Table of Contents

What is DAX? 3
Data Model Components 3
Star Schema Data Model 4
Columnar Database 5
Types of DAX Formulas 5
DAX User Interface in Excel Power Pivot 6
DAX User Interface in Power BI Desktop 7
DAX Fundamentals 8
- Calculated Columns 8
- RELATED DAX function 8
- Row Context 8
- Measures 9
- Filter Context 10
- COUNTROWS DAX Function is a Super Charged COUNTIFS Function 12
- Implicit Measures vs. Explicit Measures 12
- Case Where Implicit Measures Might Be OK 13
- SUMX and Iterator Functions 14
- Row Context in Iterator functions 14
- Two Step and One Step Measures 15
- Characteristics of two methods: 15
- Understanding How Filter Context and Row Context Work Together 16
- Gross Profit 16
- Using Measures in Other DAX Formulas 17
- DIVIDE Function & % Gross Profit 18
- CALCULATE Function and Filter Context 18
- ALL Function 19
- VALUES Function 19
- ALL and VALUES functions compared 19
- Example of ALL and VALUES in CONCATENATEX DAX Function 19
- ALLSELECTED Function to get a Filtered Grand Total 20
- ALLEXCEPT Function 20
- FILTER and CALCULATETABLE functions 20
- Examples of CALCULATE to Change Filter Context for Various % of Total Calculations 21
- Time Intelligence Functions such as SAMEPERIODLASTYEAR 22
- IF function 23
- HASONEVALUE function 23

VAR & RETURN to define Variables in DAX Formula 23
YOY % Change DAX Formula and Report from above formula 23
YOY % Change Formula For Partial Year Data: 24
Boolean Filters in CALCULATE 24
Overwrite Operation in CALCULATE function 26
Boolean Filter Behinds Scenes Runs as a FILTER and ALL Function Construction 26
KEEPFILTERS Function to Perform AND Logical Test Rather Then An Overwrite Operation in CALCULATE 27
FILTER and VALUES Functions to SIMULATE KEEPFILTER Result 27
Logical Operators in DAX: 28
AND Logical Test with Two Filter Arguments in CALCULATE function 29
OR Logical Test in CALCULATE function 29
KEEPFILTERS to Create Filtered Reports 30
OR Logical Test for List of Items using IN Operator in CALCULATE function 30
NOT Logical Test in CALCULATE Function to Filter to Items NOT IN List 30
CALCULATE to perform Context Transition 31
When Context Transition Causes Trouble: Context Transition for a Measure Over a Table with Duplicate Records 33
Rule for When to Use Context Transition: 34
12 Month Moving Average DAX Formula and Report 34
Moving 12 Month Average Formula For Partial Year Data: 35
Complex Filter 35
CROSSJOIN DAX Table Function 35
Star Schema Data Model and Expanded Table Diagram: 37
Table Filter To Go Backwards Across Many-To-One Relationship 38
DAX Formula Evaluation Context Summary 39
Calculating Averages at Different Grains and with Different Formulas 39
Average Transactional / Daily / Monthly Sales By Product DAX Formulas: 41
Unmatched Items in a Relationship 42
Date Table with the DAX functions GENERATE and ROW 43
Using the DISTINCTCOUNT and DIVIDE DAX Functions for faster calculating average 45
Cardinality of Tables in Iterator Functions 45
Approximate Match Lookup with DAX: 46
ADDCOLUMNS and SELECTCOLUMNS DAX Table Functions 47
DAX Studio 47
Using DAX with Existing Connections feature to extract Data From Data Model into Worksheet 49
DAX Query View in Power BI Desktop 52
Video Examples: 54
Define with References and evaluate: 57

What is DAX?

- **DAX** = Data Analysis eXpressions, where eXpression is a synonym for formula.
- DAX Formulas are created in the Data Model in Power Pivot and Power BI.
- DAX formulas are programmed to work efficiently with big data and can significantly reduce the number of intermediate steps in complex calculations by creating columns and tables within formulas at the correct grain.
- DAX includes many functions that are almost identical to Excel worksheet functions, such as SUM, AVERAGE, and EOMONTH, as well as functions that are unique to DAX, such as RELATED, SUMX, and CALCULATE.
- **4 Main Advantages of DAX Formulas**:
    1. Measures are **reusable**, preformatted, metric-in-nature, aggregate formulas that you can drag and drop into Excel PivotTables/Charts and Power BI Visuals: create, format, then use over and over.
    2. DAX formulas are specifically designed to work with the compressed data from the Data Model columnar database to **calculate much more quickly over big data** than many other tools.
    3. There are many **more functions** available in the DAX formula language than there are in a Standard PivotTable; there are more than 250 DAX functions.
    4. The DAX formula language makes it easy and efficient to **use tables with different granularities within a formula**, which can reduce the number of intermediate steps in a complex calculation.

