

[Relationships in Power BI and Tabular models - SQLBI](#)

[Power BI documentation - Power BI | Microsoft Docs](#)

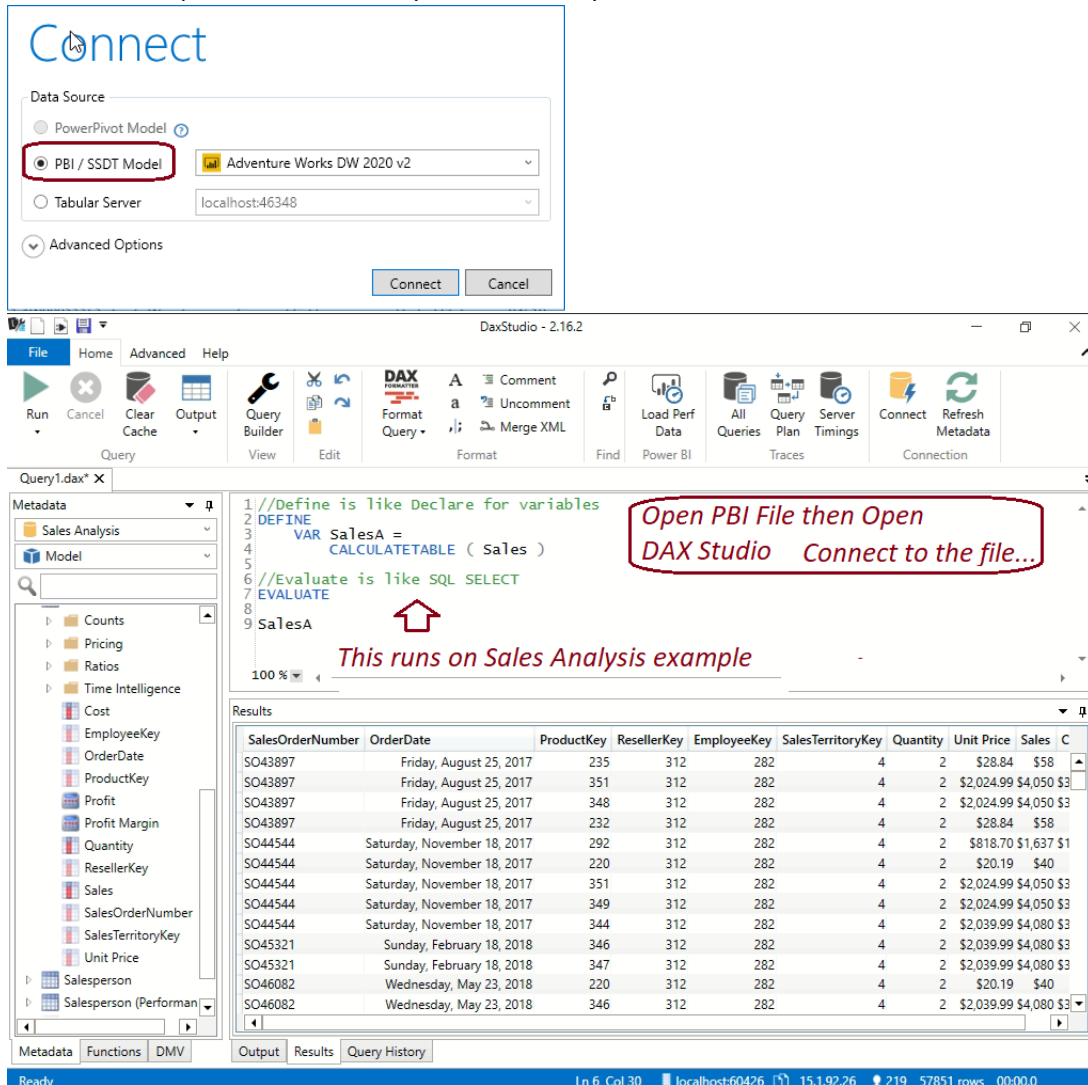
	Dimension table	Fact table
Model purpose	Stores business entities	Stores events or observations
Table structure	Includes a key column and descriptive columns for filtering and grouping	Includes dimension key columns and numeric measure columns that can be summarized
Data volume	Typically, contains fewer rows (relative to fact tables)	Can contain numerous rows
Query purpose	To filter and group	To summarize

DAX & M**Install DAX Studio**

DAX Studio: Connecting to Power BI Desktop

- If you have *installed DAX Studio* with the default All Users option, the installer will register DAX Studio with Power BI Desktop as an External Tool and you should *see a DAX Studio icon in the External Tools ribbon in Power BI Desktop*.
 - If you launch DAX Studio from there *it will open with a connection already established to the data model* in Power BI Desktop.
- Or if you launch DAX Studio while Power BI Desktop is running you can see a list of the open pbix files in the PBI / SSDT option and connect to your file that way.

3.



The screenshot shows the DAX Studio 2.16.2 interface. The 'Connect' dialog is open, with 'PBI / SSDT Model' selected. The main window displays a DAX query for 'Sales Analysis' and a table of results.

Connect Dialog:

- Data Source:
 - ☒ PBI / SSDT Model
 - ☐ PowerPivot Model
 - ☐ Tabular Server
- Adventure Works DW 2020 v2
- localhost:46348
- Advanced Options
- Connect
- Cancel

DAX Query:

```

1 //Define is like Declare for variables
2 DEFINE
3     VAR SalesA =
4         CALCULATETABLE ( Sales )
5
6 //Evaluate is like SQL SELECT
7 EVALUATE
8     SalesA
9

```

Results Table:

SalesOrderNumber	OrderDate	ProductKey	ResellerKey	EmployeeKey	SalesTerritoryKey	Quantity	Unit Price	Sales C
SO43897	Friday, August 25, 2017	235	312	282	4	2	\$28.84	\$58
SO43897	Friday, August 25, 2017	351	312	282	4	2	\$2,024.99	\$4,050 \$3
SO43897	Friday, August 25, 2017	348	312	282	4	2	\$2,024.99	\$4,050 \$3
SO43897	Friday, August 25, 2017	232	312	282	4	2	\$28.84	\$58
SO44544	Saturday, November 18, 2017	292	312	282	4	2	\$818.70	\$1,637 \$1
SO44544	Saturday, November 18, 2017	220	312	282	4	2	\$20.19	\$40
SO44544	Saturday, November 18, 2017	351	312	282	4	2	\$2,024.99	\$4,050 \$3
SO44544	Saturday, November 18, 2017	349	312	282	4	2	\$2,024.99	\$4,050 \$3
SO44544	Saturday, November 18, 2017	344	312	282	4	2	\$2,039.99	\$4,080 \$3
SO45321	Sunday, February 18, 2018	346	312	282	4	2	\$2,039.99	\$4,080 \$3
SO45321	Sunday, February 18, 2018	347	312	282	4	2	\$2,039.99	\$4,080 \$3
SO46082	Wednesday, May 23, 2018	220	312	282	4	2	\$20.19	\$40
SO46082	Wednesday, May 23, 2018	346	312	282	4	2	\$2,039.99	\$4,080 \$3

Annotations:

- Open PBI File then Open DAX Studio Connect to the file...
- This runs on Sales Analysis example

- Ready
- Relationships in Power BI and Tabular models - SQLBI
- <https://daxstudio.org/tutorials/getting-connected/#pbidesktop>

DAX is Data Analysis eXpression Language

This is the *common language between* SQL Server Analysis Services Tabular, Power BI, and Power Pivot in Excel. DAX is an expression language, and unlike M, it is very similar to Excel functions.

1. Use DAX in SSMS – good reads but need to setup SSAS CUBE to use in SSMS
 - a. [DAX For SQL Folks: Part I- Intro to DAX, Power BI and Data Viz – SQLServerCentral](#)
 - b. [DAX for SQL Folks: Part II - Translating SQL Queries to DAX Queries – SQLServerCentral](#)
 - c. [DAX #2 – Installing AdventureWorks DW Tabular Model SQL Server 2012 – SQLServerCentral](#)
 - d. [Release AdventureWorks for Analysis Services · microsoft/sql-server-samples \(github.com\)](#)
 - e. <https://docs.microsoft.com/en-us/analysis-services/multidimensional-tutorial/install-sample-data-and-project>
 - f. [Analysis Services Adventure Works Internet Sales tutorial \(1500\) | Microsoft Docs](#)
 - g. [Tabular modeling overview - Analysis Services | Microsoft Docs](#)

2. [Data Analysis Expressions \(DAX\) Reference - DAX | Microsoft Docs](#) – Entry point
3. [Use DAX in Power BI Desktop - Learn | Microsoft Docs](#) – MS Learning Path
4. [DAX overview - DAX | Microsoft Docs](#)

```
Sales Rolling 12 Months =
CALCULATE(
    SUM(FactInternetSales[SalesAmount]),
    DATESBETWEEN(
        DimDate[FullDateAlternateKey],
        NEXTDAY(SAMEPERIODLASTYEAR(LASTDATE(DimDate[FullDateAlternateKey]))),
        LASTDATE(DimDate[FullDateAlternateKey])
    ),
    ALL(DimDate)
)
```

- 5.
6. Parallels between a relational database and cube from the perspective of the two languages.

Relational Database	Cube
Table	Dimension
Column	Attribute
Rows	Members

- 7.
8. space

DAX is the analytical engine in Power BI. It is best language to answer analytical questions which their responses will be different based on the selection criteria in the report.

Simple Report Examples: *What is the question/what is the math/what visual says it best?*

1. Calculate Rolling 12 Months Average of Sales.
2. Analyze growth percentage across product categories and for different date ranges.
3. Calculate year-over-year growth compared to market trends?
4. You an Analyst at Adventureworks, you get a request call to provide the Profit Margin of products sold by Resellers.

- a. You pretend you know what is being asked of you, and then you google Profit Margin to come up with the formulas below.

$$\text{Profit Margin} = \frac{\text{Profit}}{\text{Product Sales}}$$

- b. $\text{Profit} = \text{Products Sales} - \text{Cost Of Products Sold}$

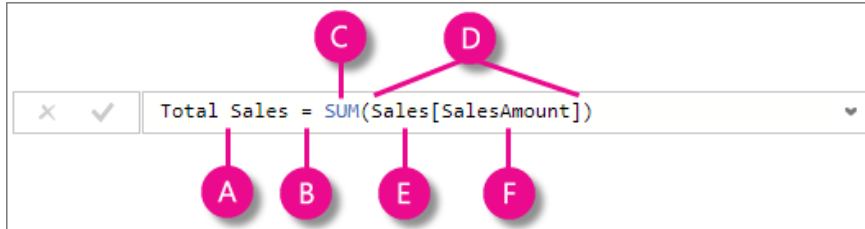
5. space

It is really hard if you want to calculated that in M, because you have to consider all different types of possibilities; Rolling 12 months for each product, for every customer, for every combination and etc.

However, if you use a DAX calculation for it, the analytical engine of DAX takes care of all different combinations selected through *Filter Context in the report*.

[DAX basics in Power BI Desktop - Power BI | Microsoft Docs](#) – reference page

Syntax: A simple DAX formula for a measure: *Use a measure to re-use it: OOP's Style*



This formula includes the following syntax elements:

- A. *Measure name*, **Total Sales**.
- B. Equals sign operator (=) *indicates the beginning of the formula; it returns a result*.
- C. The *DAX function* **SUM**, which adds up all of the numbers in the **Sales[SalesAmount]** column.
- D. Parenthesis (), wrap an expression that contains one or more *arguments*. Most functions require at least one argument. An argument *passes a value to a function*.
- E. The *referenced table*, **Sales**.
- F. The *referenced column*, **[SalesAmount]**, in the Sales table. The column is the argument on which to aggregate a SUM.

Read this formula as: For the measure named Total Sales, calculate (=) the SUM of values in the [SalesAmount] column in the Sales table.

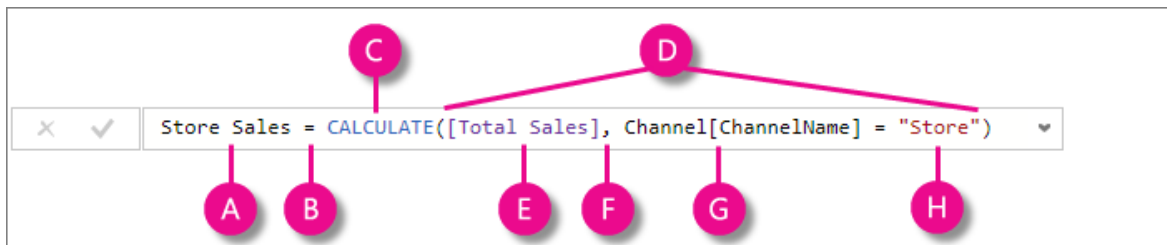
When added to a report, this measure calculates and returns values by *summing up sales amounts for each of the other fields we include*, for example, Cell Phones in the USA.

You might be thinking, "Isn't this measure doing the same thing as if I were to just add the SalesAmount field to my report?" Well, yes. But, there's a *good reason to create our own measure* that sums up values from the SalesAmount field: We can *use it as an argument in other formulas*. *Reuse*

Filter context

Can most easily be applied *by adding fields to a visualization*, filter context can also be applied in a DAX formula by *defining a filter using functions* such as ALL, RELATED, FILTER, CALCULATE, by relationships, and by other measures and columns.

A measure named Store Sales:



- A. The *measure name*, **Store Sales**.
- B. The equals sign operator (=), which indicates the beginning of the formula.
- C. The *CALCULATE function*, which evaluates an expression, as an *argument*, in a context that is modified by the specified filters.
- D. Parenthesis (), wrap *arguments* – separated by a comma.
- E. A *measure [Total Sales] in the same table as an expression* with the formula: =SUM(Sales[SalesAmount]).
- F. A comma (,), which *separates the first expression argument from the filter argument*.

- G. The *fully qualified referenced column*, **Channel[ChannelName]**. (Table[Column])
 - a. The row context.
 - b. Each *row in this column specifies a channel*, such as Store or Online.
- H. The particular value, **Store**, as a *filter context*.

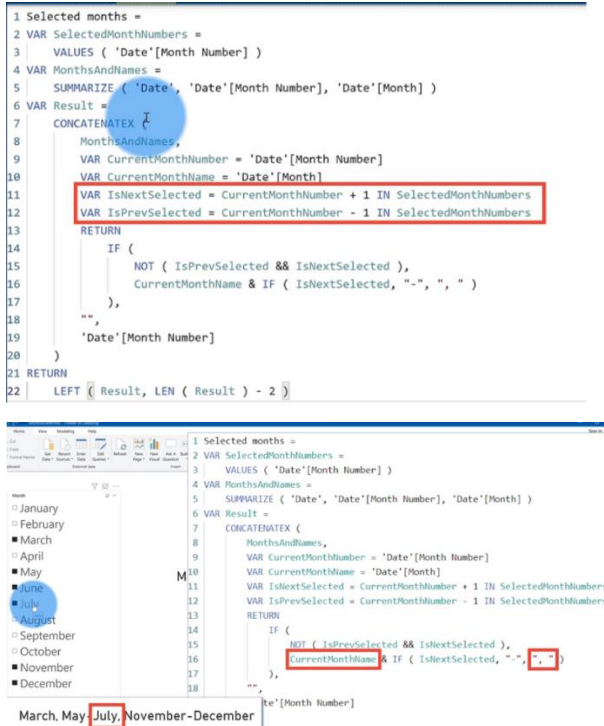
This formula ensures only sales values defined by the Total Sales measure are calculated only for rows in the Channel[ChannelName] column, with the value Store used as a filter.

