
■ Computers
□ Games and Toys
□ Home Appliances
□ Music, Movie and Audio Books
□ TV and Video

Subcategory	Sales Amount	Pct Correct
Computers Accessories	\$43,756.54	5.34%
Desktops	\$146,317.82	17.85%
Laptops	\$219,571.94	26.79%
Monitors		7.75%
Printers, Scanners & Fax	\$55,913.37	6.82%
Projectors & Screens	\$290,528.85	35.45%
Total	\$819,596.07	100.00%

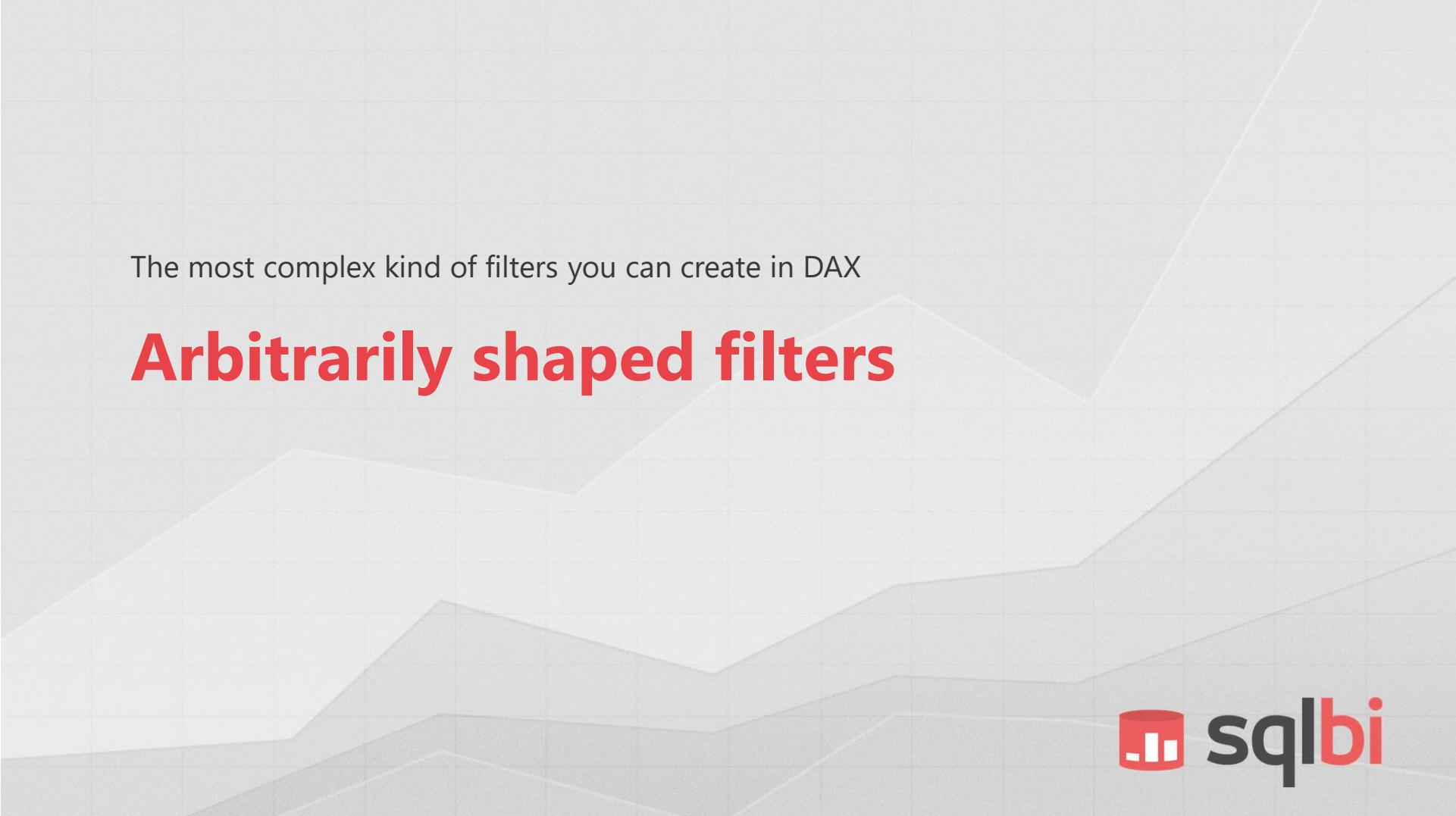
DIVIDE ([Sales Amount],
 CALCULATE ([Sales Amount],
 ALL (ProductSubcategory),
 ProductCategory))

Category	Product Category	Product Subcategory	Product
ProductCategoryKey			
ProductCategoryKey			
Subcategory			
ProductSubcategoryKey			
ProductSubcategoryKey			
Product Name			
Manufacturer			
Color			
ProductKey			

Same result using ALLEXCEPT

You can write the same code with ALLEXCEPT, removing filters from ProductSubcategory but not from the table ProductCategory that is part of ProductSubcategory expanded table.

```
DIVIDE (
    [Sales Amount],
    CALCULATE (
        [Sales Amount],
        ALLEXCEPT ( ProductSubcategory, ProductCategory )
    )
)
```



The most complex kind of filters you can create in DAX

Arbitrarily shaped filters



Filter equivalency



Year	Month
2007	January
2007	February
2008	January
2008	February



Year	Month
2007	January
2008	February

A simple filter can be simplified as the intersection of single-column filters

Arbitrarily shaped filters

Example of a filter context built with complex features of the UI. Can be created in DAX, too.

Calendar Year	Calendar Year	Sales Amount
<input type="checkbox"/> June	CY 2007	1,817,150.62
<input type="checkbox"/> July	November	825,601.87
<input type="checkbox"/> August	December	991,548.75
<input type="checkbox"/> September		
<input type="checkbox"/> October		
<input checked="" type="checkbox"/> November	CY 2008	1,256,846.69
<input checked="" type="checkbox"/> December	January	656,766.69
<input checked="" type="checkbox"/> CY 2008	February	600,080.00
<input checked="" type="checkbox"/> January		
<input checked="" type="checkbox"/> February		
<input type="checkbox"/> March	Total	3,073,997.31

Arbitrarily shaped filters in code

```
CALCULATE (
    ...
    FILTER (
        CROSSJOIN (
            ALL ( Date[Year] ),
            ALL ( Date[Month] )
        ),
        OR (
            AND ( Date[Year] = 2007, Date[Month] = "December" ),
            AND ( Date[Year] = 2008, Date[Month] = "January" )
        )
    )
)
```



Year	Month
2006	December
2007	January

Arbitrarily shaped filters

- Cannot be simplified
- Stores the relationship between the columns
- In the example, month depends from year:
 - 2007 → December
 - 2008 → January
- Need to understand how the filter operators handle arbitrarily shaped sets



Year	Month
2007	December
2008	January

Average of monthly sales

Simple measure that computes the monthly average of sales.

Monthly Avg Sales :=

```
AVERAGEX (
    VALUES ( 'Date'[Month] ),
    [Sales Amount]
)
```

Calendar Year	Sales Amount	Monthly Avg Sales
CY 2007	1,817,150.62	908,575.31
November	825,601.87	825,601.87
December	991,548.75	991,548.75
CY 2008	1,256,846.69	628,423.35
January	656,766.69	656,766.69
February	600,080.00	600,080.00
Total	3,073,997.31	1,709,299.98

OVERWRITE and complex filters

Overwrite destroys arbitrarily shaped filters



Use unique columns when iterating

The best solution for this scenario is to make sure that the column being iterated contains unique values. In our case, a column with year and month fits perfectly.

Monthly Avg Sales :=

```
AVERAGEX (
    VALUES ( 'Date'[Calendar Year Month] ),
    [Sales Amount]
)
```

	Calendar Year	Sales Amount	Monthly Avg Sales
CY 2007		1,817,150.62	908,575.31
November		825,601.87	825,601.87
December		991,548.75	991,548.75
CY 2008		1,256,846.69	628,423.35
January		656,766.69	656,766.69
February		600,080.00	600,080.00
Total		3,073,997.31	768,499.33

KEEPFILTERS

Another option is to use **KEEPFILTERS**, which instructs CALCULATE to use INTERSECT instead of OVERWRITE to merge the new filter context with the previous one.

```
CALCULATE (
    ....,
    KEEPFILTERS ( Color = "Red" )
)

SUMX (
    KEEPFILTERS ( Products ),
    ...
)
```

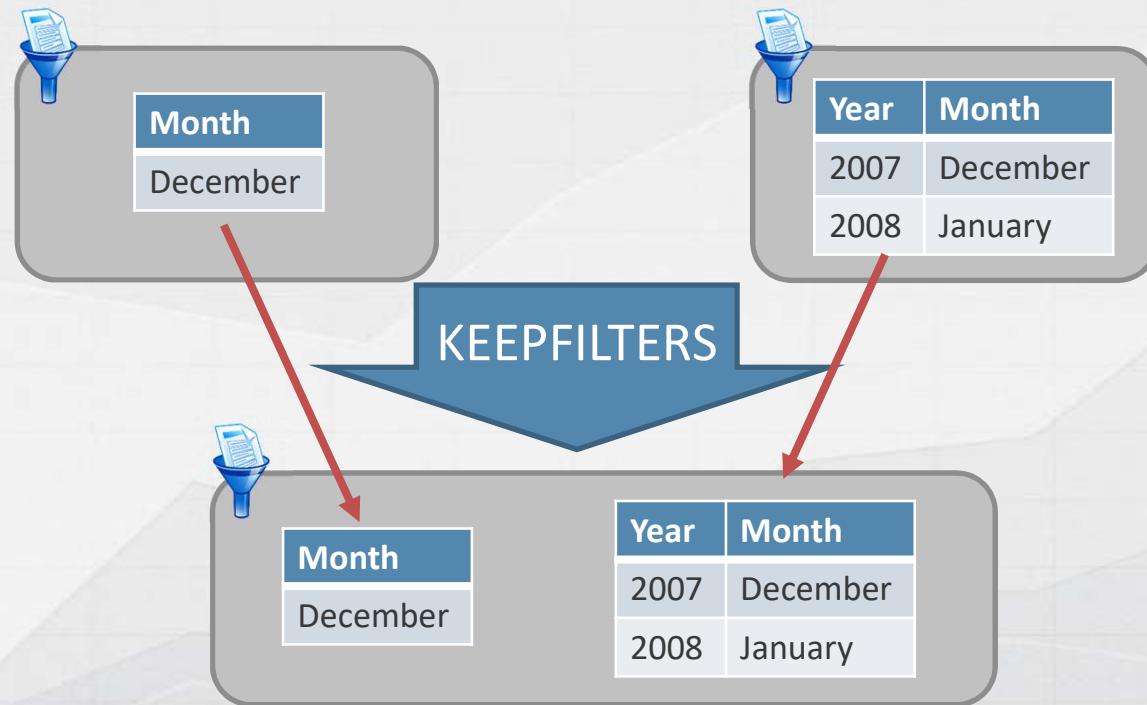
KEEPFILTERS with arbitrarily shaped filters

KEEPFILTERS modifies the behavior of CALCULATE (and the context transition happening during iteration).

```
Monthly Avg Sales :=  
  
AVERAGEX (  
    KEEPFILTERS (  
        VALUES ( 'Date'[Month] )  
    ),  
    [Sales Amount]  
)
```

KEEPFILTERS and complex filters

KEEPFILTERS preserves arbitrarily shaped filters



ALLSELECTED looks like a simple function... don't be fooled!