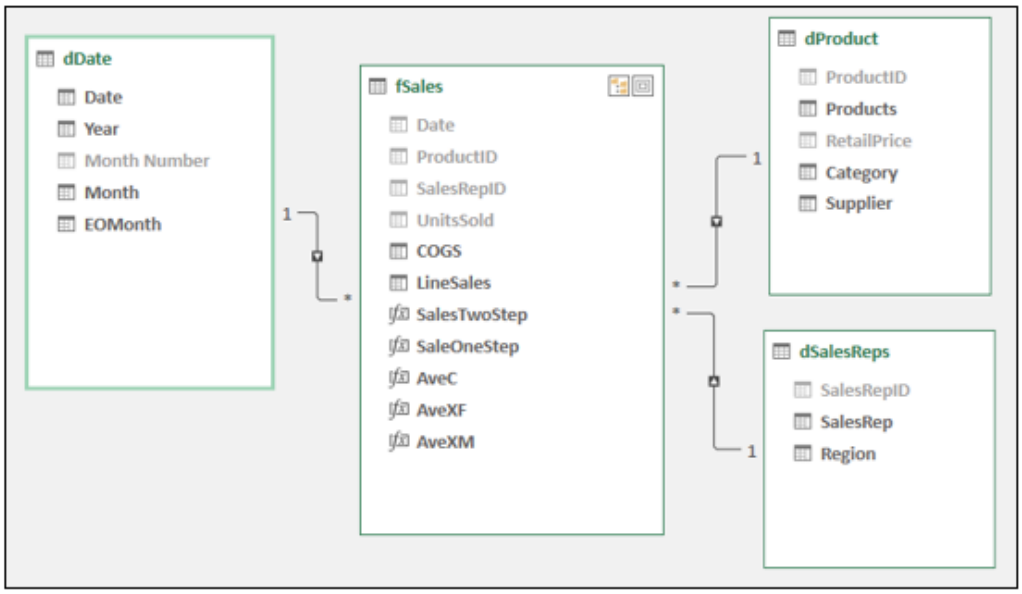
Data Model Components

- A **Data Model** is the source data and metrics for reporting and visuals in Excel Power Pivot and Power BI.
- Data Model Components:
    1. **In-RAM Columnar Database**:
        - When you load Power Query generated tables to the Data Model, the data is compressed and stored in an efficient structure with a small file size in the behind-the-scenes columnar database. The columnar database works with DAX formulas to efficiently work with big data.
    2. **Relationships Between Related Tables**:
        - Relationships are created between related tables and have these characteristics:
            1. They can replace worksheet lookup formulas.
            2. They allow you to drag & drop fields from multiple tables into Excel PivotTables and Power BI Visuals.
            3. Relationships pass filters from dimension tables to fact tables and thereby filter the fact tables and reduce the number of rows that formula must iterate over.
        - Types of Relationships:
            1. One-to- many relationship, where the primary key in the dimension table represents the one-side, and the foreign key in the fact table represents the many-side. Like with the relationship between a fact table & dimension table product field.
            2. Many-to-many relationship, where both columns can have duplicate values. Like with the relationship between a sales order fact tables and an invoice fact table, where an order can show up on many invoices, and an invoice can contain many orders.
            3. One-to-one relationship: In this type of relationship, each column contains a unique list. Like with a company car issued to an employee or student IDs issued to a student.
    3. **Three Types of DAX Formulas**
        1. DAX Calculated Columns
        2. Measures
        3. DAX Table Functions/Formulas
    4. **Hidden Columns or Tables**.
        - Columns of raw data and columns /tables that serve as intermediate steps in calculation process, but that are not needed in report area should be hidden.