- Functions: <https://docs.microsoft.com/en-us/power-bi/transform-model/desktop-quickstart-learn-dax-basics#functions>
- Context: [DAX basics in Power BI Desktop - Power BI | Microsoft Docs](#)
- [Relationships in Power BI and Tabular models - SQLBI](#)

```

1 Selected months =
2 VAR SelectedMonthNumbers =
3   VALUES ( 'Date'[Month Number] )
4 VAR MonthsAndNames =
5   SUMMARIZE ( 'Date', 'Date'[Month Number], 'Date'[Month] )
6 VAR Result =
7   CONCATENATEX (
8     MonthsAndNames,
9     VAR CurrentMonthNumber = 'Date'[Month Number]
10    VAR CurrentMonthName = 'Date'[Month]
11    VAR IsNextSelected = CurrentMonthNumber + 1 IN SelectedMonthNumbers
12    VAR IsPrevSelected = CurrentMonthNumber - 1 IN SelectedMonthNumbers
13    RETURN
14    IF (
15      NOT ( IsPrevSelected && IsNextSelected ),
16      CurrentMonthName & IF ( IsNextSelected, "-", ", " )
17    ),
18    ""
19    , 'Date'[Month Number]
20  )
21 RETURN
22 LEFT ( Result, LEN ( Result ) - 2 )

```

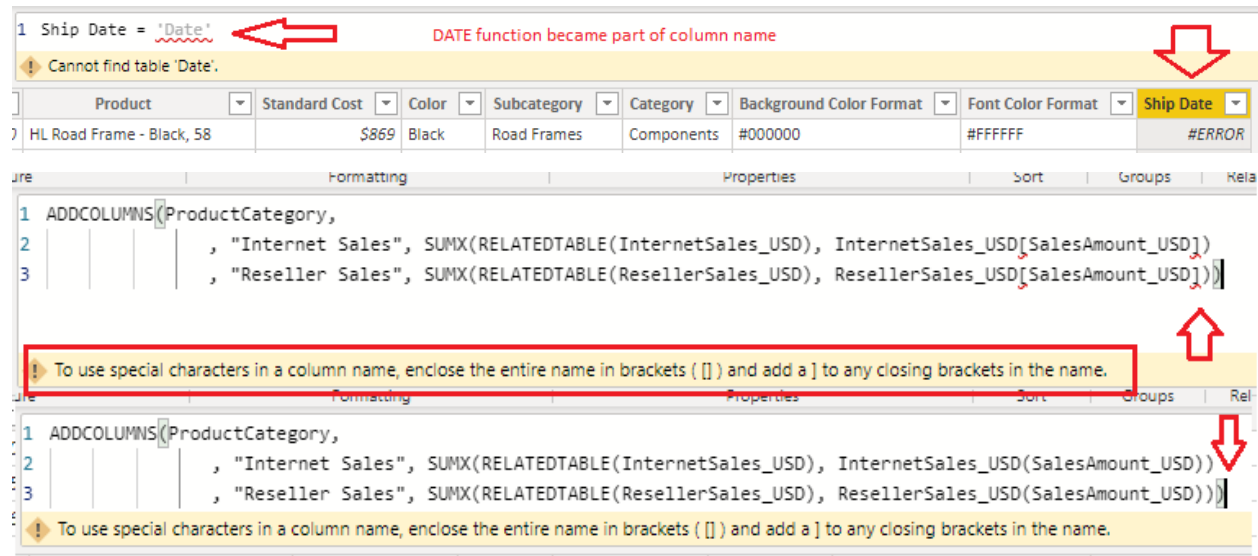


```

1 Selected months =
2 VAR SelectedMonthNumbers =
3   VALUES ( 'Date'[Month Number] )
4 VAR MonthsAndNames =
5   SUMMARIZE ( 'Date', 'Date'[Month Number], 'Date'[Month] )
6 VAR Result =
7   CONCATENATEX (
8     MonthsAndNames,
9     VAR CurrentMonthNumber = 'Date'[Month Number]
10    VAR CurrentMonthName = 'Date'[Month]
11    VAR IsNextSelected = CurrentMonthNumber + 1 IN SelectedMonthNumbers
12    VAR IsPrevSelected = CurrentMonthNumber - 1 IN SelectedMonthNumbers
13    RETURN
14    IF (
15      NOT ( IsPrevSelected && IsNextSelected ),
16      CurrentMonthName & IF ( IsNextSelected, "-", ", " )
17    ),
18    ""
19    , 'Date'[Month Number]
20  )
21 RETURN
22 LEFT ( Result, LEN ( Result ) - 2 )

```

Syntax issues



1 Ship Date = 'Date' DATE function became part of column name

Cannot find table 'Date'.

Product	Standard Cost	Color	Subcategory	Category	Background Color Format	Font Color Format	Ship Date
HL Road Frame - Black, 58	\$869	Black	Road Frames	Components	#000000	#FFFFFF	#ERROR

```

1 ADDCOLUMNS(ProductCategory,
2   , "Internet Sales", SUMX(RELATEDTABLE(InternetSales_USD), InternetSales_USD(SalesAmount_USD))
3   , "Reseller Sales", SUMX(RELATEDTABLE(ResellerSales_USD), ResellerSales_USD(SalesAmount_USD)))
  
```

To use special characters in a column name, enclose the entire name in brackets ([]) and add a] to any closing brackets in the name.

```

1 ADDCOLUMNS(ProductCategory,
2   , "Internet Sales", SUMX(RELATEDTABLE(InternetSales_USD), InternetSales_USD(SalesAmount_USD))
3   , "Reseller Sales", SUMX(RELATEDTABLE(ResellerSales_USD), ResellerSales_USD(SalesAmount_USD)))
  
```

To use special characters in a column name, enclose the entire name in brackets ([]) and add a] to any closing brackets in the name.

Using the empties as SARG?

The **BLANK** data type deserves a special mention. DAX uses BLANK for both database **NULL** and for blank cells in Excel. BLANK *doesn't mean zero*. Perhaps it might be simpler to think of it as the "*absence of a value*".

Two DAX functions are related to the BLANK data type: the **BLANK** DAX function returns BLANK, while the **ISBLANK** DAX function tests whether an expression evaluates to BLANK.

[DAX function reference - DAX | Microsoft Docs](#)

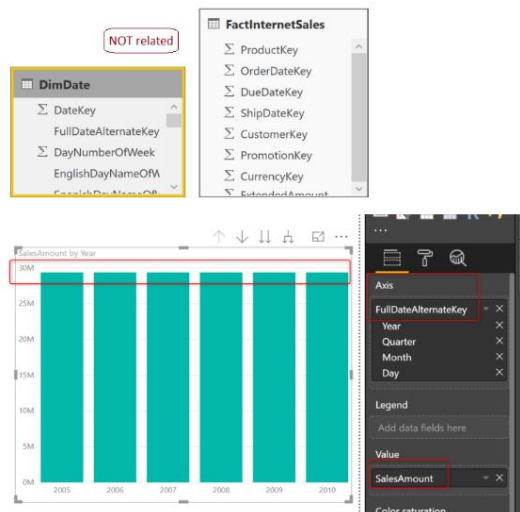
All comparison operators, except strict equal to (==), treat BLANK as equal to the number zero, an empty string (""), the date December 30, 1899, or FALSE. It means that the expression [Revenue] = 0 will be TRUE when the value of [Revenue] is either zero or BLANK. In contrast, [Revenue] == 0 is TRUE only when the value of [Revenue] is zero

Comment Code

Select multiple lines of code you want to comment 2 Press Ctrl+KC to comment those lines 3 Press Ctrl+KU to uncomment commented lines.

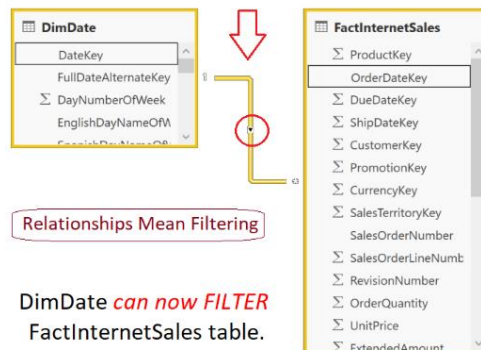
[DAX Syntax Highlighting for Notepad++ – Sascha D. Kasper \(sascha-kasper.com\)](#)

Relationships mean Filtering:

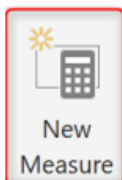
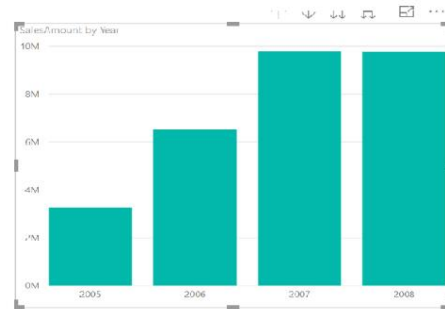


- i. Chart is *showing the same SalesAmount for every single year* from 2005 to 2010.
- ii. The *value is the grand total of the sales* in my dataset.
- iii. FullDateAlternateKey field is *NOT filtering* the FactSalesAmount table.

Filter direction Arrow points from the One side to the Many



DimDate *can now FILTER* FactInternetSales table.



```

1 Sales by Ship Date = CALCULATE(
2     SUM(FactInternetSales[SalesAmount]),
3     USERELATIONSHIP(
4         FactInternetSales[ShipDateKey],
5         DimDate[DateKey]
6     )
7 )

```

```

1 AppleSales :=
2 CALCULATE (
3     SUM ( Sales[Amount] ),
4     Product[Product] = "Apple"
5 )

```

Filter Propagation: The filter applied on Product[Product] follows the relationship between Product and Sales filtering Sales table

table expansion happens when you define a table.

```

1  DEFINE
2      VAR SalesA =
3          CALCULATETABLE ( Sales, USERELATIONSHIP ( Sales[Date], 'Date'[Date] ) )
4      VAR SalesB = contains the expanded Sales table.
5          CALCULATETABLE ( Sales, USERELATIONSHIP ( Sales[DueDate], 'Date'[Date] ) )
6  EVALUATE
7      ADDCOLUMNS ( SalesB, "Month", RELATED ( 'Date'[Month] ) )

```

RELATED accesses the related columns of the expanded version of Sales.

The two variables store the Sales table using two different relationships. SalesA uses the default relationship, whereas SalesB uses the relationship with Sales[DueDate] instead of Sales[Date]. The last ADDCOLUMNS iterates SalesB and returns the RELATED Date[Month]. What will the result be? The month of the Sales[Date] column or the month of the Sales[DueDate] column?

when ADDCOLUMNS is executed, the active relationship is the relationship using Sales[Date] and you would think that the month is the month of that date. Right? Wrong!

The correct reasoning is as follows:

SalesB contains the expanded Sales table, and that expansion happened when the active relationship was the relationship with Sales[DueDate]. As a result, Date[Month] contained in SalesB is related to Sales[DueDate], not to Sales[Date].

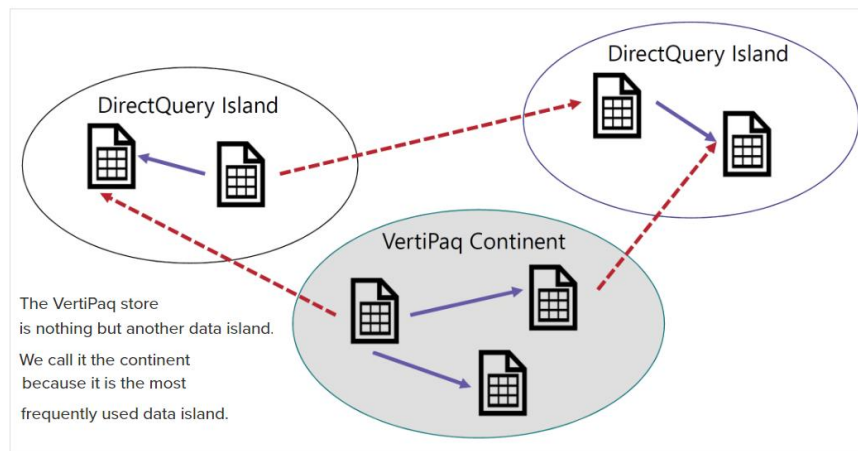
Writing Complex Formulas

1. [How To Solve a Complex DAX Problem - Excelerator BI](#)
 - a. If you want to learn how to write a formula
2. [Complex Power BI Reports using DAX Functions | CloudFronts](#)
3. Calculate and compare Sales Figure by Day
4. Calculate and compare Sales Figure by Date
5. Calculate and compare Sales Running Total
- 6.

Relationships & Filter Direction & Expanded tables in DAX - SQLBI

- The purpose of a *relationship* in a Tabular model is to *transfer a filter* while querying the model.
 - If both tables belong to the same island, then the relationship is an intra-island relationship.
 - If the two tables belong to different islands, then it is a cross-island relationship.
- Cross-island relationships are *always weak relationships*.
 - Table expansion* never crosses islands.
- A *strong* relationship the engine knows that the one-side of the relationship contains unique values.
- If the engine cannot check that the one-side of the relationship contains unique values for the key, the relationship is *weak*.
 - A weak relationship *is not used as part of table expansion*.
- A single data model can contain tables stored in VertiPaq and tables stored in DirectQuery.
- Tables in DirectQuery can originate from different data sources generating several DirectQuery data islands.

In order to differentiate between data in VertiPaq and data in DirectQuery, we say that data is either in the *continent* (VertiPaq) or in the *islands* (DirectQuery data sources).



- space

3 types of relationship cardinality available:

- One-to-many relationships:** This is the *most common type* of relationship cardinality.
 - On the one-side of the relationship the column must have unique values;
 - on the many-side the value can (and usually does) contain duplicates.
 - Some client tools differentiate between one-to-many relationships and many-to-one relationships. Still, *they are the same type of relationship*. It all *depends on the order of the tables*:
 - A one-to-many relationship between *Product* and *Sales* is the same as a many-to-one relationship between *Sales* and *Product*.
- One-to-one relationships:** This is a rather uncommon type of relationship cardinality. *On both sides of the relationship the columns need to have unique values*. A more accurate name would be “zero-or-one”-to-“zero-or-one” relationship because the presence of a row in one table does not imply the presence of a corresponding row in the other table.
- Many-to-many relationships:** *On both sides of the relationship the columns can have duplicates*. This feature was introduced in 2018, and unfortunately its name is somewhat confusing. Important to understand that many-to-many *does not* refer to the many-to-many relationship, *but to the many-to-many cardinality* of the relationship.