ALLSELECTED and shadow filter contexts



ALLSELECTED

Intuitively, ALLSELECTED is useful to restore the filter context outside of the current visual, taking into account slicers and ignoring rows and columns.

```
Pct :=  
DIVIDE (  
    [Sales Amount],  
    CALCULATE (  
        [Sales Amount],  
        ALLSELECTED ('Product'[Brand])  
    )  
)
```

Kept filters

- Brand
 - A. Datum
 - Adventure Works
 - Contoso
 - Fabrikam
 - Litware
 - Northwind Traders
 - Proseware
 - Southridge Video
 - Tailspin Toys
 - The Phone Company
 - Wide World Importers

Removed filters

Brand	Sales Amount	Pct
Adventure Works	4,011,112.28	16.88%
Contoso	7,352,399.03	30.94%
Fabrikam	5,554,015.73	23.38%
Litware	3,255,704.03	13.70%
Northwind Traders	1,040,552.13	4.38%
Proseware	2,546,144.16	10.72%
Total	23,759,927.34	100.00%

ALLSELECTED executed in a query

When Power BI runs the report, it executes a query very similar to this one. The percentages are against the selection of brands, not all the brands. As you see, ALLSELECTED still works fine.

```
EVALUATE
VAR Brands =
    TREATAS ( {
        "Adventure Works", "Contoso", "Fabrikam",
        "Northwind Traders", "Proseware", "Litware"
    },
    'Product'[Brand]
)
RETURN
    CALCULATETABLE (
        ADDCOLUMNS (
            VALUES ( 'Product'[Brand] ),
            "Sales_Amount", [Sales Amount],
            "Pct", [Pct]
        ),
        Brands
    )
```

Brand	Sales_Amount	Pct
Contoso	7,352,399.03	30.94%
Northwind Traders	1,040,552.13	4.38%
Adventure Works	4,011,112.28	16.88%
Litware	3,255,704.03	13.70%
Fabrikam	5,554,015.73	23.38%
Proseware	2,546,144.16	10.72%

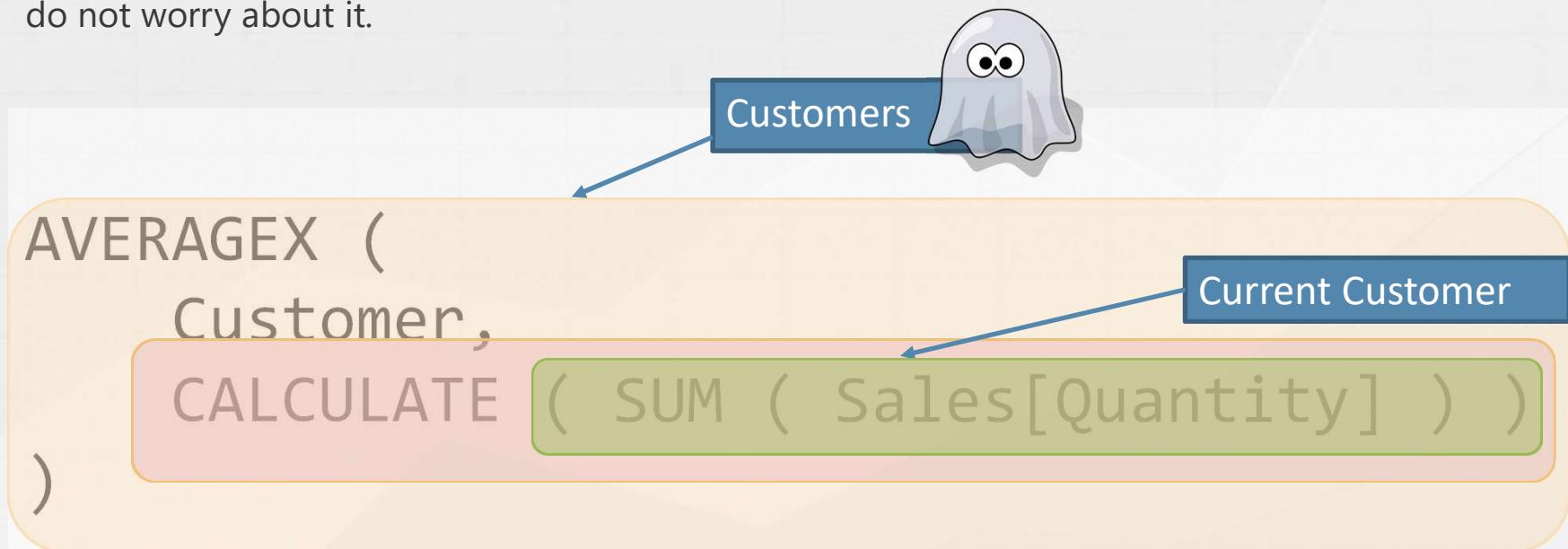


Shadow filter contexts

- In fact, ALLSELECTED does not know about the report
- This is what ALLSELECTED does:
 - When used as a table function, it returns the set of values as visible in the last *shadow filter context*
 - When used as a CALCULATE modifier, it restores the last *shadow filter context* on its parameter
- Time to discover what a shadow filter context is

What is a shadow filter context?

Iterators generate a shadow filter context. If you use CALCULATE, the innermost filter context is always more restrictive than the outer one. Thus, the shadow is **almost** useless and you typically do not worry about it.

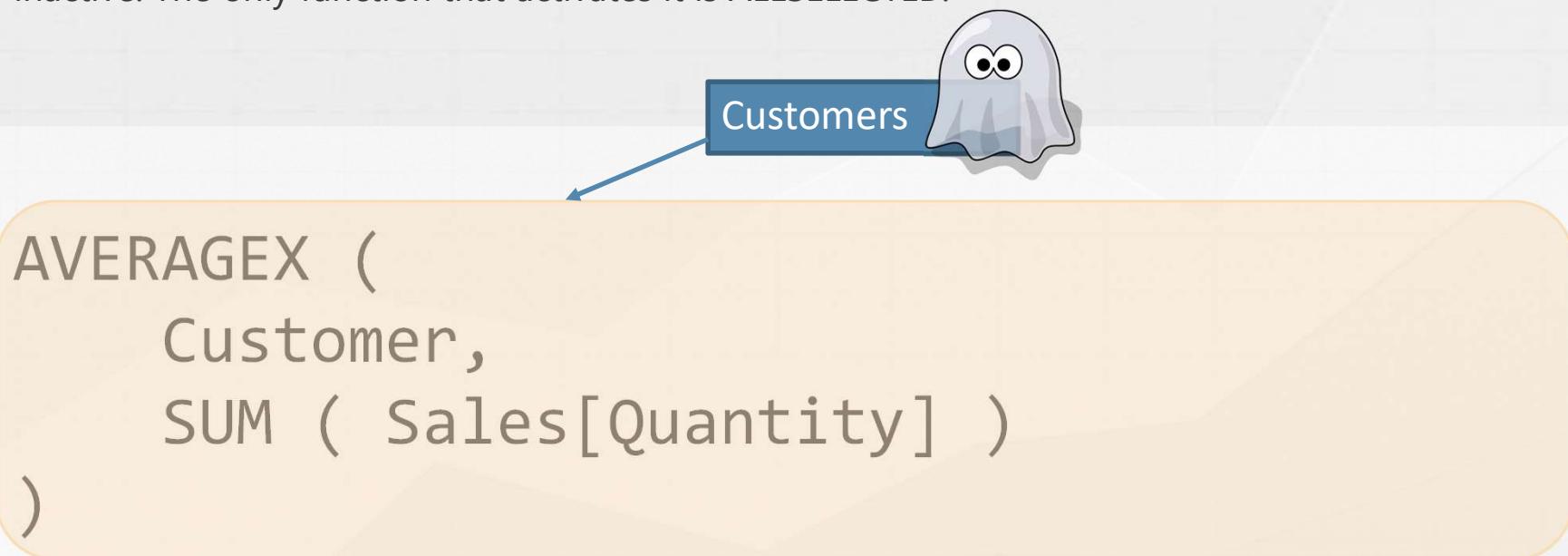


The shadow filter context generated by AVERAGEX is inactive



The shadow filter context is not active

Inside the iteration, you still see the outer filter context, because the shadow filter context is inactive. The only function that activates it is ALLSELECTED.



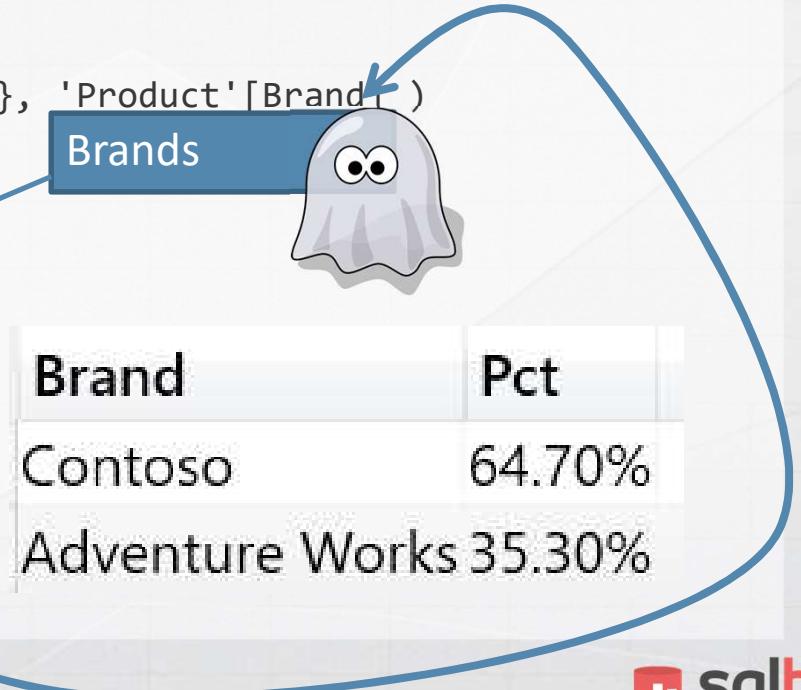
SUM is computed in the outer context, the shadow filter is inactive



ALLSELECTED restores a shadow filter context

ADDCOLUMNS creates a shadow filter context containing the two selected brands.
When ALLSELECTED is invoked, it restores the shadow filter context.

```
EVALUATE  
VAR Brands =  
    TREATAS ( { "Adventure Works", "Contoso" }, 'Product'[Brand] )  
RETURN  
    CALCULATETABLE (  
        ADDCOLUMNS (  
            VALUES ( 'Product'[Brand] ),  
            "Pct", DIVIDE (  
                [Sales Amount],  
                CALCULATE (  
                    [Sales Amount],  
                    ALLSELECTED ()  
                )  
            )  
        ),  
        Brands  
    )
```



Brand	Pct
Contoso	64.70%
Adventure Works	35.30%

ALLSELECTED restores a shadow filter context

If ADDCOLUMNS iterates over ALL brands, then ALLSELECTED restores all the brands, regardless of the outer filter created by CALCULATETABLE.

```
EVALUATE  
VAR Brands =  
    TREATAS ( { "Adventure Works", "Contoso" }, 'Product'[Brand] )  
RETURN  
    CALCULATETABLE (  
        ADDCOLUMNS (   
            ALL ( 'Product'[Brand] ),  
            "Pct", DIVIDE ( [Sales Amount],  
                CALCULATE ( [Sales Amount],  
                    ALLSELECTED () )  
            )  
        ),  
        Brands  
    )
```



ALLSELECTED and KEEPFILTERS

When the innermost CALCULATE applies its filter, it restores the shadow filter context. At the same time, it intersects the shadow filter with the outer filter, iterating over all brands but computing over selected brands only.

```
EVALUATE  
VAR Brands =  
    TREATAS ( { "Adventure Works", "Contoso" }, 'Product'[Brand] )  
RETURN  
    CALCULATETABLE (  
        ADDCOLUMNS (,  
            KEEPFILTERS ( ALL ( 'Product'[Brand] ) ),  
            "Pct", DIVIDE ( [Sales Amount],  
                CALCULATE ( [Sales Amount],  
                    ALLSELECTED ()  
                )  
            )  
        ),  
        Brands  
    )
```

ALL Brands

Brand	Pct
Contoso	64.70%
Wide World Importers	
Northwind Traders	
Adventure Works	35.30%
Southridge Video	
Litware	
Fabrikam	

ALLSELECTED rules

- In order to work, KEEPFILTERS restores the outermost iteration generated by ADDCOLUMNS or SUMMARIZECOLUMNS
- It works only if the measure is invoked as the topmost measure in the query
- Nesting ALLSELECTED inside other iterations generates very complex results, almost unpredictable

Advanced filter context



Next exercise session is composed of only a couple of formulas, nothing complex. To solve them, you need to stretch your mind and use all the knowledge of this last module. Good luck and... don't look at the solution too early.