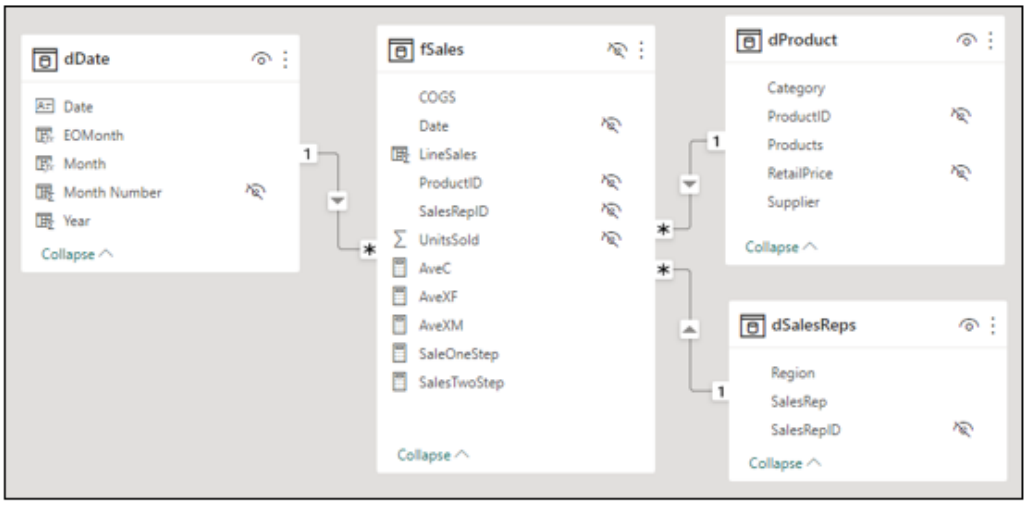
Star Schema Data Model

- A **Star Schema Data Model** contains one fact table along with one or more dimension tables, relationships between the tables and premade reusable formulas. DAX Formulas and the columnar database are designed to work efficiently with a Star Schema Data Model Design.
    1. **Fact table**: A fact table contains the data that you want to summarize or measure (such as sales amounts or units sold). Fact tables usually contain foreign keys that connect through a one-to-many relationships to a primary key in the dimension tables. Fact tables can sometimes be very large; they may contain 100,000 rows, 1 million rows, or even 100 million rows or more.
    2. **Dimension table** (lookup table): A dimension table contains a primary key field with a unique list of entities/elements (such as product IDs) with attributes in subsequent fields for dragging into report or visuals to make conditional calculations (such as product names) or lookup items (such as product prices). Dimension tables are usually much smaller than fact tables.
- **Examples:**
- We will use this data model for most of the calculations in the video. This Data Model can be found in the files named "16-M365ExcelClassStart.xlsx" and "16-M365ExcelClassStart.pbix" (there are finished files also).

Excel Power Pivot:



Power BI Desktop:



Columnar Database

- This database was specifically designed by Microsoft to work efficiently with big data for data analysis and is part of the Data Model tool.
- A columnar database is a special type of behind-the-scenes data storage location that is loaded into random access memory (RAM) when you open an Excel or Power BI file.
- A columnar database compresses the data into a smaller size and stores the data one column at a time (hence the "columnar" name).
- For each column of data that is stored, the columnar database stores a unique list of items.
- The more unique items in any given column, the larger the size of the stored column in the database.
- The database also stores a type of mapping that allows it to reconstruct records of data from the columnar database when a DAX formula makes its calculation.

Original Table with 606 records:

Date	ProductID	SalesRepID	Units
3/19/2021	2	4	80
4/8/2021	2	4	5
4/12/2021	2	4	88
4/12/2021	2	4	70
4/22/2021	2	4	6
5/12/2021	2	4	5
5/24/2021	2	4	1
6/11/2021	2	4	92
6/11/2021	2	4	91
6/17/2021	2	4	209
7/5/2021	2	4	91
7/18/2021	2	4	66
7/20/2021	2	4	110

How Columnar Database compresses tables of data:

Records in original table:	606
# Columns:	4
Total cells with data: 606*4 =	2424

Total cells in columnar database: 426 + 4 + 4 + 187 =	621

Unique counts for each column:

Count	Count	Count	Count
426	4	4	187

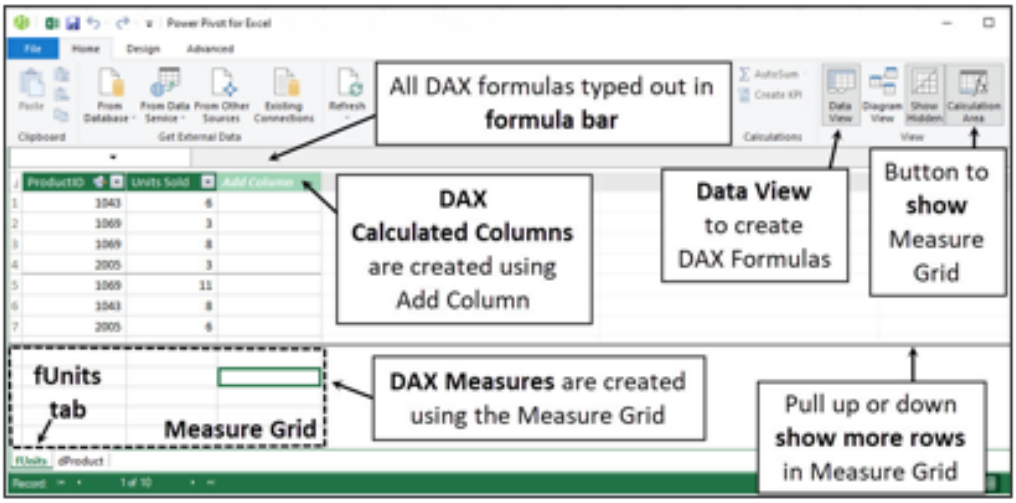
Date	ProductID	SalesRepID	Units
3/19/2021	2	4	80

Types of DAX Formulas

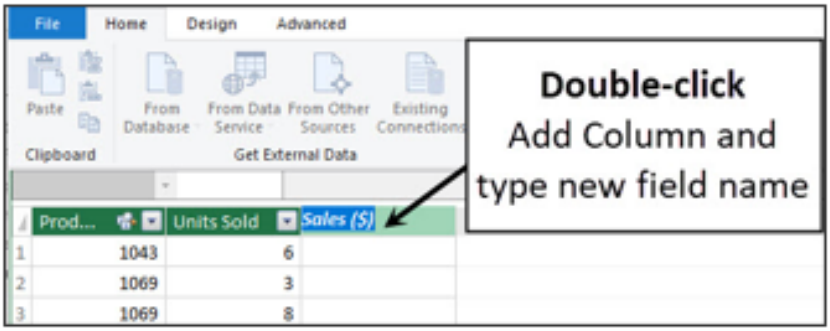
- **DAX Calculated Columns**: Formula columns added to a table in the Data Model. Like a Sales column (intermediate step before a DAX Measure aggregates) or Year/Month column (attribute field use in reports and visuals). The data created from the formula is stored in the columnar database. Calculated columns are evaluated (calculated) when the columnar database is refreshed.
- **DAX Measures**: Reusable, preformatted, aggregate formulas that can be used in reports, visuals, and other areas in the Data Model. Like a Sum or an Average. Measures have ability to work with big data, make complex calculations in less complicated ways. The data created inside the formula is NOT stored in the columnar database. Measures are evaluated (calculated) when they are dropped into a report or visual or when conditions or criteria are changed for the report or visual.
- **DAX Table Formulas**: These formulas create tables.
    - In Power Pivot you can use DAX Tables in other formulas, like using VALUES in the SUMX function.
    - In Power BI you can use DAX Tables in other formulas or you can created tables and store them in the Data Model.