In order to avoid ambiguity, use acronyms to describe the cardinality of a relationship:

- One-to-many relationships: We call them **SMR**, which stands for Single-Many-Relationship.
- One-to-one relationships: We use the acronym **SSR**, which stands for Single-Single-Relationship.
- Many-to-many relationships: We call them **MMR**, which stands for Many-Many-Relationship.

An **MMR relationship is always weak**, regardless of whether the two tables belong to the same island or not. If the developer defines both sides of the relationship as the many-side, then the relationship is automatically treated as a weak relationship with no table expansion taking place.

Each relationship has a cross-filter direction. The cross-filter direction is the direction used to propagate its effect. The cross-filter can be set to one of two values:

- **Single:** The filter context is always propagated **in one direction** of the relationship and not the other way around. In a **one-to-many** relationship, the direction **is always from the one-side** of the relationship to the many-side. This is the standard and most desirable behavior.
- **Both:** The filter context is propagated in **both directions** of the relationship. This is also called a **bidirectional cross-filter** and sometimes just a bidirectional relationship. In a one-to-many relationship, the filter context still retains its feature of propagating from the one-side to the many-side, but it also propagates from the many-side to the one-side.

The cross-filter directions available depend on the type of relationship.

- In an **SMR** relationship one can **choose** single or bidirectional.
- An **SSR** relationship **always uses** bidirectional filtering. There is no many-side.
- In an **MMR** relationship both sides are the many-side. Both sides can be the source and the target of a filter context propagation. Can **choose** the cross-filter direction to be **bidirectional**, in which case the propagation always goes both ways. If the developer chooses **single propagation**, they also must **choose which table to start the filter propagation from**. As with all other relationships, single propagation is the best practice.

The following table summarizes the different types of relationships with the available cross-filter directions, their effect on filter context propagation, and the options for weak/strong relationship.

Type of Relationship	Cross-filter Direction	Filter Context Propagation	Weak & Strong Type
SMR	Single	From one-side to many-side	Weak if cross-island, strong otherwise
SMR	Both	Bidirectional	Weak if cross-island, strong otherwise
SSR	Both	Bidirectional	Weak if cross-island, strong otherwise
MMR	Single	Must choose the source table	Always weak
MMR	Both	Bidirectional	Always weak

When two tables are linked through a strong relationship, the table on the one-side might contain the additional blank row in case the relationship is invalid –**VertiPaq Analyzer** reports a Referential Integrity violation when this happens.

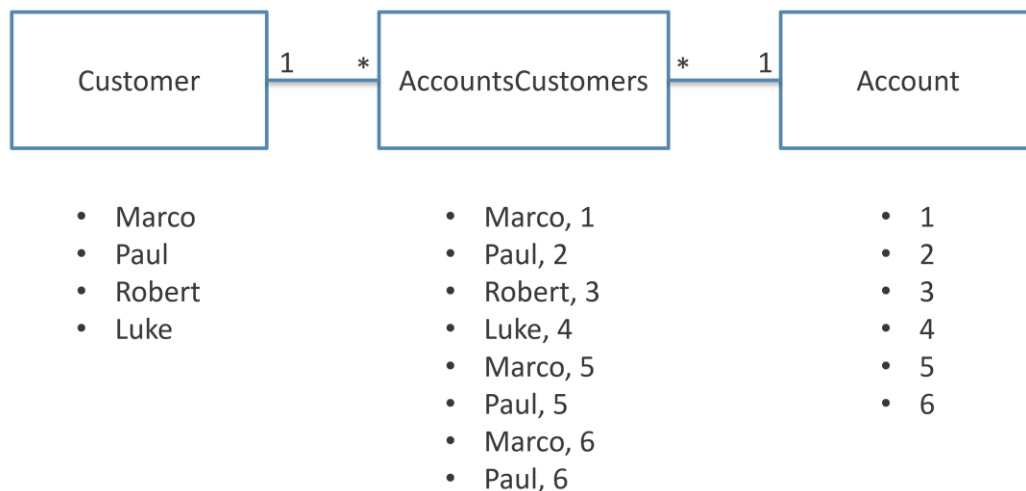
If the many-side of a strong relationship contains values not present in the table on the one-side, then a blank row is appended to the table on the one-side. The additional blank row is never added to tables involved in a weak relationship.

A physical relationship can automatically propagate a filter in the filter context based on the filter propagation direction. Moreover, *a relationship defines a unique constraint on columns that are on the one-side of a relationship.*

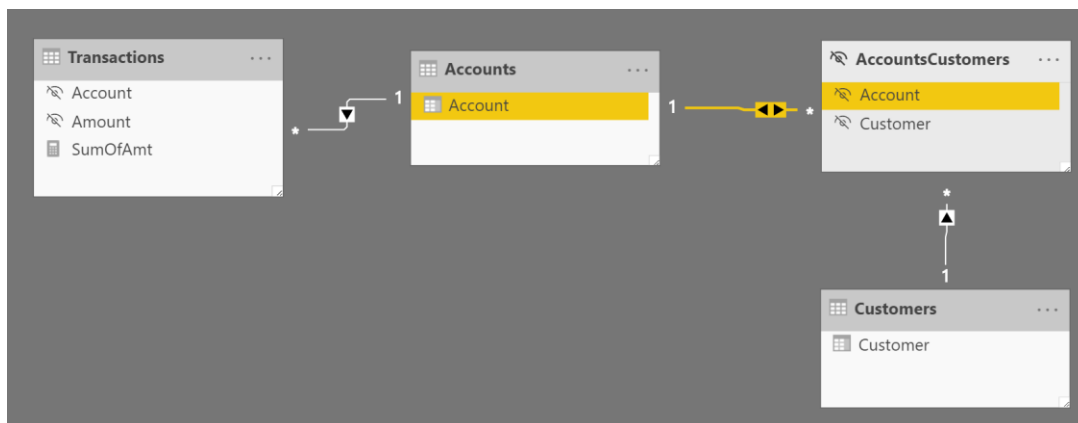
Many-to-many relationships between business entities aka dimensions

A logical many-to-many relationship between two tables representing two different business entities –also known as dimension tables in dimensional modeling – cannot be implemented using a single physical relationship.

For example, consider the case of two tables, *Customer* and *Account* in a bank. Every customer can have multiple accounts, and every account can be owned by several customers. The relationship between the *Customer* and *Account* tables requires a third table, *AccountsCustomers*, which contains one row for each existing relationship between *Account* and *Customer*.



The logical many-to-many relationship between *Customer* and *Account* is implemented through two physical SMR relationships. In order *to enable the propagation of the filter context from Customer to Account*, a bidirectional filter must be activated in the data model on the relationship between *Customer* and *AccountsCustomers*.



However, in order to *avoid ambiguity caused by bidirectional filters* in the data model, it is considered best practice to *enable the bidirectional filter only in DAX measures that require that filter propagation.*

Example, if the relationship between *Customer* and *AccountsCustomers* is defined with a single direction filter, the bidirectional filter can be activated using **CROSSFILTER** as in the following measure:

```

1      SumOfAmt :=
2      CALCULATE (
3          SUM ( Transactions[Amount] ),
4          CROSSFILTER (
5              AccountsCustomers[AccountKey],
6              Accounts[AccountKey],
7              BOTH
8          )
9      )

```

It is important to note that a many-to-many relationship between business entities does not use any MMR relationship, which is required to solve another modeling issue related to different granularities.

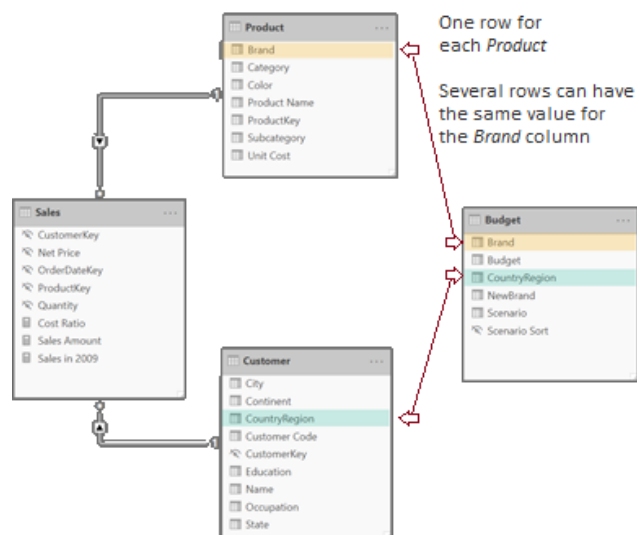
Relationships at different granularities

A logical relationship can exist between two tables whose granularity is not compatible with a physical SMR relationship.

Example, consider the following model where *the granularity of the Budget table is defined at the Customer[CountryRegion] and Product[Brand] level*.

The *Customer* table has one row for each customer, so there could be multiple rows with the same value for the *CountryRegion* column. Similarly, there is one row for each *Product* and several rows can have the same value for the *Brand* column.

Create a report that aggregates all the rows in *Budget*



The cardinality of *Product* and *Customer* is the right one for the *Sales* table.

In order to **create a report that aggregates all the rows in *Budget*** for a given product brand or customer country, one option is to apply two virtual relationships in a measure:

```

1      Budget Amount :=
2      CALCULATE (
3          SUM ( Budget[Budget] ),
4          TREATAS (
5              VALUES ( 'Product'[Brand] ),
6              Budget[Brand]
7          ),
8          TREATAS (
9              VALUES ( Customer[CountryRegion] ),
10             Budget[CountryRegion]
11         )
12     )

```

Any technique that transfers the filter using a virtual relationship implemented through a DAX expression might suffer a performance penalty compared to a solution based on physical relationships.

In order to use the more efficient SMR relationships,

Create a *CountryRegions* calculated table that contains all the values existing in either *Customer[CountryRegion]* or *Budget[CountryRegion]*,

and a *Brands* calculated table containing all the values existing in either *Product[Brand]* or *Budget[Brand]*:

```

1      Countries =
2      DISTINCT (
3          UNION (
4              DISTINCT ( Customer[CountryRegion] ),
5              DISTINCT ( Budget[CountryRegion] )
6          )
7      )
8
9      Brands =
10     DISTINCT (
11         UNION (
12             DISTINCT ( Product[Brand] ),
13             DISTINCT ( Budget[Brand] )
14         )
15     )

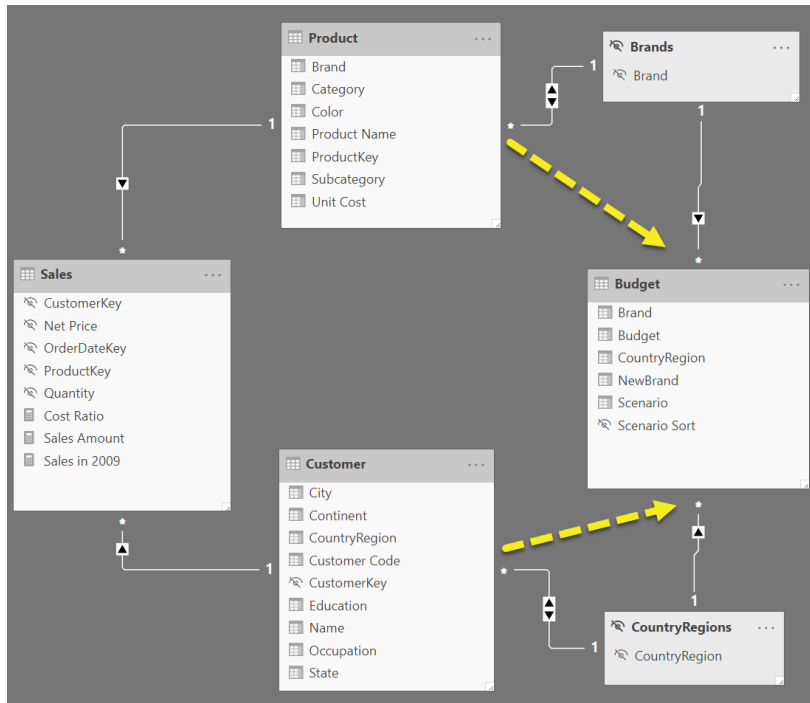
```

The *CountryRegions* and *Brands* tables *can be connected using simple SMR relationships* to the *Customer*, *Budget*, and *Product* tables.

By *enabling the bidirectional filter on the relationship between Product and Brands*, we materialize through two physical relationships

The virtual relationship *transfers the filter from Product to Budget using the Brand column in both tables*.

In a similar way, *Customer transfers the filter to Budget through the bidirectional filter* between *Customer* and *CountryRegions*.

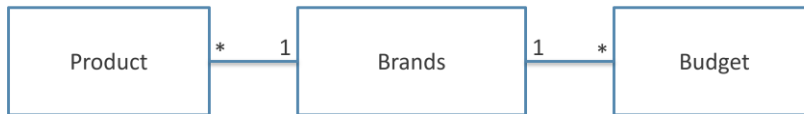


Although we recommend not to use bidirectional filters in a data model, *this is one of the few cases where this practice is safe*.

The *calculated tables* and *two physical SMR relationships* we created are just *an artifact to implement* a virtual relationship between two tables; the resulting virtual relationships transfer the filter in a single direction: *Customer filters Budget* and *Product filters Budget*.

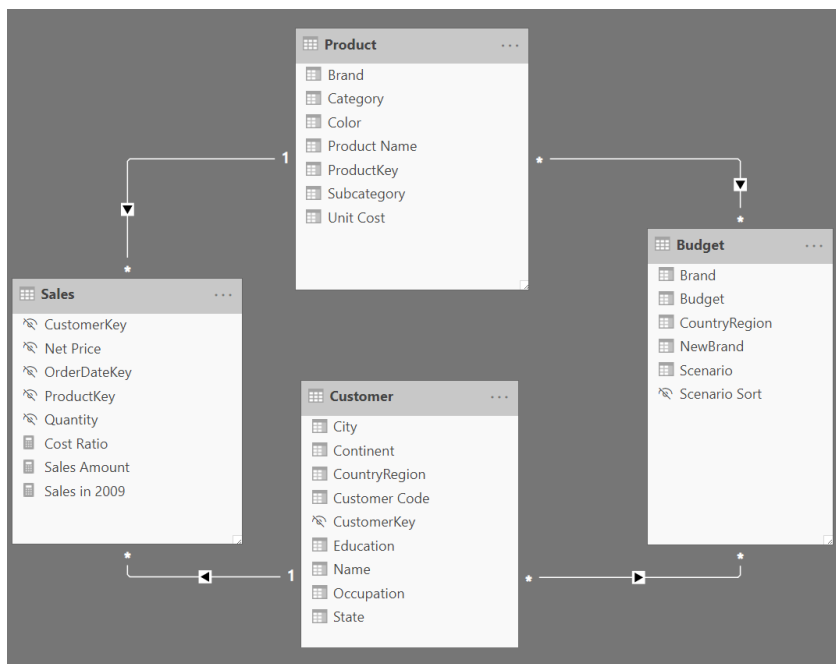
There are no ambiguities. **Note** that the *Brands* and *CountryRegions* calculated tables *do not add any new information to the data model*. They are simply *a way to materialize a table* whose cardinality allows the creation of two SMR relationships for each virtual relationship we need.