Please refer to **lab number 9** on the hands-on manual.

Let's move a step further from plain vanilla relationships

Advanced Relationships



Complex relationships

- Not all the relationships are one-to-many relationships based on a single column
- When relationships are complex ones, more DAX skills are needed
- Data modeling, in SSAS Tabular and Power BI, is very basic
- DAX is the key to unleash more analytical power

Static Segmentation



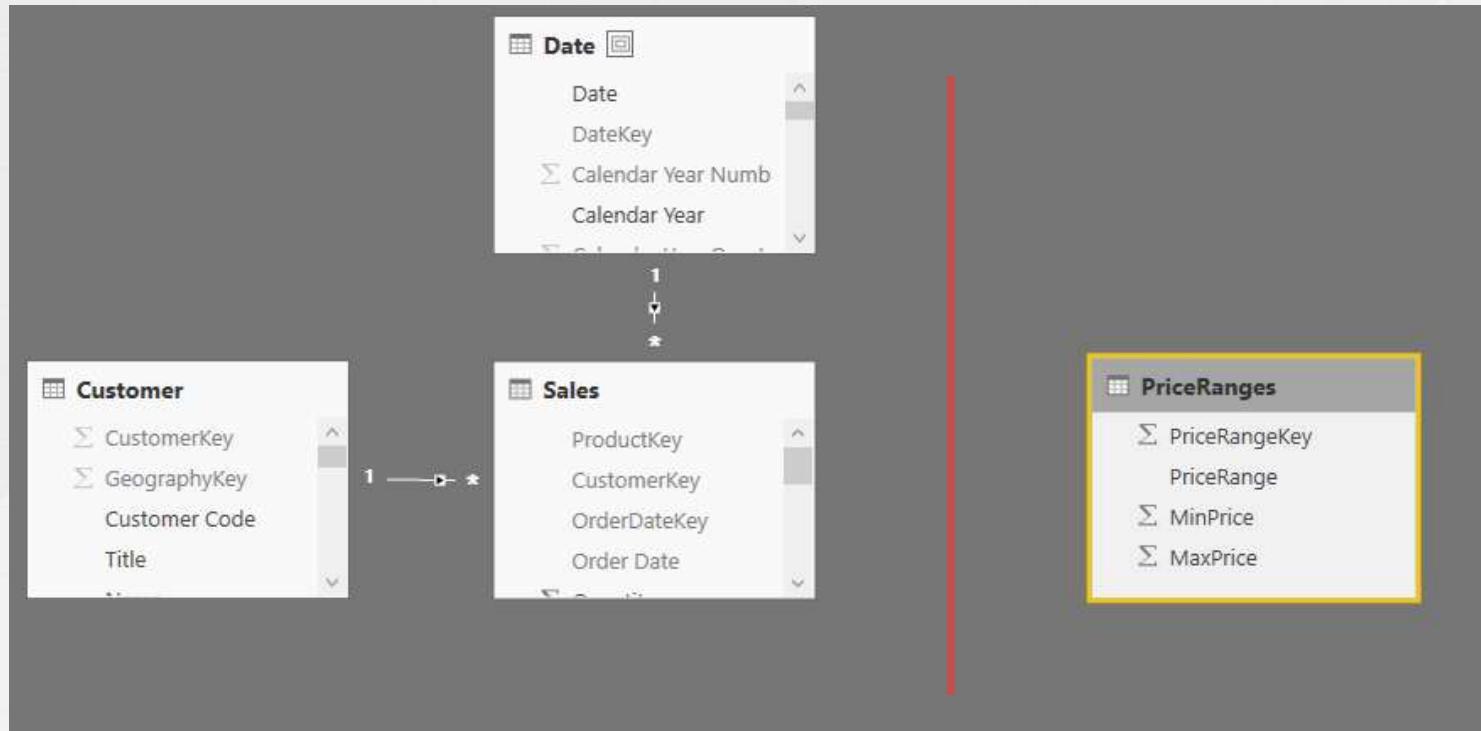
Static segmentation

- Analysis of sales based on unit price

PriceRangeKey	PriceRange	MinPrice	MaxPrice
1	VERY LOW	0	10
2	LOW	10	30
3	MEDIUM	30	80
4	HIGH	80	150
5	VERY HIGH	150	99999

PriceRange	Sales	Amount
VERY LOW	\$19,405.17	
LOW	\$66,541.73	
MEDIUM	\$116,174.19	
HIGH	\$287,543.57	
VERY HIGH	\$3,730,253.24	
Total	\$4,219,917.90	

Static segmentation: the model



The quick and dirty solution

Quick and dirty, but it works.

Very useful for prototyping a data model with the customer.

```
= IF (
    Sales[UnitPrice] <= 5,
    "01 LOW",
    IF (
        Sales[UnitPrice] <=30,
        "02 MEDIUM LOW",
        IF (
            Sales[UnitPrice] <=100,
            "03 MEDIUM HIGH",
            IF (
                Sales[UnitPrice] <= 500,
                "04 HIGH",
                "05 VERY HIGH" ) ) ) )
```

Static segmentation: the formula

By using a calculated column you can denormalize the price segment in the fact table.
Being a small-cardinality column, the size in RAM is very small.

```
Sales[PriceRange] =  
  
CALCULATE (  
    VALUES ( PriceRanges[PriceRange] ),  
    FILTER (  
        PriceRanges,  
        AND (  
            PriceRanges[MinPrice] <= Sales[Net Price],  
            PriceRanges[MaxPrice] > Sales[Net Price]  
        )  
    )  
)
```

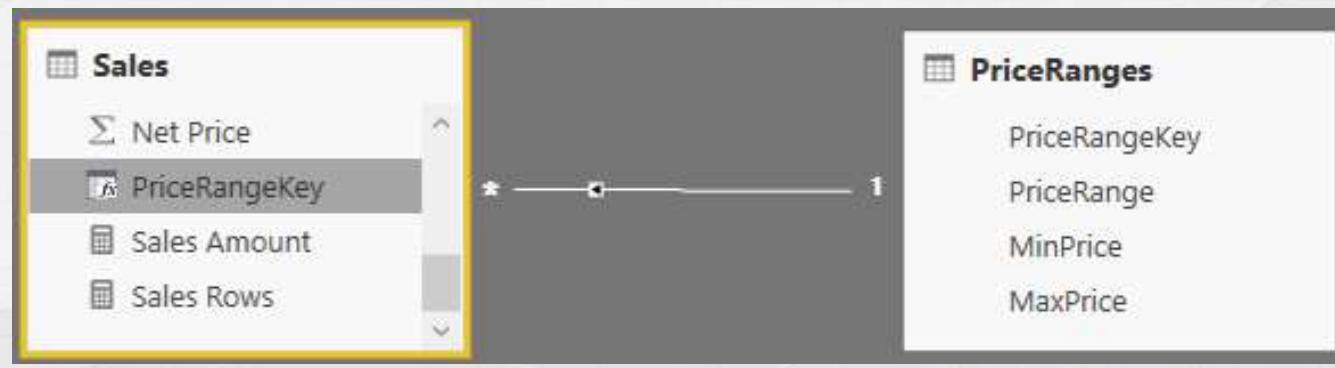
Denormalizing the key

Using a similar technique you can denormalize the key of the table to avoid replicating all of the columns. Pay attention to circular references, this is why we need to use DISTINCT or ALLNOBLANKROW instead of VALUES or ALL.

```
PriceRangeKey =  
  
CALCULATE (  
    DISTINCT ( PriceRanges[PriceRangeKey] ),  
    FILTER (  
        PriceRanges,  
        AND (  
            PriceRanges[MinPrice] <= Sales[Net Price],  
            PriceRanges[MaxPrice] > Sales[Net Price]  
        )  
    )  
)
```

Circular dependency in calculated tables

- Sales depends on PriceRanges
- If you build the relationship, the model contains a circular dependency, because – on the one side – the blank row might appear. Thus, PriceRanges depends on Sales.
- You need to use functions that do not use the blank row: DISTINCT, ALLNOBLANKROW



Denormalizing keys

- Denormalizing the key goes further than you might think at first

Using DAX, you can build any kind of relationship, no matter how fancy it is, as long as you can compute it in a DAX expression

- This is an extremely powerful technique, we call it **Calculated Relationships**

Dynamic Segmentation



Dynamic segmentation

SegmentCode	Segment	MinSale	MaxSale
1	Very Low	0	75
2	Low	75	100
3	Medium	100	500
4	High	500	1000
6	Very High	1000	999999999

This time, the relationship depends on what you put on rows and columns. In this demo, the segmentation changes over time.

Segment	CY 2007	CY 2008	CY 2009	Total
Very Low	351	266	255	810
Low	141	14	12	166
Medium	365	76	52	485
High	250	36	35	311
Very High	302	132	160	581
Total	1,409	524	514	2,353

Dynamic segmentation: the formula

Dynamic segmentation requires you to build the code in a measure, because the relationship cannot be created as a static calculated column.

```
CustInSegment :=  
  
COUNTRROWS (  
    FILTER (  
        Customer,  
        AND (  
            [Sales Amount] > MIN ( Segments[MinSale] ),  
            [Sales Amount] <= MAX ( Segments[MaxSale] )  
        )  
    )  
)
```

Segment	CY 2007	CY 2008	CY 2009	Total
Very Low	351	266	255	810
Low	141	14	12	166
Medium	365	76	52	485
High	250	36	35	311
Very High	302	132	160	581
Total	1,409	524	514	2,353

Beware of slicers...

- The previous formula is error-prone
- If the user filters data using a slicer, the selection can break the calculation (see wrong Total row)

Segment	Segment	CY 2007	CY 2008	CY 2009	Total
Very Low	Very Low	351	266	255	810
Medium	Very High	302	132	160	581
High	Total	1,409	524	514	2,353
Very High					

Dynamic segmentation: a better formula

By iterating segments and avoiding MIN and MAX, the Total sums only the visible segments.

```
CustInSegment :=  
SUMX (  
    Segments,  
    COUNTROWS (  
        FILTER (  
            Customer,  
            AND (  
                [Sales Amount] > Segments[MinSale],  
                [Sales Amount] <= Segments[MaxSale]  
            )  
        )  
    )  
)
```

Segment
Very Low
Low
Medium
High
Very High

Segment	CY 2007	CY 2008	CY 2009	Total
Very Low	351	266	255	810
Very High	302	132	160	581
Total	653	398	415	1,391