DAX User Interface in Excel Power Pivot

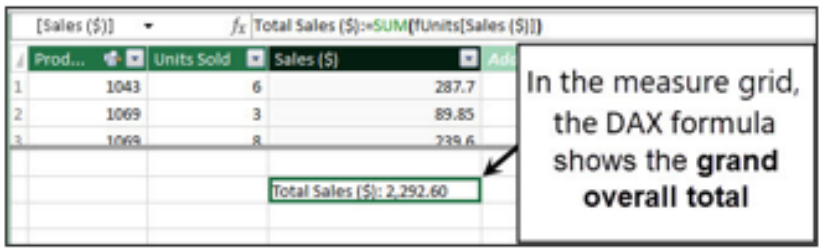
1) Create DAX Formulas in Data View



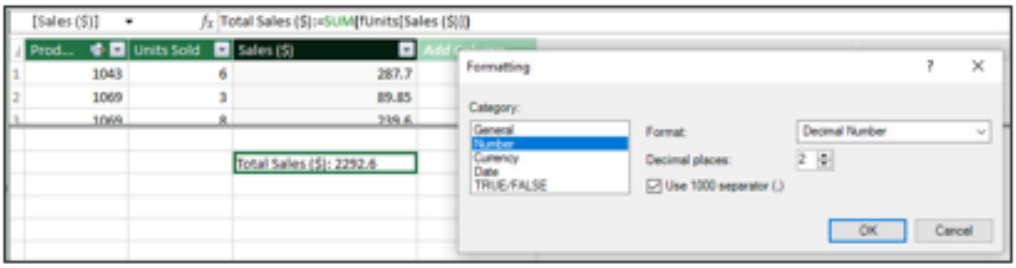
2) Calculated Columns are added to tables



3) Measures are created in Measure Grid below Table

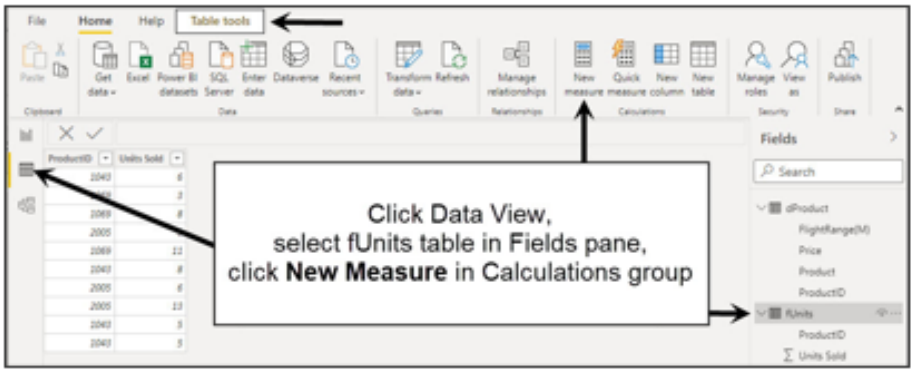


4) For Measure, add Number Formatting & check result in Measure Grid

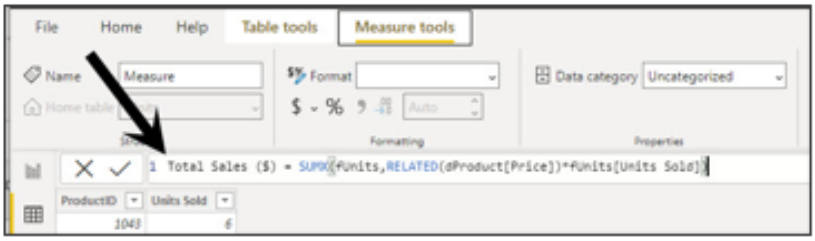


DAX User Interface in Power BI Desktop

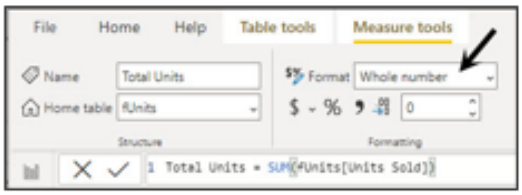
1) Create DAX Formulas in Data View



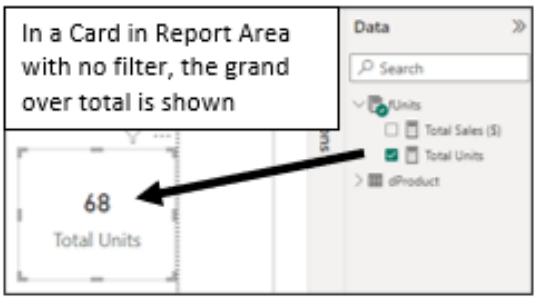
2) Create Formulas in Formula Bar (both Calculated Columns and Measures)