The following picture shows the *granularity of the three tables involved* in the virtual relationship between *Product* and *Budget*.



- Contoso Rechargeable Battery E100 Black
- Contoso USB Cable M120 White
- Litware Refrigerator L1200 Orange
- Litware Microwave E090 Grey
- Litware Microwave E080 Grey
- Fabrikam Microwave M1150 Grey
- Fabrikam Microwave M1250 Grey
- ...
- Adventure Works
- Contoso
- Fabrikam
- Litware
- Northwind Traders
- ...
- Adventure Works, USA, 3000
- Adventure Works, UK, 1000
- Contoso, USA, 10000
- Contoso, UK, 3500
- Fabrikam, USA, 4000
- Fabrikam, UK, 1200
- ...

The virtual relationship between *Product* and *Budget* does not have a one-side. We can say that such a virtual relationship has a many-to-many cardinality. The artifact we just created can be obtained using an MMR relationship with a single filter, as shown in the following model.



The relationship between *Product* and *Budget* must be defined with a many-to-many cardinality and a Single cross-filter direction. Power BI also displays a warning when you create an MMR relationship. It is important to keep the cross-filter direction as Single in order to avoid ambiguities in the data model.

Edit relationship

Select tables and columns that are related.

Budget


CountryRegion	Brand	Scenario	Budget	Scenario Sort	NewBrand
China	A. Datum	Low	171,264.00	1	Empty
China	A. Datum	Medium	201,487.00	2	Empty
China	A. Datum	High	231,710.00	3	Empty

Product

ProductKey	Product Name	Brand	Color	Subcategory	Category
743	Contoso Rechargeable Battery E100 Black	Contoso	Black	Computers Accessories	Computers
744	Contoso Dual USB Power Adapter - power adapter E30...	Contoso	Black	Computers Accessories	Computers
745	Contoso Car power adapter M90 Black	Contoso	Black	Computers Accessories	Computers

Cardinality: Many to Many (*:*)
Cross filter direction: Single (Product filters Budget)

☒ Make this relationship active
☐ Assume referential integrity
☐ Apply security filter in both directions

 This relationship has cardinality Many-Many. This should only be used if it is expected that neither column (Budget and Product) contains unique values, and that the significantly different behavior of Many-many relationships is understood. [Learn more](#)

OK Cancel

Even though the MMR relationship is a single entity in the data model, its performance is not as good as the artifact that requires one calculated table and two SMR relationships. Indeed, there are no indexes created to support an MMR relationship in the storage engine. While an MMR relationship is faster than a virtual relationship implemented using [TREATAS](#), it is not as good as an artifact based on two SMR relationships. The difference can be ignored when the cardinality of the column used in the relationship is in the range of 10 to 100 unique values. When there are thousands or more unique values in the column defining a relationship, you should consider creating the calculated table and the two SMR relationships instead of an MMR relationship.

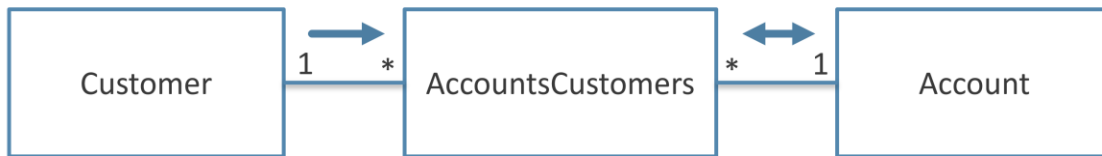
Understanding different types of many-to-many relationships

The Tabular model allows users to create two different types of many-to-many relationships:

- The **many-to-many relationships between dimensions**: these are the “classic” many-to-many relationships between dimensions in a dimensional model. These relationships require a bridge table containing data coming from the data source. The bridge table holds the information that defines the existing relationships between the entities. Without the bridge table, the relationships cannot be established.
- The **relationships at different granularities**: these relationships use columns that do not correspond to the identity of the table, so the cardinality is “many” on both ends of the virtual relationship. A physical implementation of this type of relationship can use two SMR relationships connected to an intermediate table populated with the unique values of the column defining the relationship. The Tabular model enables users to create this relationship using an MMR relationship, which is always a weak relationship.

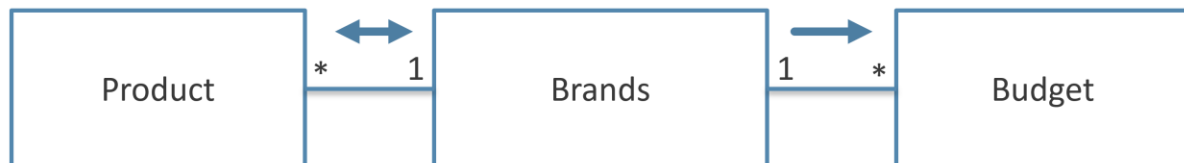
The “classic” many-to-many relationship is implemented using two SMR relationships in the order one-many/many-one. In order to implement a filter propagation from *Customer* to *Account*, the bidirectional filter must be enabled on the relationship connecting the *Account* table to the bridge table – *AccountsCustomers* in this example.

“Classic” many-to-many relationship between dimensions



The relationship at different granularities is implemented using two SMR relationships in the order many-one/one-many, or by using a single MMR relationship in a Tabular model. In both cases, the virtual relationship should propagate the filter in a single direction. If the relationship is implemented using two SMR relationships, in order to propagate the filter from *Product* to *Budget*, the bidirectional filter must be enabled on the relationship connecting the Product table to the intermediate table – *Brands* in this example.

Relationship at different granularities



A single MMR relationship can simplify the creation of a relationship at different granularities. It is considered best practice to specify the correct direction of the relationship, which should be Single. By default, Power BI creates these relationships with a bidirectional filter propagation, which can create ambiguities in the data model.

The filter direction of a weak relationship should be “single” (default is “both”)



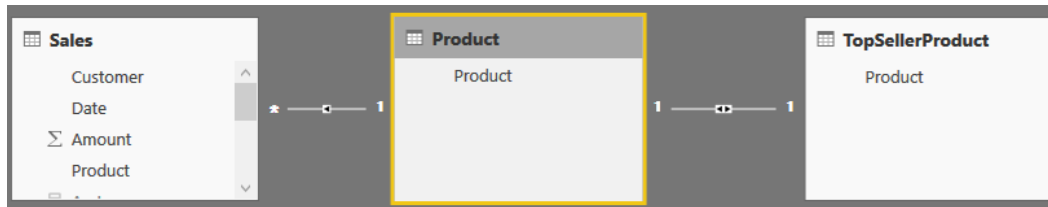
Conclusions

There are different ways to implement a logical relationship in a Tabular model. DAX provides flexible techniques to implement any kind of virtual relationship between entities, but only physical relationships in the data model provide the best results in terms of performance. Filter propagation direction and granularity of the relationships are key concepts to create the proper physical implementation of a logical relationship.

The classic notion of “many-to-many” relationships in Dimensional modeling does not correspond to what is referred to as a “many-to-many cardinality relationship” in a Tabular model. The latter is just an alternative to an artifact – based on two physical relationships and a calculated table – that enables users to define a relationship between entities with a granularity different from the granularity of the tables. For this reason, when referring to a Tabular model it is better to use the acronyms SMR and MMR to identify Single-Many-Relationship and Many-Many-Relationship, respectively. The term “many-to-many” relationship should only be used to describe the logical relationship between dimensions, which is implemented through two SMR relationships in the physical model.

An expanded table contains all the columns of the base table and all the columns of the tables that are linked to the base table through one or more cascading many-to-one or one-to-one relationships.

Consider the following diagram:



There are three tables. Each one has its expanded version:

Native columns

Related columns

Expanded (Product)

Table	Product	TopSellerProduct
Column	Product	Product
	Apple	
	Pie	Pie
	Pizza	Pizza

Expanded (TopSellerProduct)

Table	TopSellerProduct	Product
Column	Product	Product
		Apple
	Pie	Pie
	Pizza	Pizza

Expanded (Sales)

Table	Sales					Product	TopSellerProduct
Column	Customer	Date	Amount	Product	DueDate	Product	Product
	Marco	1/1/2018	100	Apple	3/1/2018	Apple	
	Marco	2/1/2018	100	Pie	4/1/2018	Pie	Pie
	Marco	3/1/2018	100	Pizza	5/1/2018	Pizza	Pizza
	Alberto	2/1/2018	500	Apple	4/1/2018	Apple	
	Alberto	3/1/2018	500	Pie	5/1/2018	Pie	Pie
	Daniele	3/1/2018	1000	Pizza	6/1/2018	Pizza	Pizza

- Expanded (Product) contains Product[Product] and TopSellerProduct[Product]
- Expanded (TopSellerProduct) contains Product[Product] and TopSellerProduct[Product]
- Expanded (Sales) contains all the columns of the three tables

The expanded version of both Product and TopSellerProduct is the same. In fact, a one-to-one relationship is known as an identity. For all intents and purposes you can consider the two expanded tables as being the same. An expanded table is created by joining the columns of two tables into a larger table using a FULL OUTER JOIN.

However, regular many-to-one relationships use the usual [LEFT](#) OUTER JOIN.

Table expansion has nothing to do with bidirectional filtering. Expansion always happens to the 1-side of a relationship. If you activate the bidirectional cross-filter on a relationship, you are not relying on table expansion. Instead, the engine pushes certain filtering conditions in the code in order to apply the filters on both sides. Thus,

in the previous model, if you enable bidirectional cross-filter on the relationship between Sales and Product, this will not add the columns of the Sales table to the expanded Product table.

Each expanded table contains both native and related columns. Native columns are the ones originally present in the table. Related columns are all the columns of related tables, added to the original table through table expansion.

Table expansion does not happen physically. The VertiPaq engine only stores native tables. Nevertheless, the whole DAX semantic is based on the theoretical concept of expanded tables.

Filter propagation

When you learn the [CALCULATE](#) function, you learn that applying a filter on the one-side of a relationship affects the many-side. In fact, if you write this measure:

```
1 AppleSales :=
2 CALCULATE (
3     SUM ( Sales[Amount] ),
4     Product[Product] = "Apple"
5 )
```

The filter applied on Product[Product] follows the relationship between Product and Sales, thus filtering the Sales table too. A better description of that same filter propagation uses the concept of expanded tables.

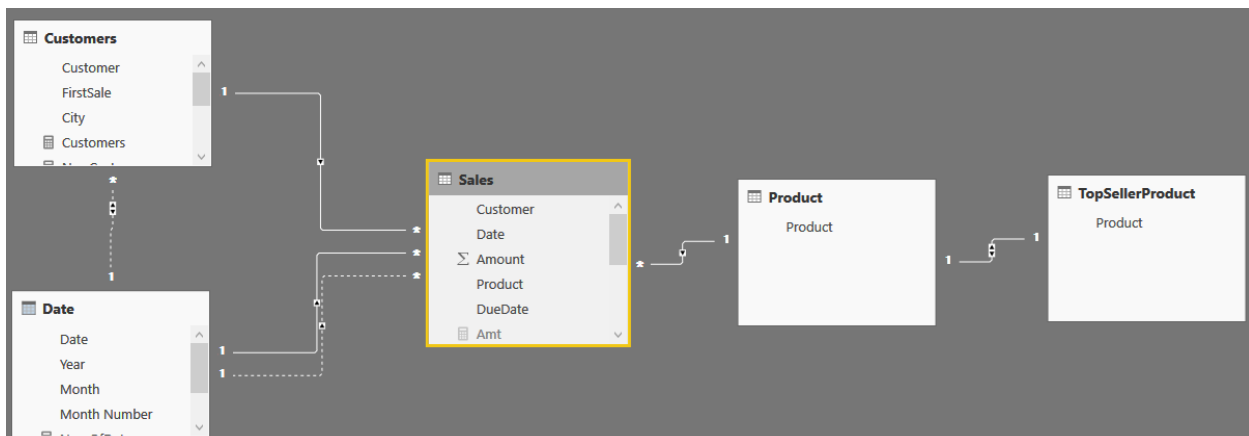
When you filter Product[Product], *all the tables that contain that column* – either native or related – are filtered. Thus, Sales is filtered by Product[Product] because the expanded version of Sales contains Product[Product].

[RELATED](#), [RELATEDTABLE](#) and table expansion

Table expansion includes the concept of relationship. In fact, a relationship is used when the table is expanded and, once you start thinking in terms of expanded tables, you no longer need to think about relationships.

Consider the [RELATED](#) function. When beginning to learn DAX, one typically thinks that [RELATED](#) lets you access columns in related tables. A more accurate way of looking at this is that [RELATED](#) lets you access the related columns of an expanded table.

As an example, consider the following model:



There are direct relationships between Sales, and Product, Date, and Customer. In more appropriate DAX language, we would say that the expanded version of Sales includes all columns of Product, Date and Customer. Thus, Product[Product] belongs to the expanded version of Sales. The expanded version of Sales includes the entire model. Therefore, you could author two columns in Sales using the [RELATED](#) function, like this:

- 1 Sales[TopSellerProduct] = RELATED (TopSellerProduct[Product])
- 2 Sales[Month] = RELATED ('Date'[Month])

The result is the following:

Customer	Date	Amount	Product	DueDate	TopSellerProduct	Month
Marco	01/01/2018	100	Apple	03/01/2018		January
Marco	02/01/2018	100	Pie	04/01/2018	Pie	February
Marco	03/01/2018	100	Pizza	05/01/2018	Pizza	March
Alberto	02/01/2018	500	Apple	04/01/2018		February
Alberto	03/01/2018	500	Pie	05/01/2018	Pie	March
Daniele	03/01/2018	1000	Pizza	06/01/2018	Pizza	March

Not all the products are top sellers, which is why there is a blank value for sales of Apple products in the TopSellerProduct column.

If you are coming from an SQL background, or if you are used to relational databases, you probably think that [RELATED](#) follows relationships. Thus, to compute the Month column, you would think that the engine followed a relationship between Sales and Date and obtained the value of the month by performing a lookup on the Date table.