The measure is still
non-additive over years

Many ways to handle many-to-many relationships

Many-to-many



Many-to-many relationships

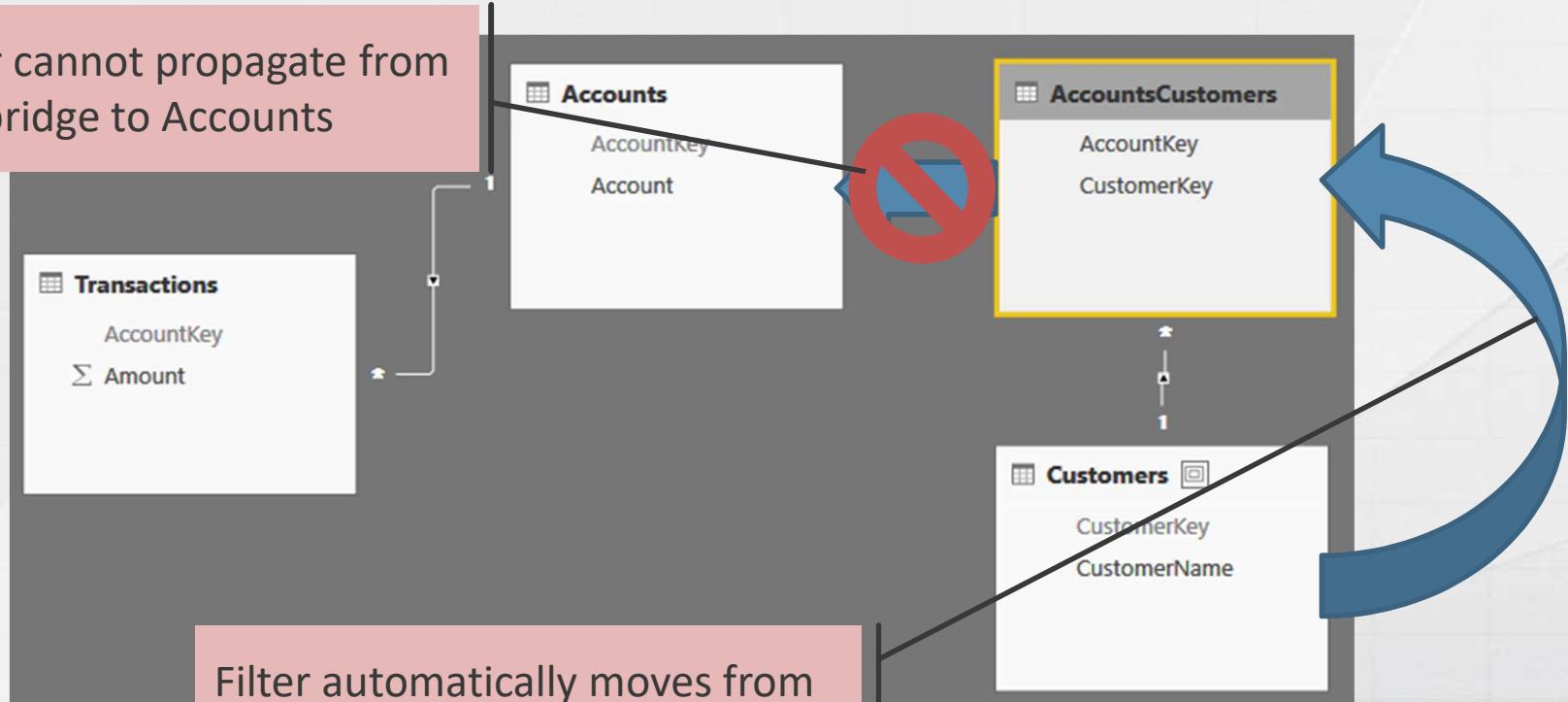
- Whenever a relationship is not just one-to-many
- Examples
 - Bank current account and holders
 - Companies and shareholders
 - Teams, roles and members
 - House and householders
- They are a powerful tool, not an issue at all
- Learning how to use them opens the doors to very powerful scenarios

Many-to-many challenges

- M2M do not work by default
 - You need to write code or update the model
- M2M generate non-additive calculations
 - Hard to understand at first glance
 - Yet, it is in their nature to be non-additive
- Performance might be an issue
 - Not always, but you need to pay attention to details

Current account example of M2M

Filter cannot propagate from the bridge to Accounts



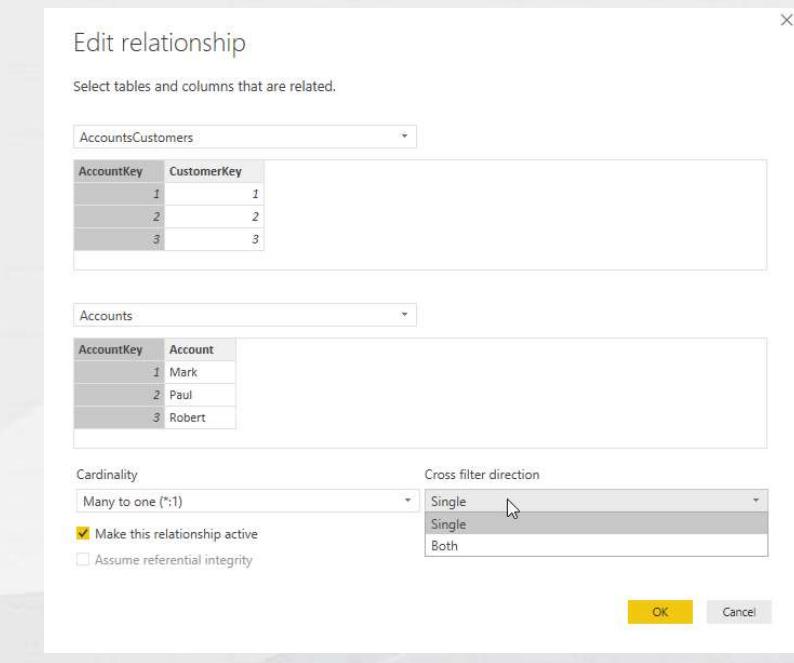
Filter automatically moves from Customers to the bridge

Possible solutions to the scenario

- Bidirectional filtering
 - Available in Power BI and SSAS 2016
- Using CROSSFILTER
 - Available in Power BI, SSAS 2016 and Excel 2016
- Expanded table filtering
 - Available in any version of DAX
- Each solution has pros and cons

Bidirectional filtering

- Enabled at the relationship level
- Let the filter context propagate both ways
- Works with any measure in the model: fewer coding means fewer errors



CustomerName	Amount	SumOfAmount
Luke	\$5,000.00	\$800.00
Mark	\$5,000.00	\$2,800.00
Paul	\$5,000.00	\$1,700.00
Robert	\$5,000.00	\$1,700.00
Total	\$5,000.00	\$5,000.00

Using CROSSFILTER

Changes the direction of a relationship for the duration of a CALCULATE statement.
This pattern must be used in every measures requiring the many-to-many behavior.

```
SumOfAmount CrossFilter =  
  
CALCULATE (  
    SUM ( Transactions[Amount] ),  
    CROSSFILTER (  
        AccountsCustomers[AccountKey],  
        Accounts[AccountKey],  
        BOTH  
    )  
)
```

Accounts with no customer

- Accounts with no customers do not appear when slicing by customer, apart from the grand-total
- Slicing by account, all the accounts are visible

CustomerName	Sum Of Amt	Account	Sum Of Amt
Luke	800.00		2,450.00
Mark	2,800.00	Luke	800.00
Paul	1,700.00	Mark	800.00
Robert	1,700.00	Mark-Paul	1,000.00
Total	7,450.00	Mark-Robert	1,000.00
		Paul	700.00
		Robert	700.00
		Total	7,450.00

Using expanded table filtering

You can obtain the same result by leveraging table expansion and filter context instead of enabling bidirectional filter on relationships. It is useful for older versions of DAX.

```
AmountM2M :=  
  
CALCULATE (  
    SUM ( Transaction[Amount] ),  
    AccountCustomer  
)
```

MANY-TO-MANY with SUMMARIZE

You can use SUMMARIZE and move the filter from AccountCustomer (already filtered by customer) and summarizing it by Account, so that the Account key filters the fact table.

```
AmountM2M :=  
  
CALCULATE (  
    SUM ( Transaction[Amount] ),  
    SUMMARIZE (  
        AccountCustomer,  
        Account[ID_Account]  
    )  
)
```

CROSSFILTER versus expanded tables

- Using CROSSFILTER the filter is propagated only when some filtering is happening
- Using expanded tables, the filter is always active
 - Optimization for expanded tables described here:
<https://www.sqlbi.com/articles/many-to-many-relationships-in-power-bi-and-excel-2016/>

CustomerName	SumOfAmount CrossFilter	SumOfAmount Table Expansion
Luke	\$800.00	\$800.00
Mark	\$2,800.00	\$2,800.00
Paul	\$1,700.00	\$1,700.00
Robert	\$1,700.00	\$1,700.00
Total	\$7,450.00	\$5,000.00

Comparison of the different techniques

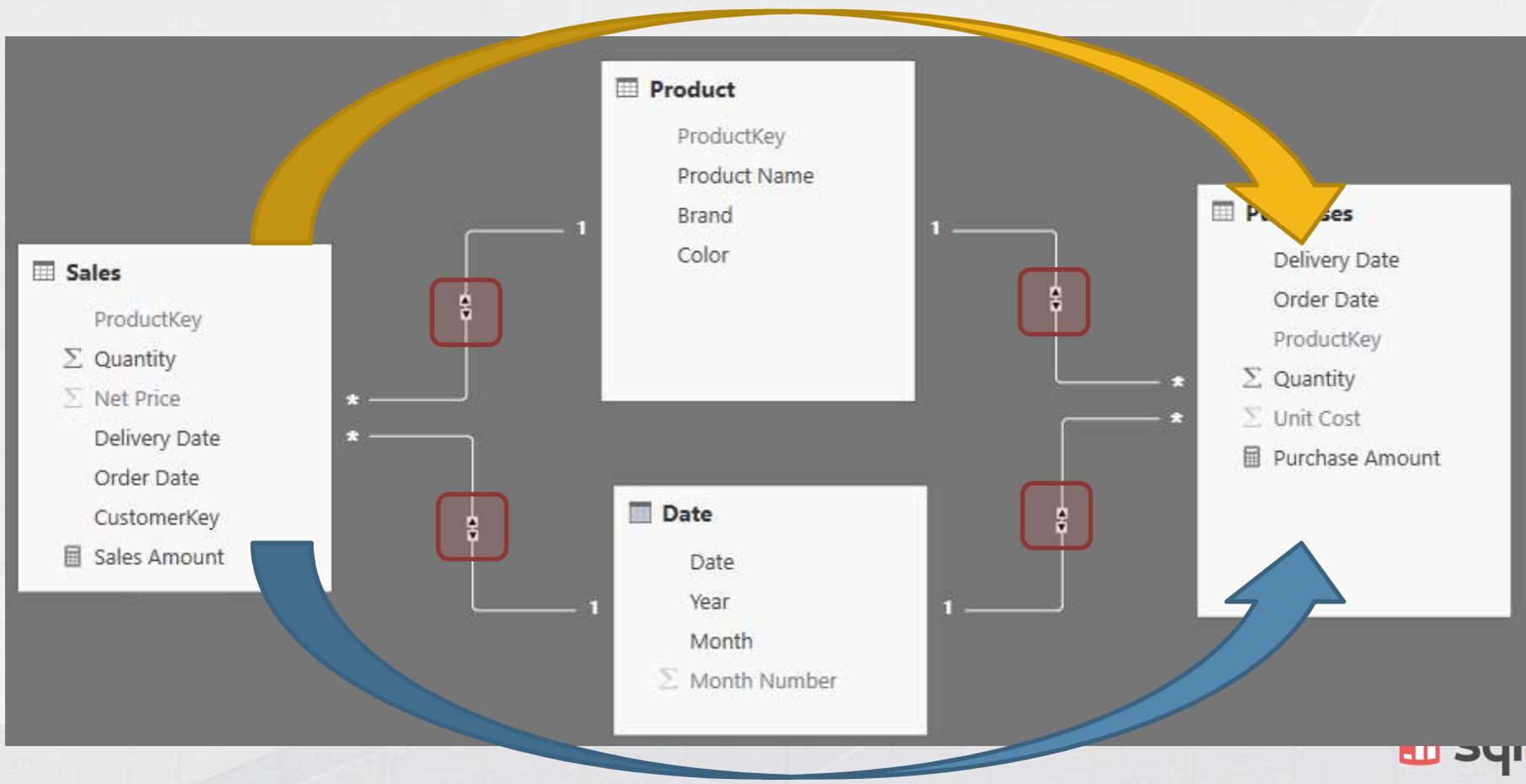
- Bidirectional filtering
 - Set in the model
 - Works with any measure
 - Might introduce ambiguity (discussed later)
- Using CROSSFILTER
 - Set in the formula
 - Requires additional coding
- Expanded table filtering
 - Set in the formula
 - Works on any version of DAX
 - Filter is always active, might slow down the model

Bidirectional filters look promising, but you need to avoid creating complex models