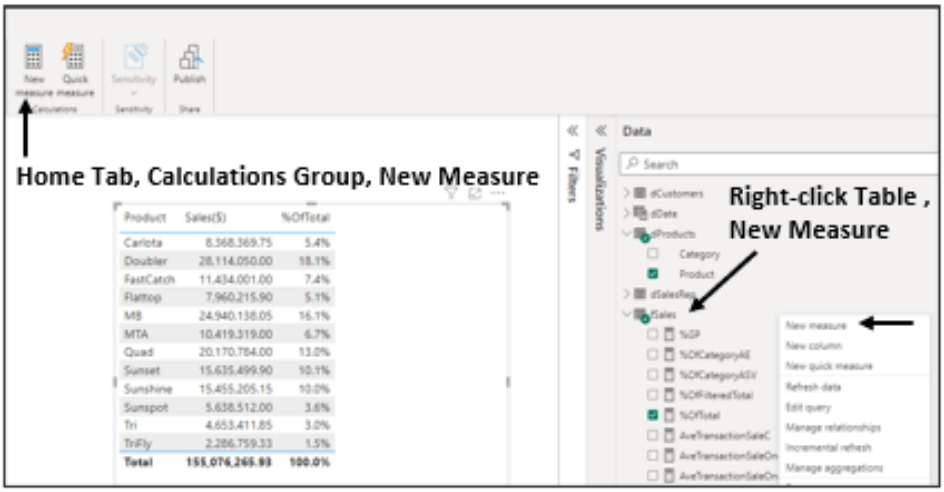
3) Add Number Formatting



4) Check Formula Result in Card in Report View



5) You can also create Measures in Report View and Test them in a Table visual:

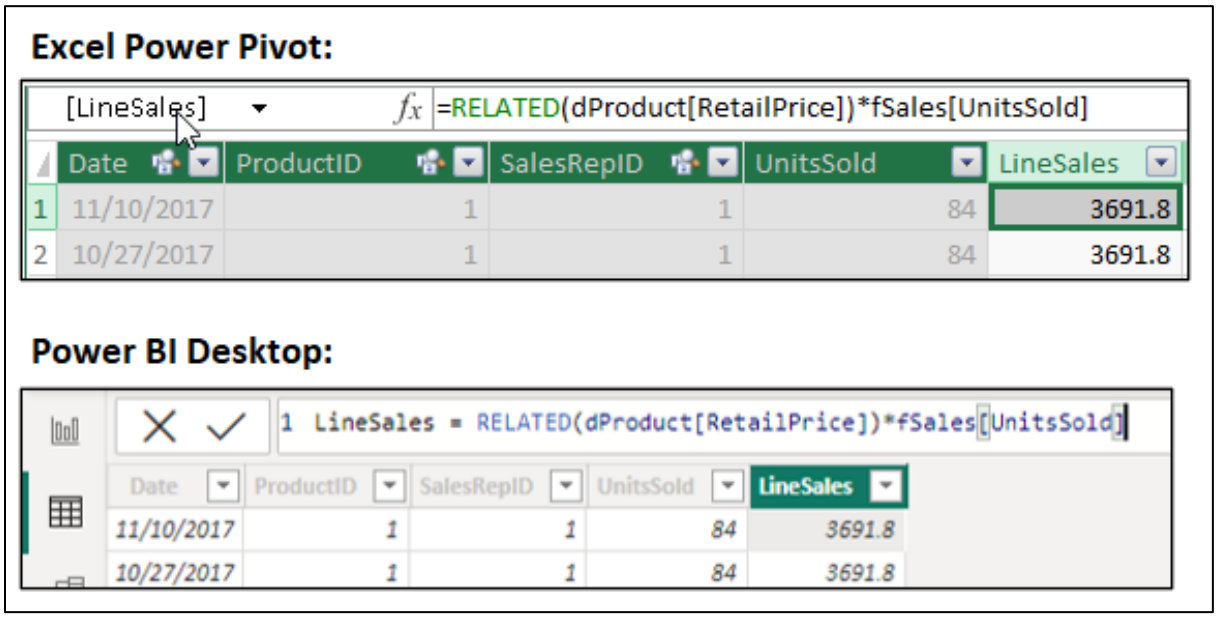


DAX Fundamentals

For most of the examples in the video, we will use in the files named “16-M365ExcelClassStart.xlsx” and “16-M365ExcelClassStart.pbix” (there are finished files also).You can open these and follow along.