DAX is different. Date[Month] belongs to the expanded version of Sales, There is a value for [RELATED](#)(Date[Month]) because Sales was expanded to include Date using a relationship. [RELATED](#) requires a row context to be active. If you remove the row context of the calculated column, then [RELATED](#) no longer works. For example, the following calculated column raises an error because [CALCULATE](#) removes the row context performing a context transition:

- 1 Sales[Wrong] = CALCULATE (RELATED (TopSellerProduct[Product]))

Table expansion and variables

One important rule about table expansion is that it happens when you define a table. Look, for example, at the following query:

- 1 DEFINE
- 2 VAR SalesA =
- 3 CALCULATETABLE (Sales, USERELATIONSHIP (Sales[Date], 'Date'[Date]))
- 4 VAR SalesB =
- 5 CALCULATETABLE (Sales, USERELATIONSHIP (Sales[DueDate], 'Date'[Date]))
- 6 EVALUATE
- 7 ADDCOLUMNS (SalesB, "Month", RELATED ('Date'[Month]))

The two variables store the Sales table using two different relationships. SalesA uses the default relationship, whereas SalesB uses the relationship with Sales[DueDate] instead of Sales[Date]. The last [ADDCOLUMNS](#) iterates SalesB and returns the [RELATED](#) Date[Month]. What will the result be? The month of the Sales[Date] column or the month of the Sales[DueDate] column? If you are still thinking in terms of relationships, you are in trouble. In fact, when [ADDCOLUMNS](#) is executed, the active relationship is the relationship using Sales[Date] and you would think that the month is the month of that date. Right? Wrong!

The correct reasoning is as follows: [RELATED](#) accesses the related columns of the expanded version of Sales. SalesB contains the expanded Sales table, and that expansion happened when the active relationship was the relationship

with Sales[DueDate]. As a result, Date[Month] contained in SalesB is related to Sales[DueDate], not to Sales[Date]. Obviously, if you iterate over SalesA, the result will be different.

RELATED in calculated columns

If the developer needs to obtain a [RELATED](#) column using an inactive relationship in a calculated column, they would be in trouble. In fact, as we demonstrated, one could activate an inactive relationship using [USERELATIONSHIP](#). However, this requires using [CALCULATE](#) which in turn, destroys the row context. Thus, the following definition of a calculated column would generate an error:

```
1 Sales[DueMonth] =
2 CALCULATE (
3     RELATED ( 'Date'[Month] ),
4     USERELATIONSHIP ( 'Date'[Date], Sales[DueDate] )
5 )
```

The error is introduced by [CALCULATE](#), which inhibits the usage of [RELATED](#). It would be great if specifying [USERELATIONSHIP](#) as part of [RELATED](#) was an option, but as of today this syntax is unavailable in DAX. [RELATED](#) always uses the active relationship and there is no way to specify an alternative relationship as part of its syntax.

A possible solution is to introduce a row context after [USERELATIONSHIP](#) changes the active relationship – so that table expansion happens with a different set of active relationships. This version of Sales[DueMonth] provides the correct result, although it is not very efficient:

```
1 Sales[DueMonth] =
2 CALCULATE (
3     MINX ( Sales, RELATED ( 'Date'[Month] ) ),
4     USERELATIONSHIP ( Sales[DueDate], 'Date'[Date] ),
5     ALL ( 'Date' )
6 )
```

This code is very intricate, and we urge motivated readers to follow the details of how it works: [CALCULATE](#) activates the new relationship through [USERELATIONSHIP](#); [ALL](#) on Date is required in order to get rid of the filter moved to Date by the context transition executed by [CALCULATE](#). In fact, the context transition still operates using the original relationship and its effect needs to be removed. The inner [MINX](#) reintroduces a row context, in order to use [RELATED](#) to retrieve the date from the Sales expanded table. Obviously, the expansion happened when the active relationship was the one needed. Looks intricate, right? It is... In fact, the suggestion is to never write code like this. It is present in the article for educational purposes but, if one ever needs a piece of code like this, it is much better to rely on a simpler version, which does not use relationships at all:

```
1 Sales[DueMonth] =
2 LOOKUPVALUE (
3     'Date'[Month],
4     'Date'[Date],
5     Sales[DueDate]
6 )
```

This latter version is faster and safer. Yet, if one wants to master relationships, it is important to also understand the previous version. Understand it – but never use it, of course.

Before leaving the topic, it is worth discussing why this alternative version of the code for the due month does not work:

```
1 Sales[DueMonth] =  
2 CALCULATE (  
3     VALUES ( 'Date'[Month] ),  
4     USERELATIONSHIP ( 'Date'[Date], Sales[DueDate] ),  
5     ALL ( 'Date' )  
6 )
```

It looks very similar to the code #6, which works. This time however, it is using [VALUES](#) instead of a less elegant [MINX](#). The reason why this code is not working is that [USERELATIONSHIP](#) only changes the active relationship in the model, so that the table expansion inside of [CALCULATE](#) uses the newly activated relationship instead of the default relationship. [USERELATIONSHIP](#), by itself, does not introduce a filter. It only activates a relationship.

Thus, context transition happens with the old relationship in place. [ALL](#)('Date') removes its effect on the Date table, but [USERELATIONSHIP](#) does not transfer the filter to the Date table. As a result, all dates are still visible. For example, if you were to apply a filter manually – by using [TREATAS](#) instead of [USERELATIONSHIP](#) – then the code would work fine although it would not rely on table expansion:

```
1 Sales[DueMonth] =  
2 CALCULATE (  
3     VALUES ( 'Date'[Month] ),  
4     TREATAS ( { Sales[DueDate] }, 'Date'[Date] )  
5 )
```

In this case, [TREATAS](#) introduces a filter by using the current value of DueDate, and it moves it as a new filter to Date[Date]. Lastly, it is worth to note that – in this case – [ALL](#) on the Date table is no longer needed, because the new filter introduced by [TREATAS](#) overrides it.

SQL Queries vs DAX Queries

We are going to make the initial learning process simple and straight forward by comparing SQL and DAX queries. We will introduce SQL clauses and syntax first followed by their DAX equivalent. We will look at how the SQL clauses in the *Logical SQL query processing steps* below translated into DAX as a functional language.

Logical SQL query processing steps

(4) SELECT (4-2) DISTINCT (4-3) TOP(<top_specification>) (4-1) <select_list>

(1) FROM <left_table> <join_type> JOIN <right_table> ON <on_predicate>

(2) WHERE <where_predicate>

(3) GROUP BY <group_by_specification>

(5) ORDER BY <order_by_list>;

By following and understanding the translation of the *Logical SQL query processing* steps above into functional DAX syntaxes, you would have learned a lot about what you need to know about DAX. Most resources on this topic do not teach or emphasize the functional nature and syntax of DAX so learners leave without a very key element. We are going to emphasize this in this section with *Summarized Functional Syntaxes* that ignores non-functional arguments.

As we translate SQL to DAX functional syntax, one should understand that logically the inner functions are processed first and passed on to the next ones in the function chain, as opposed to the numbered SQL query processing steps shown above.

Database Installation

For those who want to follow the examples by executing both the SQL and DAX queries you can do that by installing the *AdventureWorksDW* SQL Server and SSAS Tabular Model sample databases.

- To download and install the Data Warehouse versions of the *AdventureWorks* SQL Server sample database you can follow the example in this [link](#).
- To download and install the *AdventureWorks* Tabular Model sample database you can follow the examples in the links below.
 1. [Download](#)
 2. [Install](#)

SQL and DAX Client Query Tools

Most readers on this forum should be familiar with SSMS. SSMS is the primary client tool for querying SQL Server Databases. After installing the *AdventureWorks* Tabular Model sample database, one can also connect to the database and run DAX queries using SSMS by following the steps in figure 1 below.

1. First, click on the DAX query button in SSMS.
2. This will launch the screen that lets you select and connect to the SSAS Server with the Tabular Database you install above.
3. Make sure you select the database you want to query
4. Proceed to write and execute your DAX queries

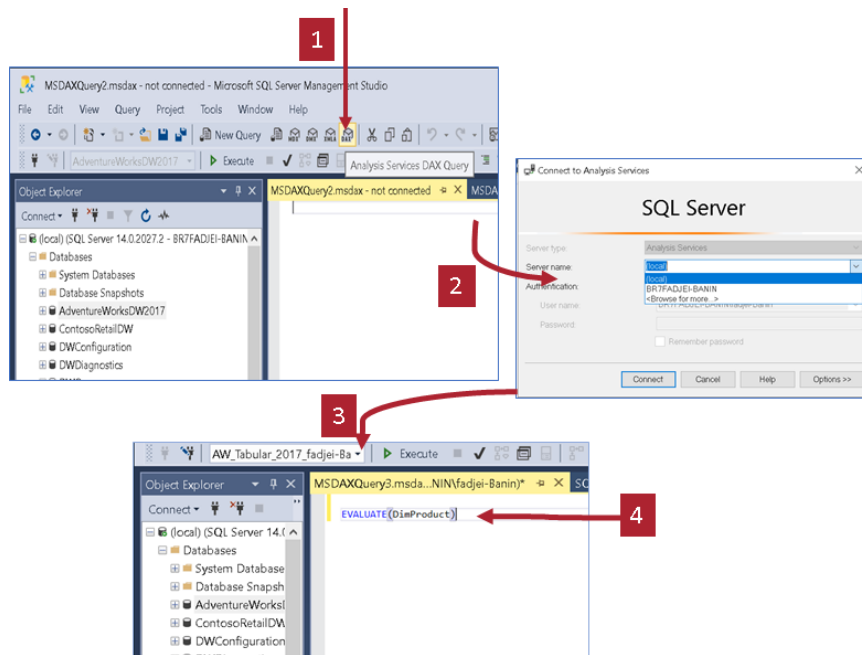


figure 1: Showing how to connect to SSAS Tabular Databases and write DAX queries.

However, if you are going to run a lot of ad-hoc DAX queries and test DAX calculations and expressions, then I suggest you install and use [DAX Studio](http://daxstudio.org) since it offers more DAX features, like formatting. Please visit <http://daxstudio.org> to read the full documentation and download the latest release of DAX Studio.

DAX Table and Column Name Syntax

It is important to know the general format for referencing table and columns in DAX, as shown below;

'Table Name'	// Table reference
'Table Name'[Column Name]	// Column reference

If the table name has no spaces, the single quotes can be omitted, so the syntax looks like below:

TableName[Column Name]

In this series, we are going to use the quoted version whether the table name has spaces or not.

DAX Functions Reference

We will encounter new DAX functions as we go along. For these discussions, a few key ones will be partially introduced. For an in-depth look at any function, one can follow the links provided below.

- <https://docs.microsoft.com/en-us/dax/dax-function-reference>
- <https://dax.guide/>

These function references provide detailed information including syntax, parameters, return values, and examples for each of the functions that will be used DAX queries and formulas in this series.

SQL SELECT FROM Clauses vs DAX EVALUATE Function

Just like you need the SELECT FROM statement in SQL to query tables in a Relational Database, in DAX if you need the EVALUATE function to query the data in a tabular model. Let's start with a simple statement by querying the *DimProduct* Table in both databases with SQL and DAX.

Listing 1:

SQL:

```
Select * From DimProduct
```

DAX:

```
EVALUATE('DimProduct')
```

Both queries in Listing 1 above return all the rows from the DimProduct table. From the DAX query, we see the use of the EVALUATE function, which is the only required function for querying in DAX. Note that EVALUATE was not used with any column projection function so the effect is similar to SELECT * FROM.

Intro to the EVALUATE Function

To query data directly in a tabular model, you need the EVALUATE function. The EVALUATE function is used to return the result set as a table. It is the equivalent to using SELECT FROM statement in T-SQL to return columns and rows from a table.

```
EVALUATE('table name')
```

In this expression 'table name' refers to the name of a table, a table function, or an expression that returns a result set.

Note: Every time you have a DAX function that accepts a table expression as an argument, you can write the name of a table in that parameter, or you can write a function or an expression, that returns a table.

Let's explore the functional nature of DAX a bit further. In the next logic, we will query the top 5 records from the DimProduct table as shown in Listing 2 below.

Listing 2:

SQL:

```
SELECT TOP(5)
*
FROM DimProduct
```

DAX:

```
EVALUATE(TOPN(5,'DimProduct'))
```

TOPN is DAX equivalent of the SQL TOP function, as you can see from Listing 2, the TOPN function is just nested within the EVALUATE function as below, illustrating the functional syntax of DAX.

Summarized Functional Syntax:

```
EVALUATE(TOPN())
```

SQL *WHERE* Clause vs the DAX *FILTER* Function

The DAX *FILTER* Function is the function used to achieve the effects of the *WHERE* clause in the SQL query logical steps.

In the following example, we will write a query to restrict the top 5 records from the DimProduct table to where the color of the product is black and the status of the product is marked as current. The SQL and DAX equivalent of this logic is as shown in *Listing 3* below.

Listing 3:

SQL:

```
SELECT TOP(5)
    *
FROM DimProduct
WHERE [DimProduct].[Color]='Black'
AND [DimProduct].[Status]='Current'
```

DAX (unformatted):

```
EVALUATE(TOPN( 5, ( FILTER( 'DimProduct', 'DimProduct'[Color] = "Black" && 'DimProduct'[Status] = "Current" ) ) )
)
```

DAX(formatted):

```
EVALUATE
(
    TOPN (
        5,
        (
            FILTER (
                'DimProduct',
                'DimProduct'[Color] = "Black"
                && 'DimProduct'[Status] = "Current"
            )
        )
    )
)
```

```
)
)
)
```

Two options of the DAX query are shown in *Listing 3*: an unformatted single line and a more functional, formatted version. The versions are to illustrate the point that, no matter how complex it is, a DAX expression is like a single function call as shown in the functional syntax below.

Summarized Functional Syntax:

```
EVALUATE(TOPN((FILTER())))
```

The complexity of the DAX code comes from the complexity of the expressions that one uses as parameters for the outermost function.

DAX Formatting

As formulas start to grow in length and complexity, it is extremely important to format the code to make it human-readable. Functional language formatting follow certain rules and could be a time-consuming operation. The Folks at SQLBI created a free tool can that transform your raw DAX formulas into clean, beautiful and readable formatted code. You can find the website at www.daxformatter.com. On the website, you can also learn the syntax rules used to improves the readability DAX expressions.

Intro: FILTER Function

The filter function is one of the most important DAX functions you will encounter. The FILTER function, however, has a very simple role: it is an iterator table function. It gets a table, iterates over the table and returns a table that has the same columns as in the original table, but contains only the rows that satisfy a filter condition applied row by row.

The syntax of FILTER is the following;

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

FILTER iterates the <table> and, for each row, evaluates the <condition>, which is a Boolean expression. When the <condition> evaluates to TRUE, the FILTER returns the row; otherwise, it skips it.

Note From a logical point of view, FILTER executes the <condition> for each of the rows in <table>. However, internal optimizations in DAX might reduce the number of these evaluations up to the number of unique values of column references included in the <condition> expression. The actual number of evaluations of the <condition> corresponds to the “granularity” of the FILTER operation. Such a granularity determines FILTER performance, and it is an important element of DAX optimizations.