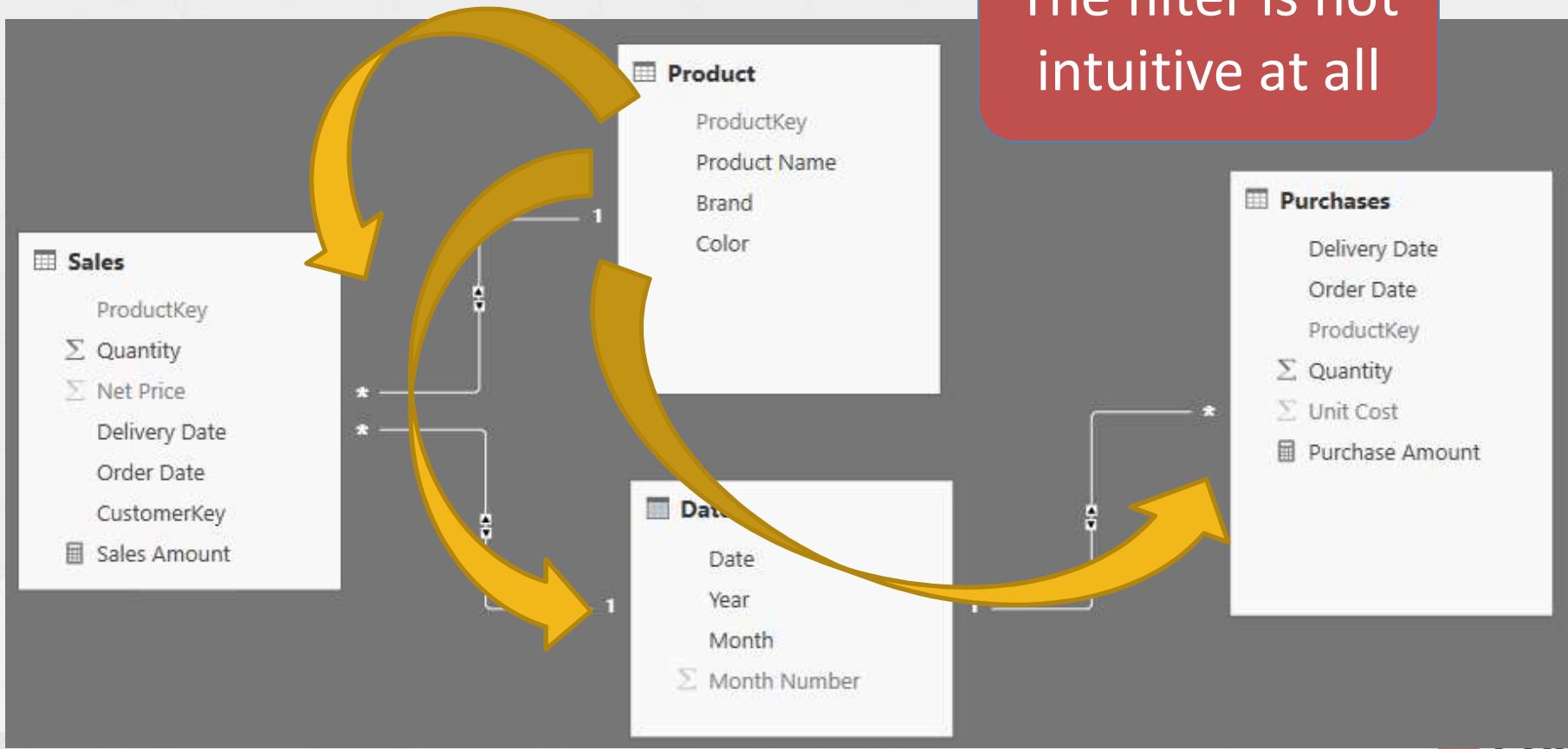
Ambiguity and bidirectional filters



What is ambiguity?



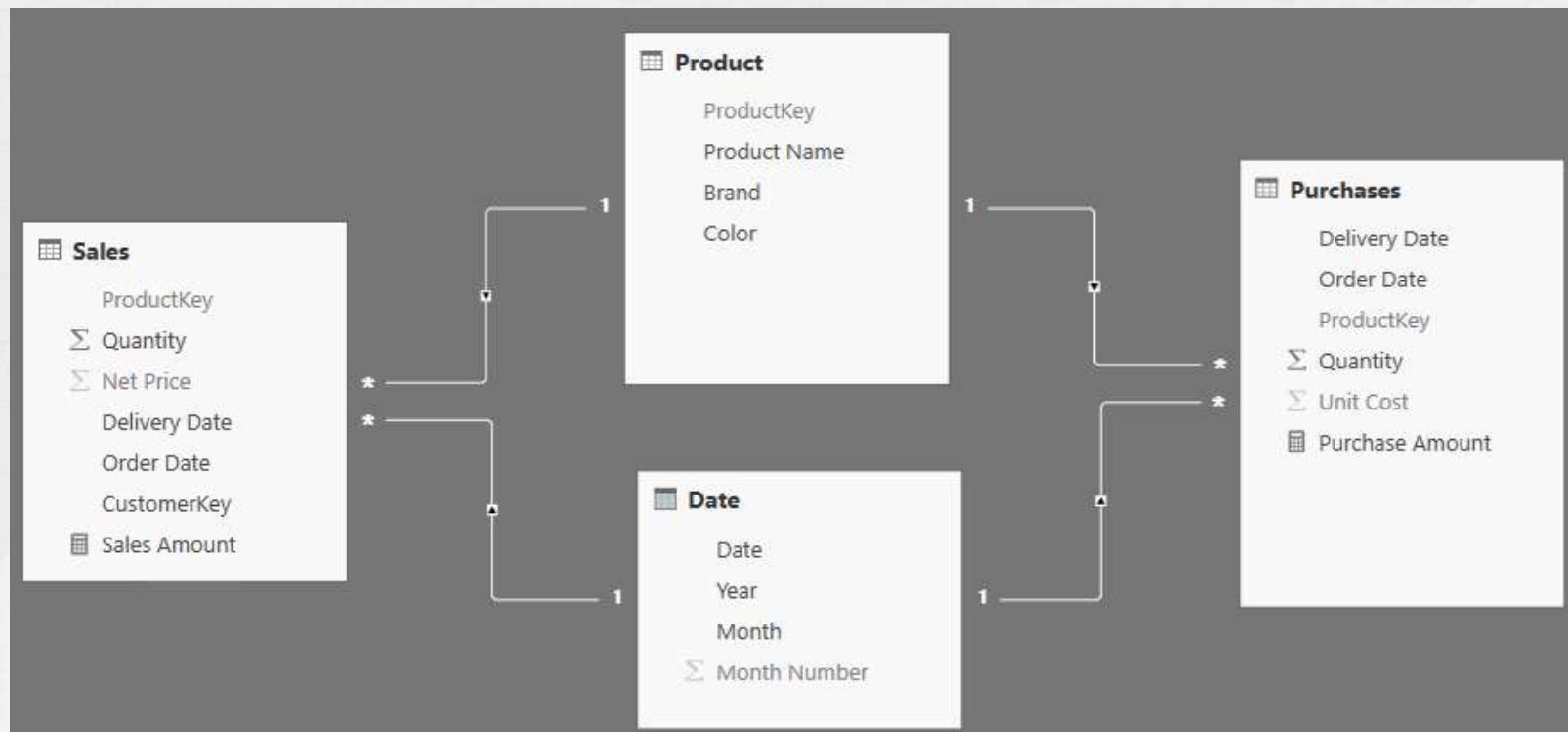
Solving ambiguity



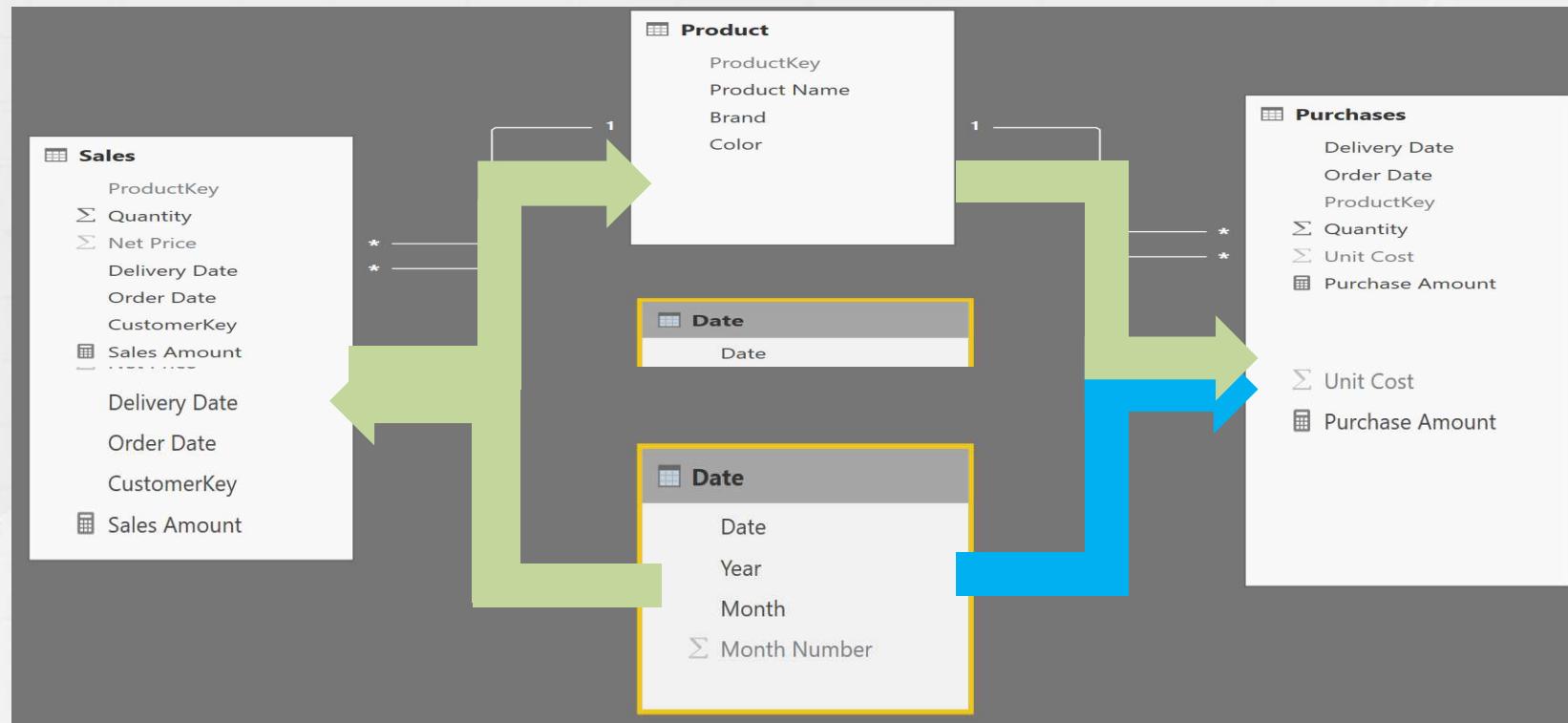
Relationship rules

- Single star schema
 - Enable bidirectional filtering
 - Beware of performance
- Any other model, including multiple star schemas
 - Keep all relationship unidirectional
 - Enable bidirectional when needed
 - **Only** when needed