SQL ORDER BY Clause vs DAX ORDER BY Parameter

The DAX *Order By* keywords or optional parameter is similar to the SQL *Order By* keywords. The query in Listing 4 below shows how the previous result set from the DimProduct table is ordered by EnglishProductName.

Listing 4:

SQL:

```
SELECT TOP(5)

    *

FROM    [DimProduct]

WHERE   [DimProduct].[Color]='Black'

AND     [DimProduct].[Status]='Current'

ORDER BY [DimProduct].[EnglishProductName]
```

DAX:

```
EVALUATE

(

    TOPN (

        5,

        (

            FILTER (

                'DimProduct',

                'DimProduct'[Color] = "Black"

                && 'DimProduct'[Status] = "Current"

            )

        )

    )

)

ORDER BY 'DimProduct'[EnglishProductName]
```

Note that in DAX ORDER BY is not a function; it is rather an optional parameter of EVALUATE function.

Summarized Functional Syntax:

```
EVALUATE(TOPN((FILTER()))

ORDER BY
```

SQL *Select_list* vs DAX Projection Functions

In SQL, the *select_list* enables you to outline the columns you want in your query result set, as in;

```
SELECT column1, column2,... FROM
```

In DAX there are a couple of functions you could use to project table columns to achieve the same results as SQL *select_list*. For *DAX querying* purposes we will use the very efficient SUMMARIZECOLUMNS function which has been designed to be the “one function fits all” to run queries.

The example that follows shows a simple query to illustrate the use of SUMMARIZECOLUMNS function. The queries show how to project *EnglishProductName* and *Color* columns from the DimProduct table.

Listing 5:**SQL:**

```
SELECT  
  
    [EnglishProductName] ,  
  
    [Color]  
  
FROM DimProduct
```

DAX (Option 1):

```
EVALUATE  
  
(  
  
    SUMMARIZECOLUMNS (  
  
        'DimProduct'[EnglishProductName],  // column name  
  
        'DimProduct'[Color],              // column name  
  
        'DimProduct'                      // Base Table name  
    )  
)
```

DAX (Option 2) :

```
EVALUATE  
  
(  
  
    SUMMARIZECOLUMNS (  
  
        'DimProduct'[EnglishProductName], // column name  
  
        'DimProduct'[Color]              // column name  
    )  
)
```



```
)
```

As we can see from the listing above there are two options of the query for DAX, *Option 1* with the name of the table and *Option 2* without the name of the table.

Note that option 2 works and it goes to show that the table expression is optional. However, when selecting from multiple tables you must specify the *base table* to use for the join operation, if not you obtain a cross join. On the other hand, if there is an aggregation expression in your query, the DAX engine would infer the *base table* from the aggregation expression. In this series, we are going to use the first option by always specifying the base table whether projecting from one or more tables.

Let's try a more complex logic by selecting columns in our previous queries from *Listing 4* as shown in *Listing 6* below.

Listing 6:**SQL:**

```
SELECT TOP(5)
    [EnglishProductName] ,
    [Color]
FROM    DimProduct
WHERE   DimProduct.[Color]='Black'
AND     DimProduct.[Status]='Current'
ORDER BY DimProduct.[EnglishProductName]
```

DAX:

```
EVALUATE
(
    TOPN (
        5,
        (
            SUMMARIZECOLUMNS (
                'DimProduct'[EnglishProductName],
                'DimProduct'[Color],
                FILTER (
                    'DimProduct',
                    'DimProduct'[Color] = "Black"
                )
            )
        )
    )
)
```

```

        && 'DimProduct'[Status] = "Current"
    )
)
)
)
)
ORDER BY 'DimProduct'[EnglishProductName]

```

Summarized Functional Syntax:

```

EVALUATE(TOPN(SUMMARIZECOLUMNS(FILTER()))))
ORDER BY

```

SQL GROUP BY Vs DAX Aggregation

Unlike SQL, where you have to explicitly state the *group by* clause, in DAX the feature is built into SUMMARIZECOLUMNS and other column projection functions. The *group by* feature is forced by the introduction of an aggregation function within the SUMMARIZECOLUMNS function as shown in *Listing 7* below.

Listing 7:

SQL:

```

SELECT salesordernumber,
        Sum([salesamount])AS 'SumOfSales'
FROM FactResellersales
GROUP BY salesordernumber

```

DAX:

```

EVALUATE
(
    SUMMARIZECOLUMNS (
        'FactResellerSales'[SalesOrderNumber],
        'FactResellerSales',
        "SumOfSales", SUM ( 'FactResellerSales'[SalesAmount] )
    )
)

```

```
)
)
```

Summarized Functional Syntax:

```
EVALUATE( SUMMARIZECOLUMNS( SUM() ) )
```

As we can see from the DAX logic in Listing 7 above, by the mere introduction of the SUM function, SUMMARIZECOLUMNS function knows to group by the projected columns. Always remember that you must specify *table expression* argument after the columns you projecting and before the aggregated or calculated columns.

Intro: SUMMARIZECOLUMNS Function

We've learned that SUMMARIZECOLUMNS is an extremely powerful query function with all the features needed to execute a query. It returns a summary table over a set of groups. SUMMARIZECOLUMNS lets you specify:

- A set of new columns to add to the result.
- A set of columns used to perform *Group-By*, with the option of producing subtotals. SUMMARIZECOLUMNS automatically groups data by selected columns. The result is equivalent to SQL Select Distinct.
- A set of *filters* to apply to the model before performing the group-by.
- You must specify the table expression argument after the columns you projecting and before the aggregated or calculated columns.
- SUMMARIZECOLUMNS automatically removes from the output any row for which all the added columns produce a blank value.
- In [SUMMARIZECOLUMNS](#) you can add multiple filter tables, which could be useful for queries applied to complex data models with multiple fact tables.
- Because of this aggregating feature SUMMARIZECOLUMNS always return a distinct number of record

SQL Joins vs DAX Relationships

In Part I, we learned that in SQL Server databases, tables are isolated units whilst in Tabular Models, tables are all joined into one physical data model. When you build a tabular model, the relationships you define between tables at design time get established as actual relationships because the DAX engine joins all the tables together with a left join. So essentially, tabular model tables are extended tables. Please refer to Part I of the series where these concepts are well explained.

What these concepts mean is that unlike SQL where you specify joins within the query, DAX uses an automatic LEFT OUTER JOIN in the query whenever you use columns related to the primary table (the table on the left side of the Join).

Let's see how these concepts play out in the SQL and DAX queries. In the next listing, we are going to select from two tables *FactResellerSales* and *DimReseller* with a query that outlines the Reseller name and related total sales amount.

Listing8:

SQL:

```
SELECT  resellername,

        Sum (salesamount) AS sumofsales

FROM    factresellersales

LEFT JOIN dimreseller

ON factresellersales.resellerkey = dimreseller.resellerkey

GROUP BY dimreseller.resellername

ORDER BY dimreseller.resellername
```

DAX:

```
EVALUATE

(

    SUMMARIZECOLUMNS (

        'DimReseller'[ResellerName],

        'FactResellerSales',

        "SumOfSales", SUM ( 'FactResellerSales'[SalesAmount] )

    )

)

ORDER BY 'DimReseller'[ResellerName]
```

As we can see from *Listing 8* above, in SQL we explicitly use a Join statement. On the other hand, because the DAX engine knows the existing relationship between the *FactResellerSales* base table and the *DimReseller* Tables in the model, we can project any column from the two tables without an explicit join statement and seen in the DAX version of the query.

Similarly, if we want to filter the result set above to only Resellers in North America, in SQL we need to explicitly join the *DimSalesTerritory* in order to filter the *SalesTerritoryGroup* as shown in *Listing 9* below. On the other hand, in the DAX version, we just place a filter on the *DimSalesTerritory* Table, and we are good to go. The DAX engine knows the existing relationship between the *FactResellerSales* base table and the *DimSalesTerritory* Table in the model.

Listing 9:**SQL:**

```
SELECT  Resellername,

        Sum (salesamount) AS sumofsales

FROM    factresellersales
```

```

LEFT JOIN dimreseller ON factresellersales.resellerkey = dimreseller.resellerkey

LEFT JOIN dimsalesterritory

ON factresellersales.salesterritorykey = dimsalesterritory.salesterritorykey

WHERE dimsalesterritory.salesterritorygroup = 'North America'

GROUP BY dimreseller.resellername

```

DAX:

```

EVALUATE

(

SUMMARIZECOLUMNS (

'DimReseller'[ResellerName],

FILTER (

'DimSalesTerritory',

'DimSalesTerritory'[SalesTerritoryGroup] = "North America"

),

'FactResellerSales',

"SumOfSales", SUM ( 'FactResellerSales'[SalesAmount] )

)

)

```

Functional Syntax:

```
EVALUATE( SUMMARIZECOLUMNS ( FILTER(),SUM() ) )
```

SQL Sub Queries Vs DAX Sub Queries

Now, let's modify the query in *Listing 9* using a subquery to identify Resellers in North America with sales of more than 500K. *Listing 10* below shows the SQL and DAX version of the subquery.

Listing 10:**SQL:**

```

SELECT resellername,

sumofsales

```

```
FROM (
    SELECT resellername,
           Sum (salesamount) AS sumofsales
    FROM factresellersales
    LEFT JOIN dimreseller
        ON factresellersales.resellerkey = dimreseller.resellerkey
    LEFT JOIN dimsalesterritory
        ON factresellersales.salesterritorykey = dimsalesterritory.salesterritorykey
    WHERE dimsalesterritory.salesterritorygroup = 'North America'
    GROUP BY dimreseller.resellername
) AS subq
WHERE subq.sumofsales > 500000
ORDER BY resellername
```

DAX:

```
EVALUATE
FILTER (
    SUMMARIZECOLUMNS (
        'DimReseller'[ResellerName],
        FILTER (
            'DimSalesTerritory',
            'DimSalesTerritory'[SalesTerritoryGroup] = "North America"
        ),
        'FactResellerSales',
        "SumOfSales", SUM ( 'FactResellerSales'[SalesAmount] )
    ),
    [SumOfSales] > 500000
)
```

```
ORDER BY DimReseller[ResellerName]
```

Functional Syntax:

```
EVALUATE( FILTER( SUMMARIZECOLUMNS( FILTER(), SUM() ) ) )
```

As we can see from the DAX version, a subquery is just a matter of nesting query within a Filter function. In the code above, the subquery that retrieves *ResellerName* and *SumOfSales* is later fed into a FILTER function that retains only the rows where SumOfSales is greater than 500K.

As you get conversant with DAX's functional nature and logical query steps, you will discover that selecting from multiple tables and using subqueries is much easier than in SQL.

Intro to DAX Query Variables and Query Measures

We've learned that EVALUATE is the function required to execute a DAX query. We are going to wrap up DAX queries by looking at DAX *Query Variables* and *Query Measures*. These are variables and calculations you can pre-define and use in your EVALUATE query statements.

A very useful side of DAX is that one can build more complex calculations on top of existing ones, that's what *Query Measures* and *Query Variables* lets you do in DAX queries. They enable you to test complex algorithms with queries by breaking them down into simpler reusable calculations.

The DEFINE keyword is the optional EVALUATE parameter needed to define *Query Variables* and *Query Measures*. Listing 10 below shows a simple example of how to define *Query Variables* and *Query Measures* and call them in an EVALUATE statement.

Listing 10:

```
DEFINE

VAR Profitlimit = 5000

MEASURE 'FactResellerSales'[profit] = SUM ('FactResellerSales'[SalesAmount] ) - SUM
('FactResellerSales'[TotalProductCost])

EVALUATE

FILTER (

SUMMARIZECOLUMNS (

'DimDate'[CalendarYear],

'FactResellerSales',

"Profit", 'FactResellerSales'[profit]

),

[Profit] > Profitlimit

)
```


ORDER BY 'DimDate'[CalendarYear]

In Listing 10 above, the DEFINE keyword has been used to define a *Query Variables* with the VAR keyword and *Query Measures* with the MEASURE keyword. The variable and Measure are then used in the EVALUATE query logic.

Note that in the definition of the *Query Measures*, you must specify the table that hosts the measure. In the example, we are hosting the measure *Profit* in the *FactResellerSales*.

In the next example, let's assume you an Analyst at Adventureworks, you get a request call to provide the Profit Margin of products sold by Resellers. You pretend you know what is being asked of you, and then you google Profit Margin to come up with the formulas below.

$$\text{Profit Margin} = \frac{\text{Profit}}{\text{Product Sales}}$$

$$\text{Profit} = \text{Products Sales} - \text{Cost Of Products Sold}$$

The logic *Listing 11* below shows how you translate the algorithm with DAX Queries by breaking it down into simpler calculations using *Query Measures*.

Listing 11:

```

DEFINE

MEASURE 'FactResellerSales'[ProductSales] =

    SUM ('FactResellerSales'[SalesAmount])

MEASURE 'FactResellerSales'[CostOfProductSold] =

    SUM ('FactResellerSales'[TotalProductCost])

MEASURE 'FactResellerSales'[Profit] = [ProductSales] - [CostOfProductSold]

MEASURE 'FactResellerSales'[ProfitMargin] =

    DIVIDE ( [Profit], [ProductSales] )

EVALUATE

SUMMARIZECOLUMNS (

    'DimProduct'[EnglishProductName],

    'FactResellerSales',

    "Profit Margin", 'FactResellerSales'[ProfitMargin]

)

```

As you can see from *Listing 11*, two Query Measures *ProductSales* and *CostOfProductSold* defined and hosted on the *FactResellerSales* are then subsequently used in the *Profit* calculations. The *ProfitMargin* measure simply invokes and divide the *Profit* and *ProductSales* measures. Finally, using the EVALUATE function, the product names are projected with *ProfitMargin* measure.

In the final **example**, we are going to learn a trick that lets you create arbitrarily shaped filters to use in your queries.

Let's say you are *often asked to filter the query* in *Listing 11* above. E.g. you are often asked to provide ProfitMargin for specific scenarios, like particular Resellers in a particular geography for particular years, etc. Normally one will have to use filter functions to achieve these results (similar to how we translated the SQL *WHERE* Clause above).

In DAX there is a function, called **TREATAS**, that provides a way to add arbitrarily shaped filters to your query making such requests easy to reproduce.

In the example in *Listing 12* below, the previous query in *Listing 11* has been modified to add an arbitrarily shaped filter using the TREATAS function.

Listing 12:

```
DEFINE

MEASURE 'FactResellerSales'[ProductsSales] =

    SUM ('FactResellerSales'[SalesAmount])

MEASURE 'FactResellerSales'[CostOfProductsSold] =

    SUM ('FactResellerSales'[TotalProductCost])

MEASURE 'FactResellerSales'[Profit] = [ProductsSales] - [CostOfProductsSold]

MEASURE 'FactResellerSales'[ProfitMargin] =

    DIVIDE ( [Profit], [ProductsSales] )

EVALUATE

SUMMARIZECOLUMNS (

    'DimDate'[CalendarYear],

    'DimProduct'[EnglishProductName],

    'FactResellerSales',

    // Arbitrary filter

    TREATAS (

        { ( 2013, "Progressive Sports" )

          ,( 2012, "Progressive Sports" )

        },

        'DimDate'[CalendarYear],

        DimReseller[ResellerName]
```

```

),

// End Arbitrary filter

"Profit Margin", 'FactResellerSales'[ProfitMargin]

)

```

In *Listing 12* the query filters ProfitMargin for products specific to a Reseller named "Progressive Sports" for the years 2012 and 2013. The various filter parameters are passed to the query in *Listing 11* by forming arbitrarily shaped filters with the function as shown below.

```

// Start Arbitrary filter

TREATAS (

    {

        ( 2013, "Progressive Sports" ),

        ( 2012, "Progressive Sports" )

    },

    'DimDate'[CalendarYear], DimReseller[ResellerName]

),

// End Arbitrary filter

```

Note that the approach uses the DAX Table Constructor that allows you to define an anonymous table directly in code. The parenthesis { } represents the virtual table in this case two columns and two rows as below.

2012	Progressive Sports
2013	Progressive Sports

When applied in the query it creates a virtual relationship between a virtual table defined with specific virtual values to the base table (the expanded version). The result of the virtual table expression is passed as filters to columns passed as arguments in the TREATAS function, in this case, *CalendarYear* and *ResellerName*. Remember that these columns are columns in the extended version of the *FactResellerSales* base table (please refer to the Expanded Table section in [Part I](#)).

As we can see the [TREATAS](#) function offer one of the best ways to implement a virtual relationship in DAX queries. This approach is very useful if you have a small number of unique values to propagate in the filter. You can read more on DAX Table Constructor [here](#) and on the TREATAS function [here](#).

Intro: DEFINE keyword

DEFINE introduces a query definition section that allows one to define local entities like tables, columns, query variables, and query measures that are valid for all the following EVALUATE statements. There can be a single definition section for the entire query, even though the query can contain multiple EVALUATE statements.

Query Variables

- Variables can be any data type, including entire tables.
- *Query Variables* cannot be used in *Query Measures*.

Query Measures

- In the definition of the measure, you must specify the table that hosts the measure.
- Query measures are useful for two purposes:
 1. To write complex expressions that can be called multiple times inside the query.
 2. They are useful for debugging and for performance tuning measures for instance before they are used in Power BI reports

Summary

After learning some fundamentals about DAX and the Tabular Model in Part I, we had set out to continue the learning process by understanding the functional nature DAX by simply translating SQL queries to DAX queries. We have accomplished that by translating all the causes in the SQL logical processing steps into DAX functional syntaxes.

Take-away

We learned that EVALUATE is the only required function to run a DAX query. Finally, we learned how to use the optional DEFINE parameter to define query variables and query measures that could be used as arguments or parameters within the body of an EVALUATE statement.

Able to accomplish all DAX queries with three functions namely EVALUATE, SUMMARIZECOLUMNS and, FILTER. By learning how these three functions arguments to the summarized functional syntax below one can accomplish most of the DAX queries we wrote in this section.

```
EVALUATE( SUMMARIZECOLUMNS( FILTER() ))
```

M is the *scripting language behind the scene for Power Query*.

M is the informal name of this language. the *formal name is*: Power Query Formula Language!

M is Data Transformation engine in Power BI.

You can *use M for doing any data preparation and data transformation before loading that into your model*.

Instead of bringing three tables of DimProduct, DimProductSubcategory, and DimProductCategory, you can merge them all together in Power Query, and create a single DimProduct including all columns from these tables, and load that into the model.

Loading all of these into the model and *using DAX to relate these with each other means consuming extra memory for something that is not required to be in the model*.

M can simply combine those three tables with each other and based on "Step Based" operational structure of M, they can be simply used to create a final data set.

```

let
    FirstAndLastDayOfMonth = (date) =>
        let
            dated=Date.FromText(date),
            year=Date.Year(dated),
            month=Date.Month(dated),
            FirstDateText=Text.From(year)&"-"&Text.From(month)&"-01",
            FirstDate=Date.FromText(FirstDateText),
            daysInMonth=Date.DaysInMonth(dated),
            LastDateText=Text.From(year)&"-"&Text.From(month)&"-"&Text.From(daysInMonth),
            LastDate=Date.FromText(LastDateText),
            record=Record.AddField([], "First Date of Month", FirstDate),
            resultSet=Record.AddField(record, "Last Date of Month", LastDate)
        in
            resultSet
in
    FirstAndLastDayOfMonth("30/07/2015")

```

Pivot and Unpivot with Power BI - RADACAD

```

let
    DayNumberOfYear= (date) => //start of the code
        let //Function name, and input parameter
            dated=DateTime.FromText(date) //start of function body
        in //date conversion from text
            dated //start of function output
        in //function body output
            dated //start of output lines generated
in //call function by a value
    DayNumberOfYear("07/28/2015")

```

I've put some comments in above script to help you understand each line. In general DayNumberOfYear is name of the function. It accepts an input parameter "date", and convert the parameter from text value to DateTime. the last line of the code calls the function with specific date ("07/28/2015").

**** Note that Date Conversion function is locale dependent. So if the date time of your system is no MM/DD/YYYY then you have to enter date as it formatted in your system (look below the clock on right hand side bottom of your monitor to check the format).**

Day Number of Year, Power Query Custom Function - RADACAD

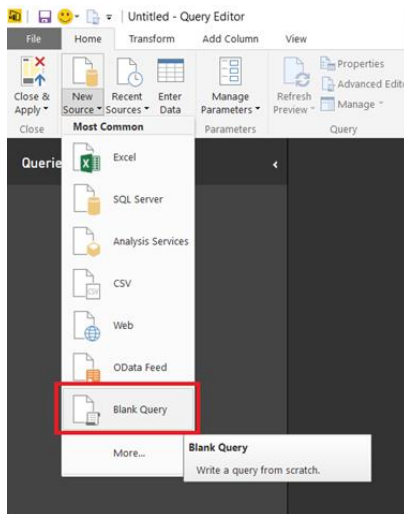
Understanding Let Expressions In M For Power BI And Power Query

When you *start writing M code* for loading data in Power Query or Power BI, one of the first things you'll do is *open up the Advanced Editor* for a query you've already built using the UI.

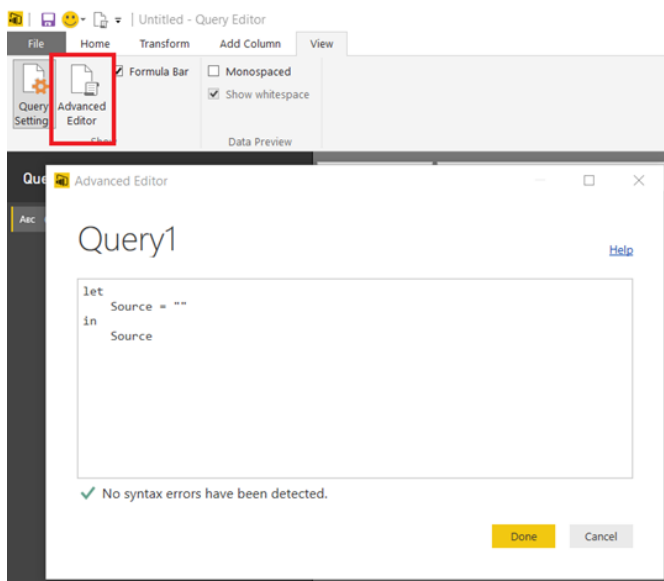
The first step to doing so is to understand how *let* expressions work in M.

Each query that you create in Power BI Desktop or Power Query is a single expression that, when evaluated, returns a single value – and that single value is usually, but not always, a table that then gets loaded into the data model.

To illustrate this, open up Power BI Desktop (the workflow is almost the same in Power Query), click the Edit Queries button to open the Query Editor window and then click New Source/Blank Query to create a new query.

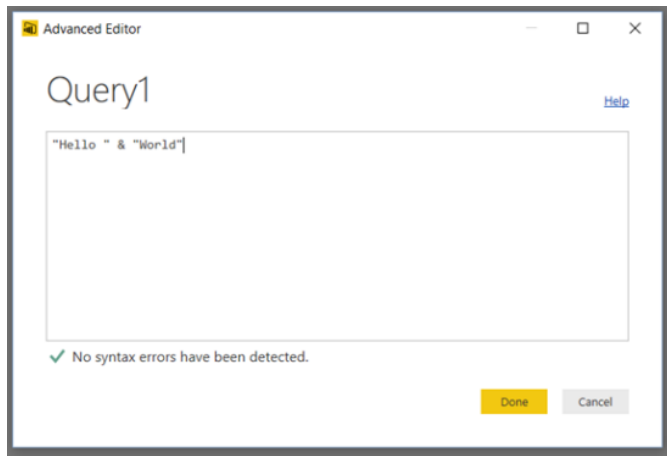


Next, go to the View tab and click on the Advanced Editor button to open the Advanced Editor dialog:

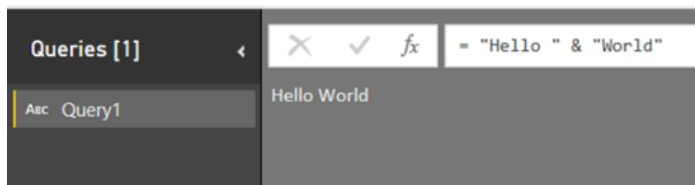


You'll notice that this doesn't actually create a blank query at all, because there is some code visible in the Advanced Editor when you open it. Delete everything there and replace it with the following M expression:

1 "Hello " & "World"



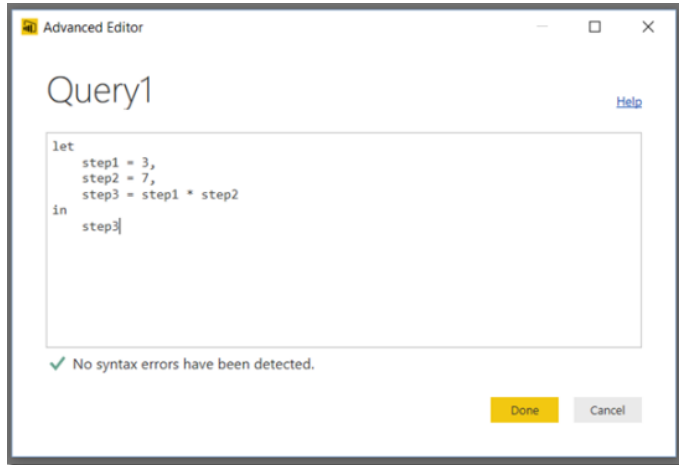
Hit the Done button and the expression will be evaluated, and you'll see that the query returns the text value "Hello World":



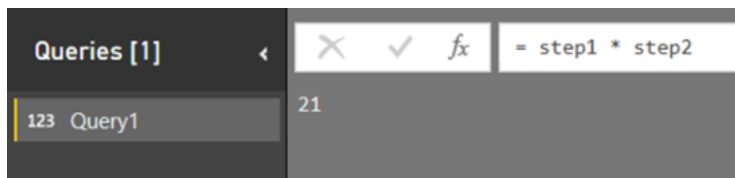
Notice how the ABC icon next to the name of the Query – Query1 – indicates that the query returns a text value. Congratulations, you have written the infamous "Hello World" program in M!

You might now be wondering how the scary chunk of code you see in the Advanced Editor window for your real-world query could possibly be a single expression – but in fact it is. This is where *let* expressions come in: they allow you to break a single expression down into multiple parts. Open up the Advanced Editor again and enter the following expression:

```
1 let
2   step1 = 3,
3   step2 = 7,
4   step3 = step1 * step2
5 in
6   step3
```



Without knowing anything about M it's not hard to guess that this bit of code returns the numeric value 21 (notice again that the 123 icon next to the name of the query indicates the data type of the value the query returns):

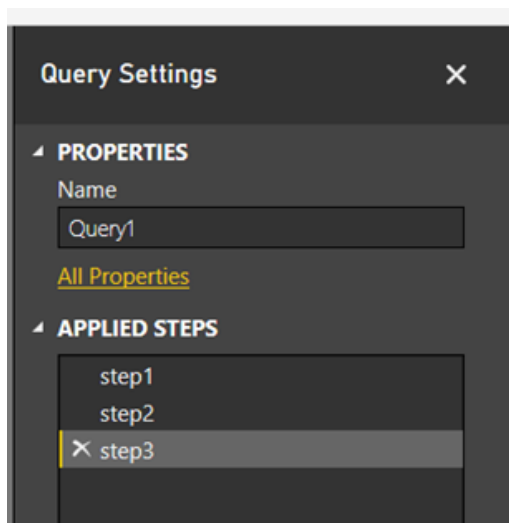


In the M language a *let expression* consists of two sections.

After the *let* comes a list of *variables*, each of which has a name and an expression associated with it. In the previous example there are three variables: step1, step2 and step3.

Variables can refer to other variables; here, step3 refers to both step1 and step2. Variables can be used to store values of any type: numbers, text, dates, or even more complex types like records, lists or tables; here, all three variables return numbers.

The Query Editor is usually clever enough to display these variables as steps in your query and so *displays them in the Applied Steps pane* on the right-hand side of the screen:

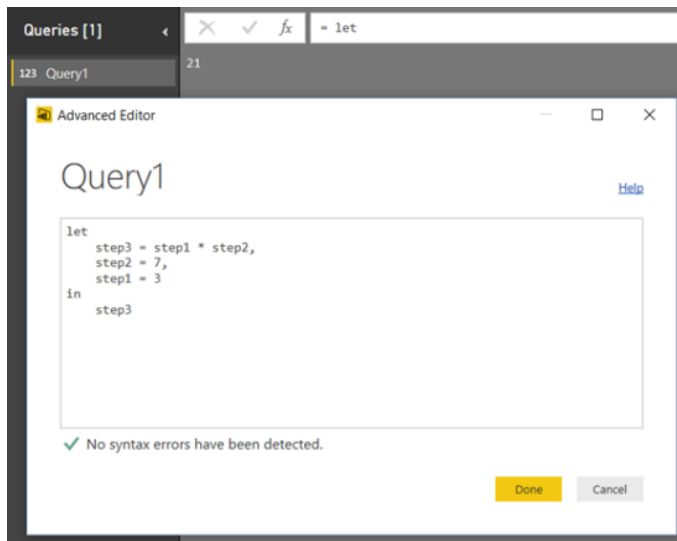


The value that the *let* expression returns is given in the *in* clause. In this example the *in* clause returns the value of the variable *step3*, which is 21.