Correct model, only unidirectional relationships



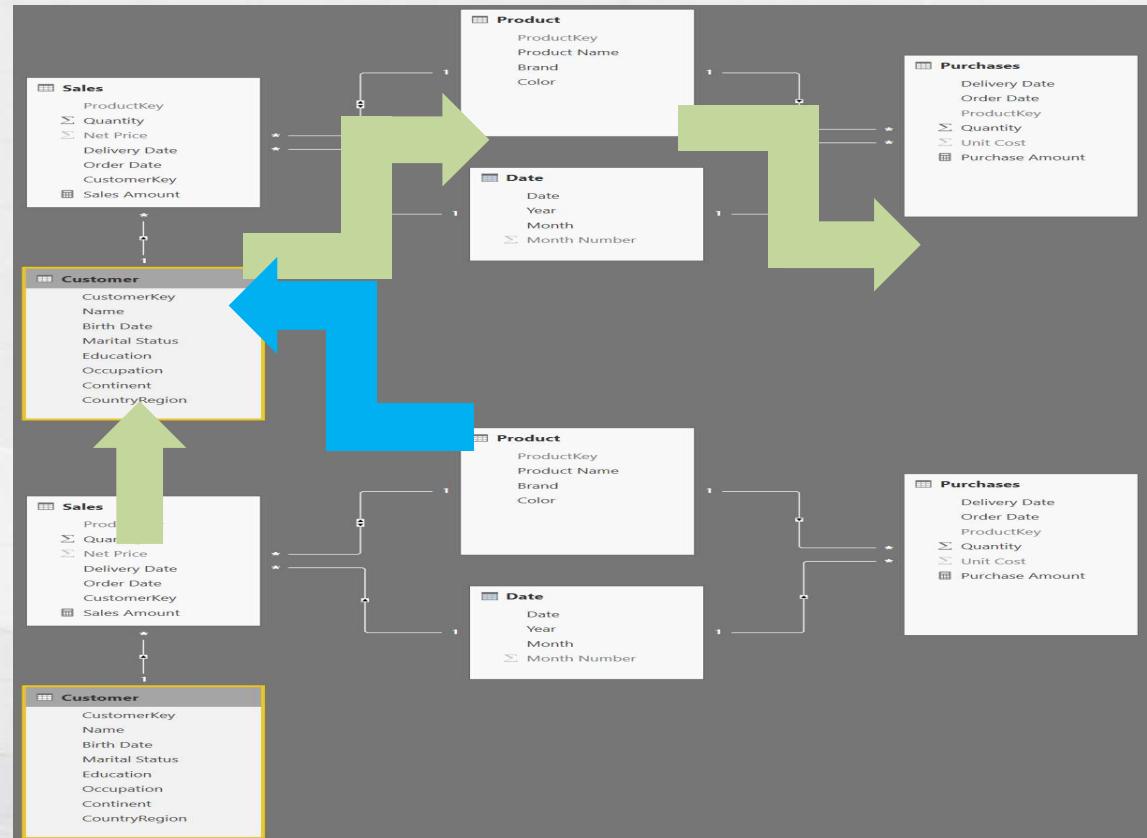
Is this an ambiguous model?



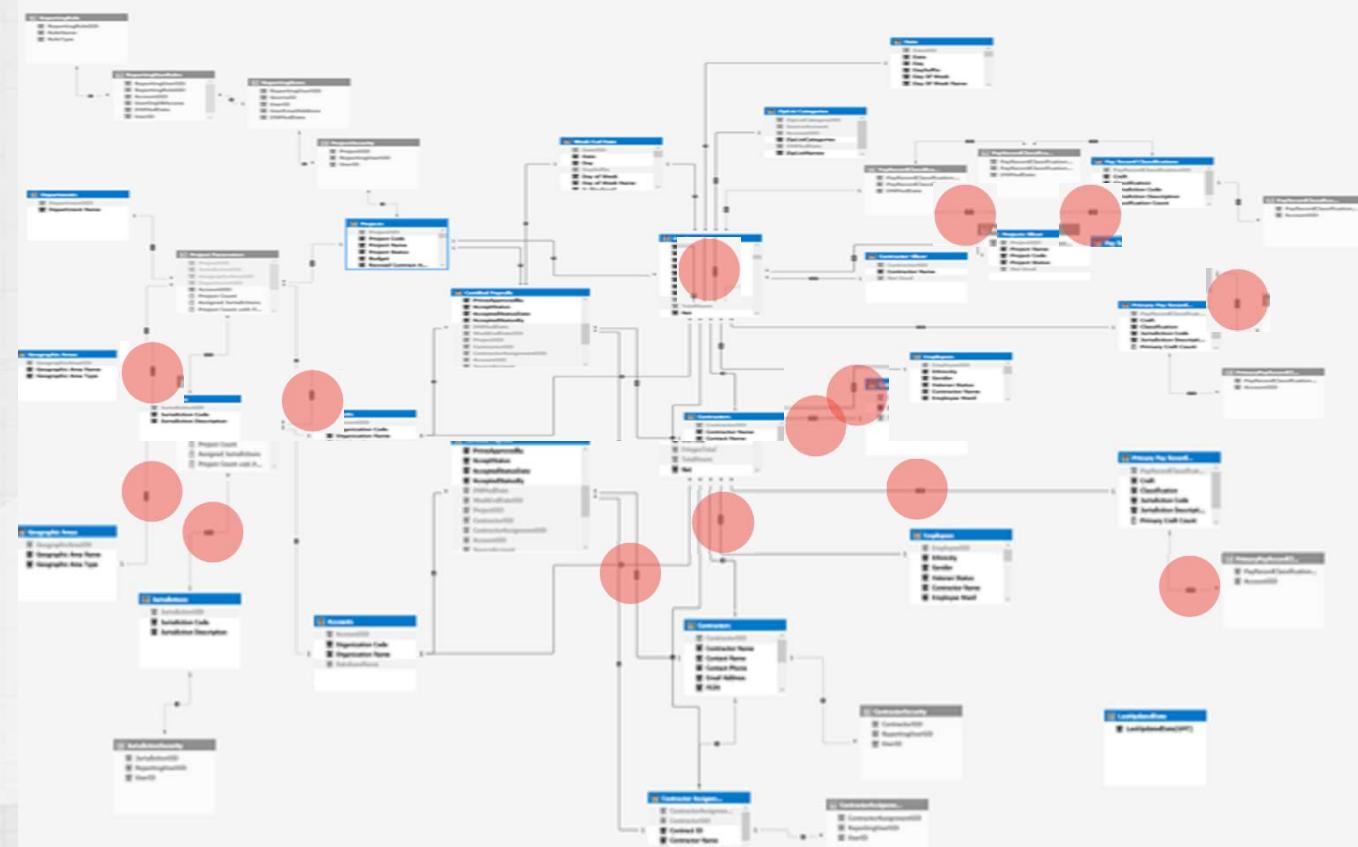
Things are more complex in a larger model

Few questions:

- Does Date filter Sales?
- Does Customer filter Purchases?
- Does Date filter Purchases?
- Which subset of sales is actually filtering Purchases?



What about a real model?



Never work with an ambiguous model

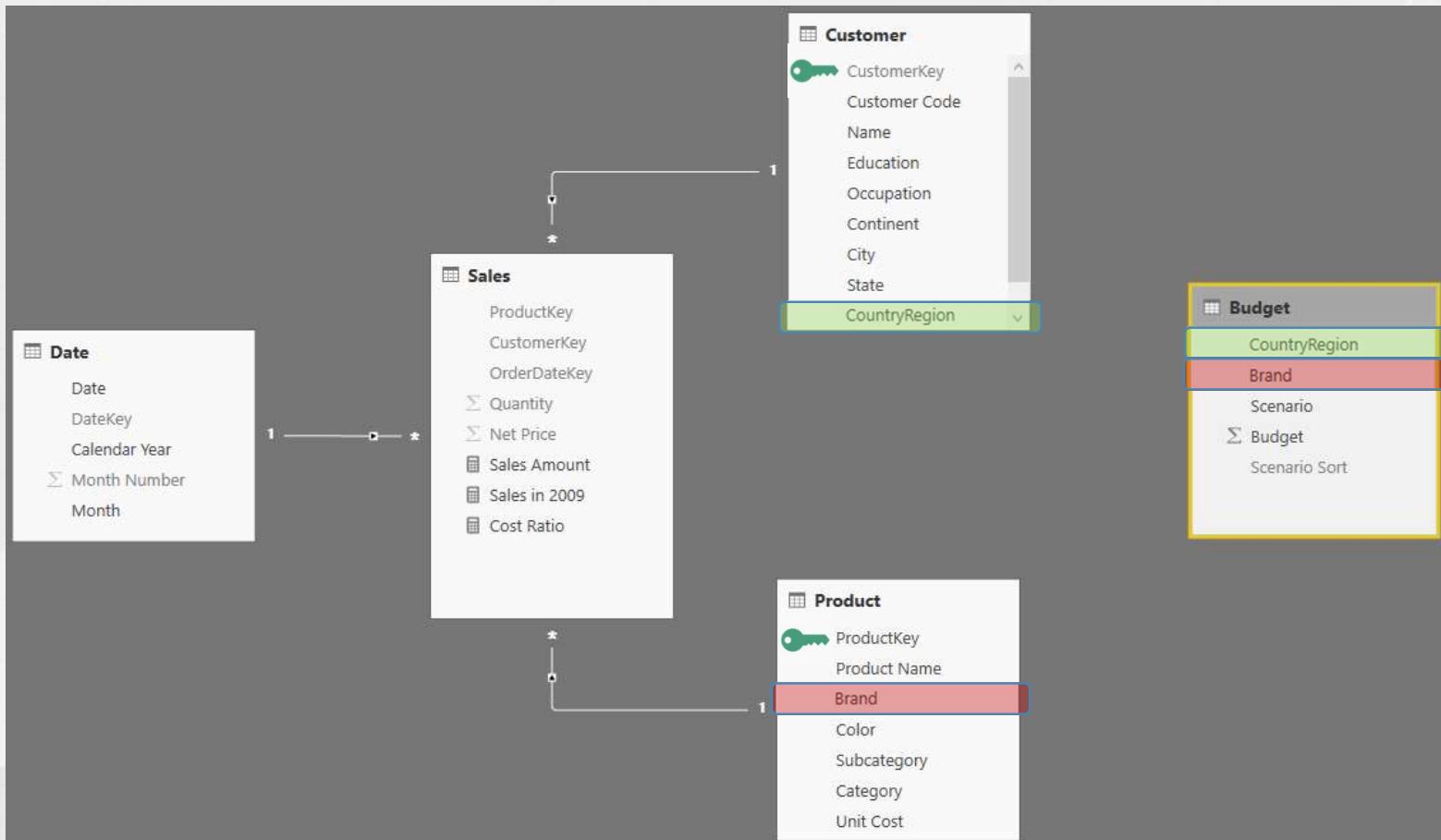
- Ambiguity is not always evident
 - The engine might consider as non-ambiguous a model that is ambiguous
 - The disambiguation rules have never been published
 - Adding a table might completely change the relationships setup
- Bidirectional cross-filter is the major culprit
 - Don't use it just "because it looks nicer to the user"
 - Wrong values are quite never nice

Relationships at different granularities are a challenge

Working at different granularity



Analyzing budget data



Missing relationship

- Without the relationship, the model does not work
- The relationship exists, but at a different granularity
- In fact, dimensions have granularity too
- Need to build a relationship at a different granularity

Brand	Sales in 2009	Budget
A. Datum	1,823,681.13	44,855,187.00
Adventure Works	4,878,941.52	44,855,187.00
Contoso	9,113,675.42	44,855,187.00
Fabrikam	7,933,936.37	44,855,187.00
Litware	4,668,613.86	44,855,187.00
Northwind Traders	826,993.38	44,855,187.00
Proseware	3,664,900.11	44,855,187.00
Southridge Video	1,892,420.79	44,855,187.00
Tailspin Toys	606,558.34	44,855,187.00
The Phone Company	1,891,590.92	44,855,187.00
Wide World Importers	3,317,561.02	44,855,187.00
Total	40,618,872.86	44,855,187.00

Problems to solve

- Budget is at the year level, needs to slice by month too
- Brand is not a key in Product
- CountryRegion is not a key in Customer
- We will see several solutions
 - Calculated tables to allocate the budget
 - DAX code to simulate relationships
 - Weak relationships
 - Creation of new tables to slice

Using TREATAS

TREATAS can change the data lineage of a column, transforming the data lineage of Product and Customer columns in Budget ones.

```
Budget 2009 :=
```

```
CALCULATE (
    SUM ( Budget[Budget] ),
    TREATAS (
        VALUES ( 'Product'[Brand] ),
        Budget[Brand]
    ),
    TREATAS (
        VALUES ( Customer[CountryRegion] ),
        Budget[CountryRegion]
    )
)
```

Use DAX to move the filters

You can use DAX to move the filter from the Product[Brand] column to the Budget[Brand] one, and repeat the same operation for the CountryRegion pair of columns.