It's important to understand that the *in* clause can reference any or none of the variables in the variable list. It's also important to understand that, while the variable list might look like procedural code it isn't, it's just a list of variables that can be in any order. The UI will always generate code where each variable/step builds on the value returned by the previous variable/step but when you're writing your own code the variables can be in whatever order that suits you.

For example, the following query also returns the value 21:

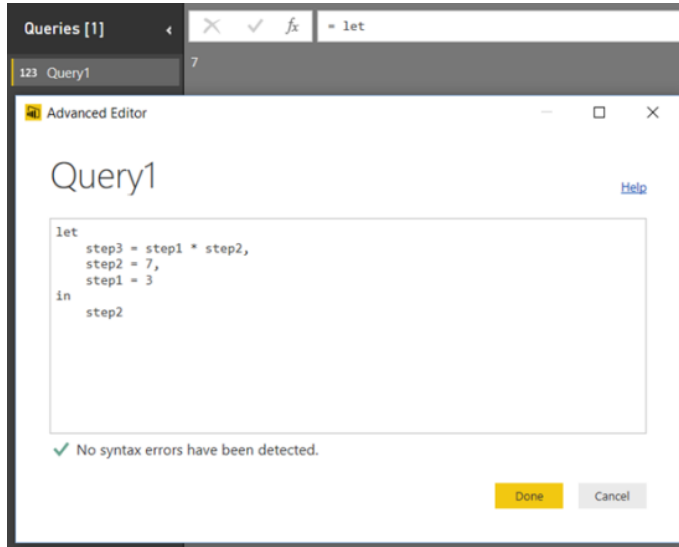
```
1 let
2   step3 = step1 * step2,
3   step2 = 7,
4   step1 = 3
5 in
6   step3
```



The *in* clause returns the value of the variable *step3*, which in order to be evaluated needs the variables *step2* and *step1* to be evaluated; the order of the variables in the list is irrelevant (although it does mean the Applied Steps no longer displays each variable name). What is important is the chain of dependencies that can be followed back from the *in* clause.

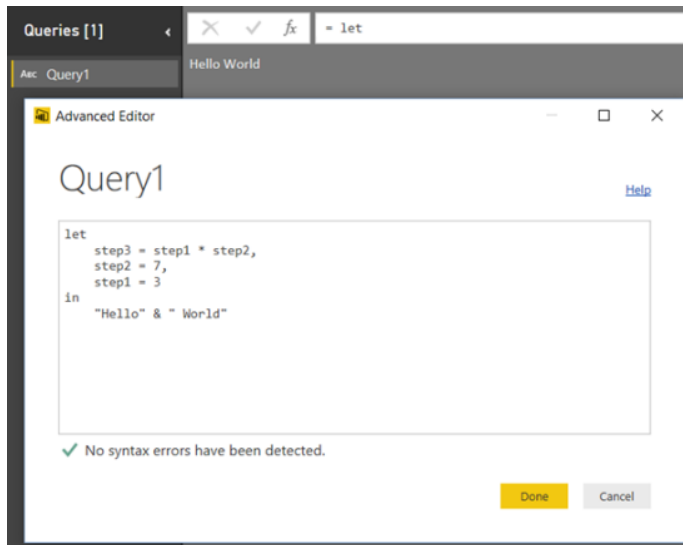
To give another example, the following query returns the numeric value 7:

```
1 let
2   step3 = step1 * step2,
3   step2 = 7,
4   step1 = 3
5 in
6   step2
```



In this case, step2 is the only variable that needs to be evaluated for the entire *let* expression to return its value. Similarly, the query

```
1 let
2   step3 = step1 * step2,
3   step2 = 7,
4   step1 = 3
5 in
6   "Hello" & " World"
```



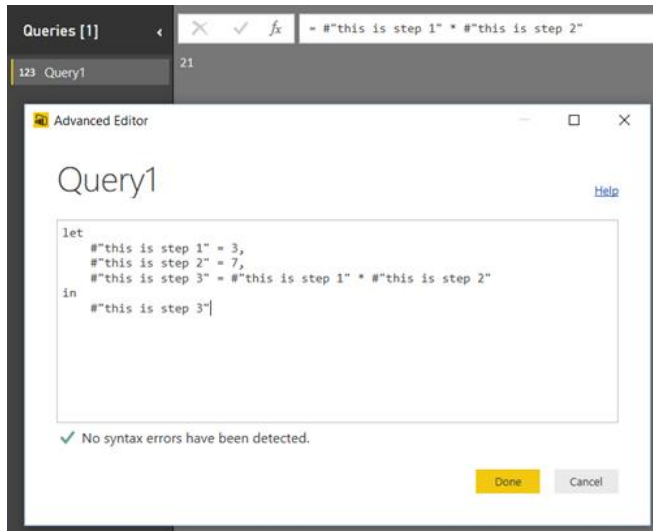
...returns the text value "Hello World" and doesn't need to evaluate any of the variables step1, step2 or step3 to do this.

The last thing to point out is that if the names of the variables contain spaces, then those names need to be enclosed in double quotes and have a hash # symbol in front. For example here's a query that returns the value 21 where all the variables have names that contain spaces:

```

1let
2  #"this is step 1" = 3,
3  #"this is step 2" = 7,
4  #"this is step 3" = #"this is step 1" * #"this is step 2"
5in
6  #"this is step 3"

```



How does all this translate to queries generated by the UI? Here's the M code for a query generated by the UI that connects to SQL Server and gets filtered data from the DimDate table in the Adventure Works DW database:

```

1let
2  Source = Sql.Database("localhost", "adventure works dw"),
3  dbo_DimDate = Source[{"Schema":"dbo",Item="DimDate"}][Data],
4  #"Filtered Rows" = Table.SelectRows(dbo_DimDate,
5                                     each ([DayNumberOfWeek] = 1))
6in
7  #"Filtered Rows"

```

Regardless of what the query actually does, you can now see that there are *three variables declared here*, `"Filtered Rows"`, `dbo_DimDate` and `Source`, and the *query returns the value of the "Filtered Rows" variable*.

You can also see that in order to evaluate the `"Filtered Rows"` variable the `dbo_DimDate` variable must be evaluated, and in order to evaluate the `dbo_DimDate` variable the `Source` variable must be evaluated.

The `Source` variable connects to the Adventure Works DW database in SQL Server; `dbo_DimDate` gets the data from the `DimDate` table in that database, and `"Filtered Rows"` takes the table returned by `dbo_DimDate` and filters it so that you only get the rows here the `DayNumberOfWeek` column contains the value 1.

fx = Table.SelectRows(dbo_DimDate, each ([DayNumberOfWeek] = 1))

	DateKey	FullDateAlternateKey	DayNumberOfWeek	EnglishDayNameOfWeek	SpanishDayNameOfWeek
1	20010701	01/07/2001	1	Sunday	Domingo
2	20010708	08/07/2001	2	Sunday	Domingo
3	20010715	15/07/2001	3	Sunday	Domingo
4	20010722	22/07/2001	4	Sunday	Domingo
5	20010729	29/07/2001	5	Sunday	Domingo
6	20010805	05/08/2001	6	Sunday	Domingo
7	20010812	12/08/2001	7	Sunday	Domingo
8	20010819	19/08/2001	8	Sunday	Domingo
9	20010826	26/08/2001	9	Sunday	Domingo

Query Settings

PROPERTIES

Name

DimDate

All Properties

APPLIED STEPS

Source

Navigation

Filtered Rows

That's really all there is to know about *let* expressions.

A Power Query M formula language query is composed of formula **expression** steps that create a mashup query.

A formula expression can be evaluated (computed), yielding a value.

The **let** expression encapsulates a set of values to be computed, assigned names, and then used in a subsequent expression that follows the **in** statement. For example, a let expression could contain a **Source** variable that equals the value of **Text.Proper()** and yields a text value in proper case.

Let expression

let

Source = Text.Proper("hello world")

in

Source

In the example above, Text.Proper("hello world") is evaluated to "Hello World".

The next sections describe value types in the language.

Primitive value

A **primitive** value is single-part value, such as a number, logical, text, or null. A null value can be used to indicate the absence of any data.

Type	Example value
Binary	00 00 00 02 // number of points (2)
Date	5/23/2015
DateTime	5/23/2015 12:00:00 AM
DateTimeZone	5/23/2015 12:00:00 AM -08:00
Duration	15:35:00

Type	Example value
Logical	true and false
Null	null
Number	0, 1, -1, 1.5, and 2.3e-5
Text	"abc"
Time	12:34:12 PM

Function value

A **Function** is a value which, when invoked with arguments, produces a new value. Functions are written by listing the function's **parameters** in parentheses, followed by the goes-to symbol `=>`, followed by the expression defining the function. For example, to create a function called "MyFunction" that has two parameters and performs a calculation on parameter1 and parameter2:

let

```
MyFunction = (parameter1, parameter2) => (parameter1 + parameter2) / 2
```

in

```
MyFunction
```

Calling the `MyFunction()` returns the result:

let

```
Source = MyFunction(2, 4)
```

in

```
Source
```

This code produces the value of 3.

Structured data values

The M language supports the following structured data values:

- [List](#)
- [Record](#)

- [Table](#)
- [Additional structured data examples](#)

Note

Structured data can contain any M value. To see a couple of examples, see [Additional structured data examples](#).

List

A List is a zero-based ordered sequence of values enclosed in curly brace characters { }. The curly brace characters { } are also used to retrieve an item from a List by index position. See [List value](#_List_value).

Note

Power Query M supports an infinite list size, but if a list is written as a literal, the list has a fixed length. For example, {1, 2, 3} has a fixed length of 3.

The following are some **List** examples.

Value	Type
{123, true, "A"}	List containing a number, a logical, and text.
{1, 2, 3}	List of numbers
{ {1, 2, 3}, {4, 5, 6} }	List of List of numbers
{ [CustomerID = 1, Name = "Bob", Phone = "123-4567"], [CustomerID = 2, Name = "Jim", Phone = "987-6543"] }	List of Records
{123, true, "A"}{0}	Get the value of the first item in a List. This expression returns the value 123.

Value	Type
{	
{1, 2, 3},	Get the value of the second item from the first List element. This
{4, 5, 6}	expression returns the value 2.
{0}{1}	

Record

A **Record** is a set of fields. A **field** is a name/value pair where the name is a text value that is unique within the field's record. The syntax for record values allows the names to be written without quotes, a form also referred to as **identifiers**. An identifier can take the following two forms:

- identifier_name such as OrderID.
- #"identifier name" such as #"Today's data is: ".

The following is a record containing fields named "OrderID", "CustomerID", "Item", and "Price" with values 1, 1, "Fishing rod", and 100.00. Square brace characters [] denote the beginning and end of a record expression, and are used to get a field value from a record. The follow examples show a record and how to get the Item field value.

Here's an example record:

let Source =

```
[
    OrderID = 1,
    CustomerID = 1,
    Item = "Fishing rod",
    Price = 100.00
]
```

in Source

To get the value of an Item, you use square brackets as Source[Item]:

let Source =

```
[
    OrderID = 1,
```

```
CustomerID = 1,  
Item = "Fishing rod",  
Price = 100.00  
]
```

```
in Source[Item] //equals "Fishing rod"
```

Table

A **Table** is a set of values organized into named columns and rows. The column type can be implicit or explicit. You can use #table to create a list of column names and list of rows. A **Table** of values is a List in a **List**. The curly brace characters { } are also used to retrieve a row from a **Table** by index position (see [Example 3 – Get a row from a table by index position](#)).

Example 1 - Create a table with implicit column types

```
let  
Source = #table(  
    {"OrderID", "CustomerID", "Item", "Price"},  
    {  
        {1, 1, "Fishing rod", 100.00},  
        {2, 1, "1 lb. worms", 5.00}  
    })
```

```
in
```

```
Source
```

Example 2 – Create a table with explicit column types

```
let  
Source = #table(  
    type table [OrderID = number, CustomerID = number, Item = text, Price = number],  
    {  
        {1, 1, "Fishing rod", 100.00},  
        {2, 1, "1 lb. worms", 5.00}  
    }  
)
```

```
in
```

```
Source
```


Both of the examples above creates a table with the following shape:

OrderID	CustomerID	Item	Price
1	1	Fishing rod	100.00
2	1	1 lb. worms	5.00

Example 3 – Get a row from a table by index position

let

```
Source = #table(
type table [OrderID = number, CustomerID = number, Item = text, Price = number],
{
    {1, 1, "Fishing rod", 100.00},
    {2, 1, "1 lb. worms", 5.00}
}
)
```

in

```
Source{1}
```

This expression returns the follow record:

OrderID	2
CustomerID	1
Item	1 lb. worms
Price	5

Additional structured data examples

Structured data can contain any M value. Here are some examples:

Example 1 - List with [Primitive](#_Primitive_value_1) values, [Function](#_Function_value), and [Record](#_Record_value)

let

```
Source =
```

```
{
  1,
  "Bob",
  DateTime.ToText(DateTime.LocalNow(), "yyyy-MM-dd"),
  [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0]
}
```

in

Source

Evaluating this expression can be visualized as:

A List containing a Record	
1	
"Bob"	
2015-05-22	
OrderID	1
CustomerID	1
Item	"Fishing rod"
Price	100.0

Example 2 - Record containing Primitive values and nested Records

let

Source = [CustomerID = 1, Name = "Bob", Phone = "123-4567", Orders =

```
{
  [OrderID = 1, CustomerID = 1, Item = "Fishing rod", Price = 100.0],
  [OrderID = 2, CustomerID = 1, Item = "1 lb. worms", Price = 5.0]
}]
```

in

Source

Evaluating this expression can be visualized as:

A record containing a List of Records		
CustomerID	1	
Name	"Bob"	
Phone	"123-4567"	
Orders	OrderID	1
	CustomerID	1
	Item	"Fishing rod"
	Price	100.0
	OrderID	2
	CustomerID	1
	Item	"1 lb. worms"
	Price	5.0

Note: Although many values can be written literally as an expression, a value is not an expression. For example, the expression 1 evaluates to the value 1; the expression 1+1 evaluates to the value 2. This distinction is subtle, but important. Expressions are recipes for evaluation; values are the results of evaluation.

If expression

The **if** expression selects between two expressions based on a logical condition. For example:

if $2 > 1$ then

$2 + 2$

else

$1 + 1$

The first expression ($2 + 2$) is selected if the logical expression ($2 > 1$) is true, and the second expression ($1 + 1$) is selected if it is false. The selected expression (in this case $2 + 2$) is evaluated and becomes the result of the **if** expression (4).

References

1. Understanding Let Expressions In M For Power BI And Power Query:
 - a. <https://blog.crossjoin.co.uk/2016/05/22/understanding-let-expressions-in-m-for-power-bi-and-power-query/>
2. Expressions, values, and let expression:
 - a. <https://docs.microsoft.com/en-us/powerquery-m/expressions-values-and-let-expression>
3. space