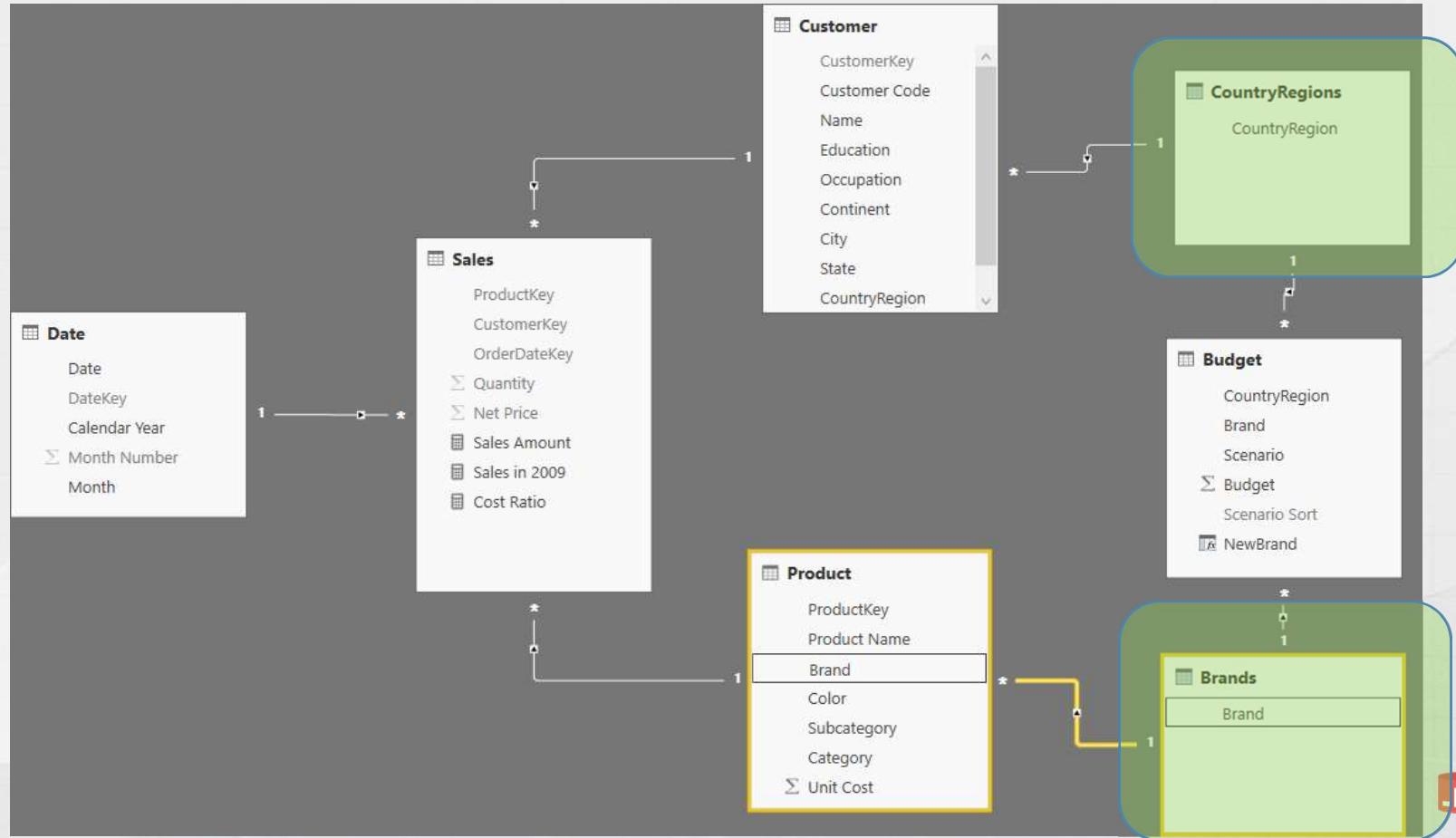
Budget 2009 :=

```
CALCULATE (
    SUM ( Budget[Budget] ),
    Budget[Brand] IN VALUES ( 'Product'[Brand] ),
    Budget[CountryRegion] IN VALUES ( Store[CountryRegion] )
)
```

Using DAX to move filter

- Flexibility
 - You change the filter context in a very dynamic way
 - Full control over the functions used
- Complexity
 - Every measure need to be authored using the pattern
 - Error-prone
- Speed
 - Using DAX to move a filter is sub-optimal
 - Leverages the slower part of the DAX engine (FE)

Calculated tables to slice dimensions



Using calculated tables

In Power BI and Analysis Services the intermediate tables can be built as calculated tables.

```
Brands =  
DISTINCT (   
    UNION (   
        DISTINCT ( Product[Brand] ),  
        DISTINCT ( Budget[Brand] )  
    )  
)  
  
CountryRegions =  
DISTINCT (   
    UNION (   
        DISTINCT ( Customer[CountryRegion] ),  
        DISTINCT ( Budget[CountryRegion] )  
    )  
)
```

Use ALLNOBLANKROW or
DISTINCT to avoid circular
dependency issues.

In fact, if you use ALL or VALUES,
the result depends from the
existence of the blank row.

Use the correct column to slice

Using Customer[CountryRegion]

CountryRegion	Sales in 2009	Budget
China	4,606,828.52	44,855,187.00
Germany	3,715,974.54	44,855,187.00
United States	32,296,069.79	44,855,187.00
Total	40,618,872.86	44,855,187.00

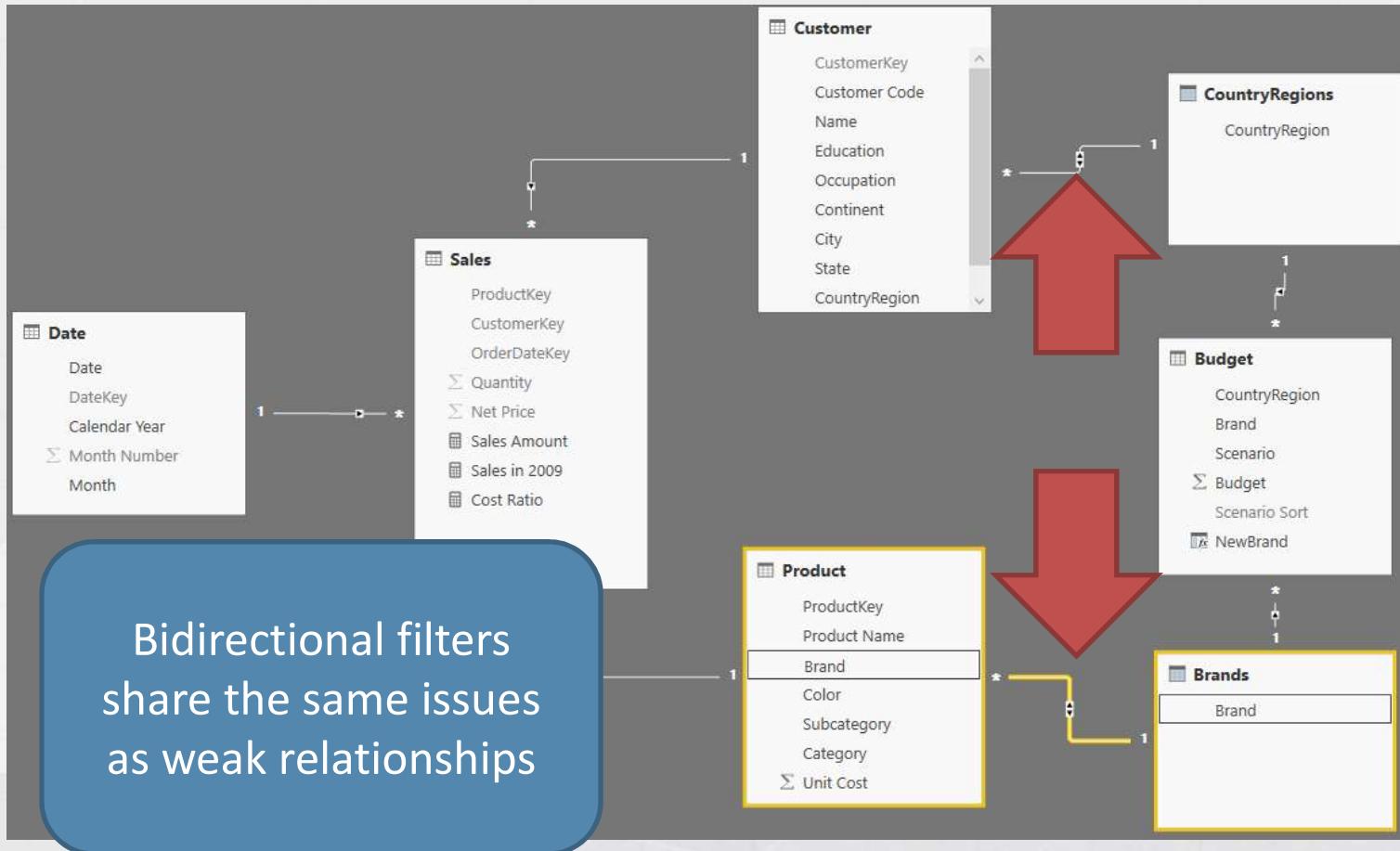
Using CountryRegions[CountryRegion]

CountryRegion	Sales in 2009	Budget
China	4,606,828.52	5,052,385.00
Germany	3,715,974.54	4,176,007.00
United States	32,296,069.79	35,626,795.00
Total	40,618,872.86	44,855,187.00

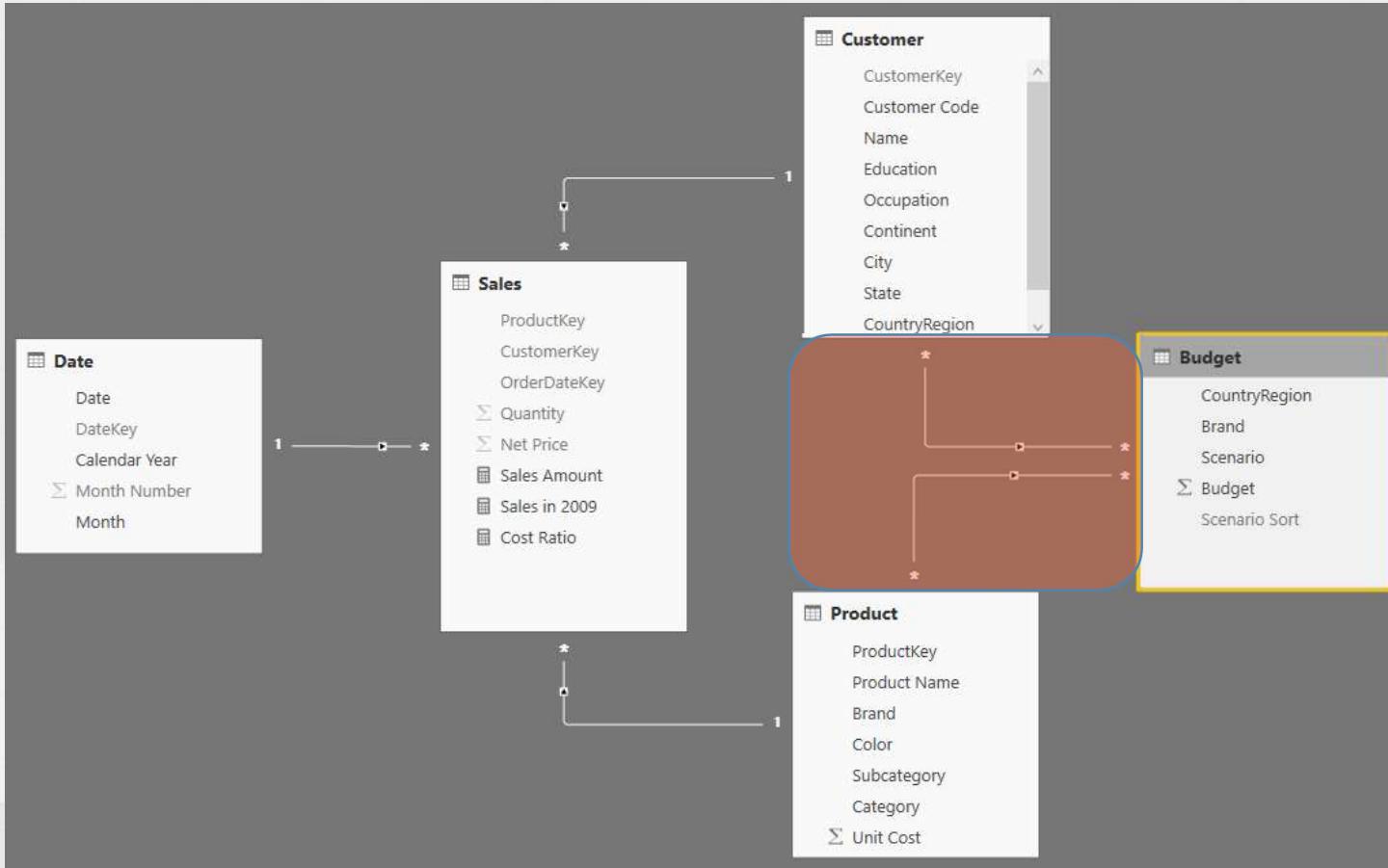
Having one table for each attribute might result in a complex data model.

The alternative is to work on ETL and make sure that all brands and country regions are present.

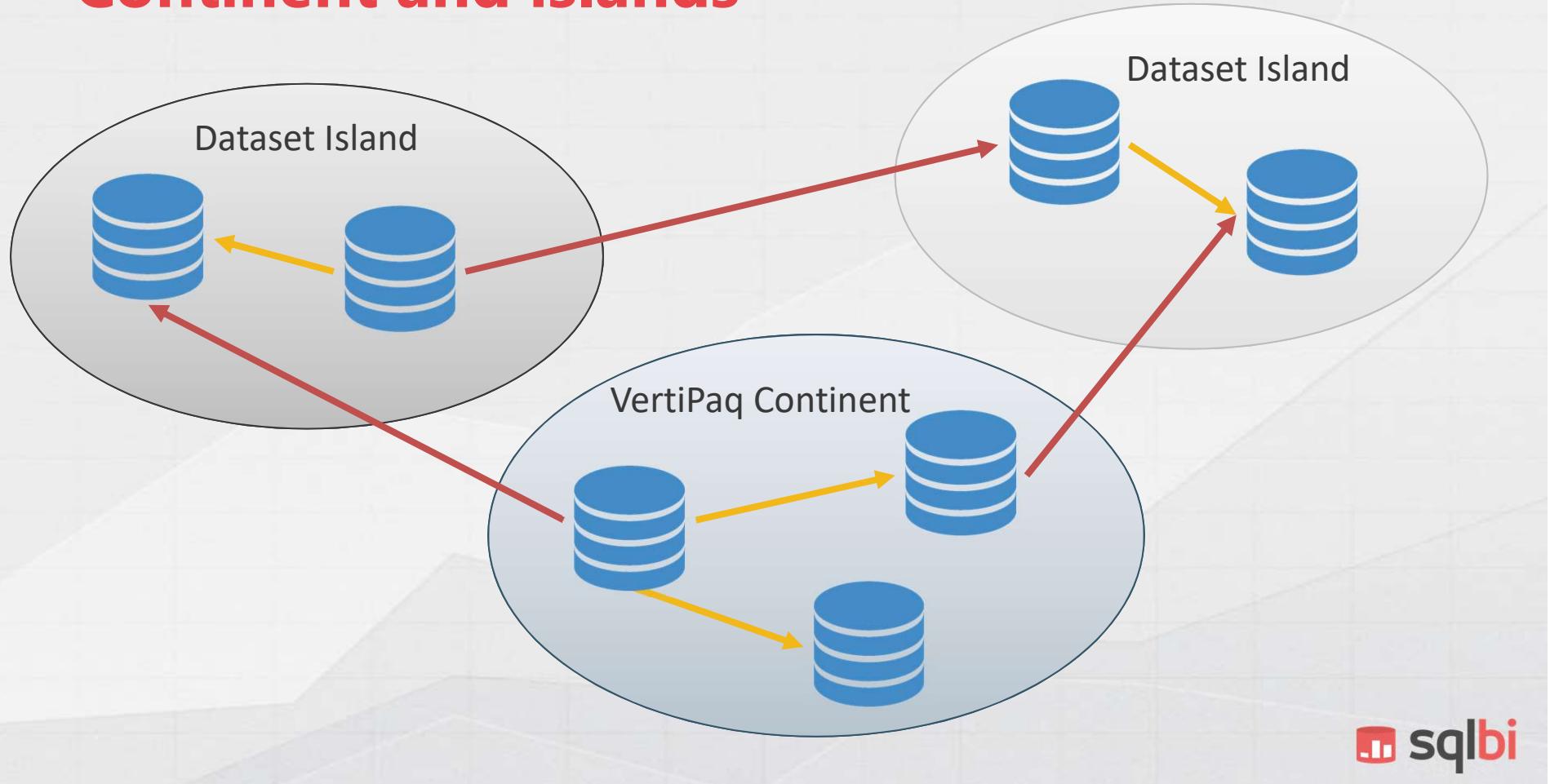
Leveraging bidirectional filtering



Leveraging weak relationships



Continent and islands



Weak relationships

- Both sides of a relationship can be the many side
 - They are not many-to-many relationships
 - They are relationships at different granularities
- Useful if the column is not a key in both tables
 - Otherwise, a regular strong 1:M relationship works well
- Mandatory, for cross-island relationships
- Need to choose the cross-filter direction
 - Bidirectional, A filters B, B filters A
- Table expansion does not happen, blank row is not enforced

Weak: no expansion, no blank row

Slice by Product[Brand]

Brand	Budget
Adventure Works	4,985,172.00
Contoso	7,127,903.00
Fabrikam	8,667,819.00
Litware	4,284,028.00
Northwind Traders	911,918.00
Proseware	3,192,659.00
Southridge Video	1,643,555.00
Tailspin Toys	600,524.00
The Phone Company	2,233,721.00
Wide World Importers	3,579,429.00
Total	39,004,512.00

Slice by Budget[Brand]

Brand	Budget
A. Datum	1,777,784.00
Adventure Works	4,985,172.00
Contoso	7,127,903.00
Fabrikam	8,667,819.00
Litware	4,284,028.00
Northwind Traders	911,918.00
Proseware	3,192,659.00
Southridge Video	1,643,555.00
Tailspin Toys	600,524.00
The Phone Company	2,233,721.00
Wide World Importers	3,579,429.00
Total	39,004,512.00

Scenario recap

- With DAX code the model is easy but slow
- Using relationships makes the model faster
- Using weak relationships the model becomes easier
 - Not available in all the versions of Tabular
- Calculated tables and bidirectional filters works the same
 - Available in Power BI and Analysis Services
 - Not available in Excel 2016 (unless you use CROSSFILTER)
- Still, we have granularity issues
 - As soon as you slice data at a different granularity
 - Different granularity leads to very complex results

Granularity is still a big issue

- Browsing at a different granularity produces wrong results
- Filtering still happens at the brand level, even though you are browsing by Color
- This is a very dangerous model, it shows wrong values without any warning

Color	Sales in 2009	Budget
Azure	191,923.74	2,044,452.00
Black	9,404,421.55	44,855,187.00
Blue	3,280,806.78	42,286,408.00
Brown	1,927,520.95	34,831,090.00
Gold	749,326.52	25,359,003.00
Green	1,629,013.20	34,663,373.00
Grey	3,994,161.17	43,806,481.00
Orange	1,278,078.15	29,252,507.00
Pink	1,048,841.42	35,450,683.00
Purple	12,717.19	18,979,372.00
Red	1,411,967.81	40,241,956.00
Silver	6,844,517.75	44,855,187.00
Silver Grey	297,279.51	20,209,533.00
Transparent	5,020.81	8,197,089.00
White	8,328,308.48	42,810,735.00
Yellow	214,967.83	20,869,460.00
Total	40,618,872.86	44,855,187.00

Checking granularity in the report

You can use DAX to understand at which granularity the user is browsing your data, so to detect when the granularity of the current context is different from the one of the budget table.

```
ProductsAtBudgetGranularity :=  
  
CALCULATE (  
    COUNTROWS ( Product ),  
    ALL ( Product ),  
    VALUES ( Product[Brand] )  
)  
  
ProductsAtSalesGranularity :=  
    COUNTROWS ( Product )
```

	Brand	Color	ProductsAtBudgetGranularity	ProductsAtSalesGranularity
A. Datum	Azure		132	14
	Black		132	18
	Blue		132	4
	Gold		132	4
	Green		132	14
	Grey		132	18
	Orange		132	18
	Pink		132	18
	Silver		132	18
	Silver Grey		132	6
Adventure Works	Total		132	132
	Black		192	54
	Blue		192	12
	Brown		192	15

Hiding the wrong granularity

Using a simple IF statement, you can clear out values which cannot be safely computed.

```
Budget 2010 :=  
IF (  
    AND (  
        [ProductsAtBudgetGranularity] = [ProductsAtSalesGranularity],  
        [CustomersAtBudgetGranularity] = [CustomersAtSalesGranularity]  
    ),  
    SUM ( Budget[Budget] )  
)
```



	Brand	Color	Sales in 2009	Budget Protected
A. Datum	Azure	191,923.74		
	Black	205,669.24		
	Blue	102,471.96		
	Gold	64,605.00		
	Green	162,939.48		
	Grey	241,288.79		
	Orange	223,352.14		
	Pink	254,624.79		
	Silver	264,136.40		
	Silver Grey	112,669.60		
Total		1,823,681.13		5,333,352.00
Adventure Works	Black	1,321,801.58		
	Blue	309,473.90		
	Brown	384,469.12		
	Grey	272,044.15		
	Red	266,074.36		

Hiding or reallocating?

- We hid the wrong values using DAX
- Can we do something more?
- A viable option is to define a reallocation factor
 - Using sales in 2009
 - Compute the percentage of the selection against the total
 - Use the percentage to dynamically reallocate the budget
- The code is slightly more complex
- Yet very dynamic and powerful

Allocating the budget

Brand	Color	Sales in 2009	Allocation Factor	Allocated Budget
A. Datum	Azure	191,923.74	10.52%	561,280.61
	Black	205,669.24	11.28%	601,479.29
	Blue	102,471.96	5.62%	299,679.05
	Gold	64,605.00	3.54%	188,937.20
	Green	162,939.48	8.93%	476,516.19
	Grey	241,288.79	13.23%	705,648.60
	Orange	223,352.14	12.25%	653,192.91
	Pink	254,624.79	13.96%	744,649.72
	Silver	264,136.40	14.48%	772,466.40
	Silver Grey	112,669.60	6.18%	329,502.03
	Total	1,823,681.13	100.00%	5,333,352.00
Adventure Works	Black	1,321,801.58	27.09%	4,051,744.28
	Blue	309,473.90	6.34%	948,636.40
	Brown	384,469.12	7.88%	1,178,520.72

Allocation factor: the first formula

You compute the sales at the correct granularity and then divide the budget by the ratio between sales and sales at budget granularity, using a technique similar to the one used to hide values.

Allocation Factor :=

```
DIVIDE (
    [Sales in 2009],
    CALCULATE(
        [Sales in 2009],
        ALL ( Customer ),
        VALUES ( Customer[CountryRegion] ),
        ALL ( 'Product' ),
        VALUES ( 'Product'[Brand] )
    )
)
```

Beware: this is NOT the same as ALLEXCEPT

Allocated Budget := SUM (Budget[Budget]) * [AllocationFactor]

Allocation factor: the correct formula

The final formula is more complex because the allocation is valid at Brand / CountryRegion granularity.

```
Allocated Budget :=  
  
SUMX (  
    KEEPFILTERS (   
        CROSSJOIN (   
            VALUES ( 'Product'[Brand] ),  
            VALUES ( Customer[CountryRegion] )  
        )  
    ),  
    [Allocation Factor] * CALCULATE ( SUM ( Budget[Budget] ) )  
)
```

Advanced Relationships



Time to write some DAX code by yourself.

Next exercise session is focuses on some simple and not-so-simple DAX code. Choose the exercise that best fit your skills.

Please refer to **lab number 10** on the hands-on manual.

Thank you!



Check our articles, whitepapers and courses on

www.sqlbi.com