Calculated Columns

- Calculated Columns are like Custom Columns in Power Query or Table Columns in Excel Tables, where you use a formula to add data to a table such as a sales amount in a fact table or a date attribute like month in a date dimension table.
- The data is stored in the In-RAM columnar database and the column is evaluated (calculated) when you create them and when you refresh the table.
- In Power Pivot and Power BI, you create Calculated Columns in Data View. The picture below shows a Calculated Column for Line Sales.
- When you create a Calculated Column, there are no cell references like there are in the worksheet. Instead, you use column references, similar to a column reference in an Excel table, like *[@UnitsSold]*, or the each keyword and column references in Power Query, like *each [UnitsSold]*.
- Types of Calculated Columns: 1) Table record amounts that will be aggregated used in other formulas, 2) Dimension Table attribute columns, 3) Helper Columns to make Measures less complicated



RELATED DAX function

- When you have a one-to-many relationship between a fact table and a dimension table, you can use the RELATED function on the fact table many-side to perform an exact match lookup.
- Because there is a relationship, the only input needed is the reference to the column in the dimension table that has the value you want to retrieve. As shown in the above picture, to lookup a product price, all the REALTED function needs is the price column reference.

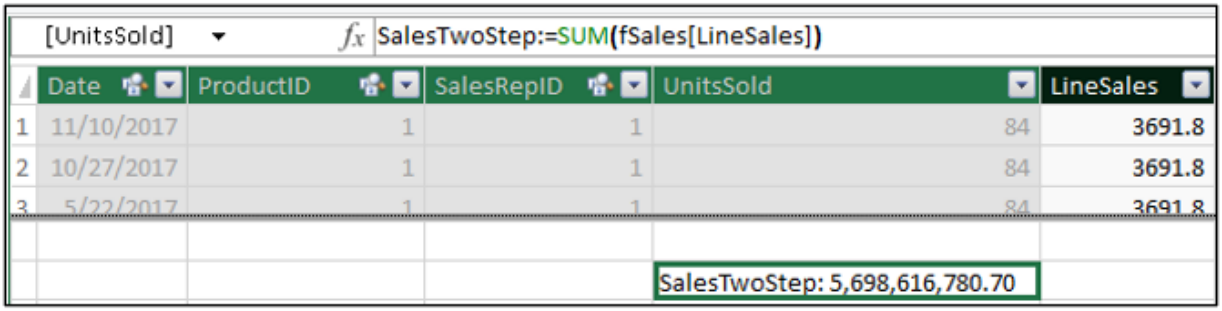
Row Context

- In a DAX calculated column, the formula automatically picks out the correct row value in each row by using something called **row context**.
- In DAX formulas, row context is automatically generated when you use calculated columns or when you use iterator functions such as the SUMX or FILTER.
- In the above picture, row context allows the formula to pick out the correct units in each row and allows the RELATED function to pick out the correct ProductID to lookup the product price.

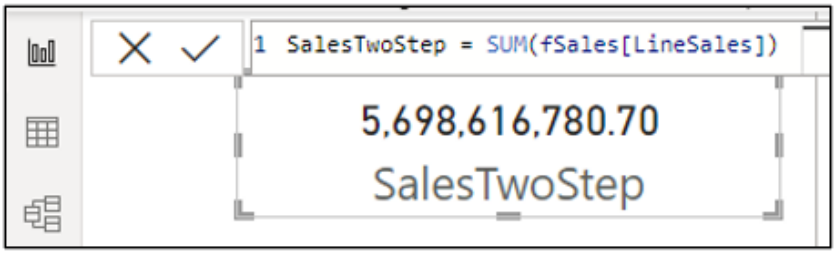
Measures

- Measures are reusable, pre-formatted, aggregate / scalar formulas that you use in PivotTable Reports, PivotCharts, Power BI Visuals and in other DAX formulas.
- The data that is created in a Measure is not stored in the In-RAM columnar database, but instead it is calculated each time you use the Measure or change the conditions or filters in a report or visual.
- As shown below, in Power Pivot you create Measures in the Measure Grid below tables that are shown in Data View. When you create a Measure in Power Pivot the assignment operator is := (colon, equal sign). The convention is: **MeasureName : = Formula**. In Excel Power Pivot, Measures can also be created in the Measures dialog box in the Excel worksheet area.
- As shown below, in Power BI, you select a table in Report view or Data view, right-click the table, point to New Measure, then create formula in formula bar. When you create a Measure in Power BI the assignment operator is = (equal sign). The convention is: **MeasureName = Formula**. In Report View, you can use a card visual to check the result.

Excel Power Pivot:



Power BI Desktop:



Filter Context

When you drop the Measure into a PivotTable or Power BI Visual, the same Measure works in every cell, but the conditions and criteria from the report or visual filter the fact table down to just the rows that match the conditions and criteria, as shown here:

Same Measure in every cell, but in this cell, filter context causes the fact table to be filtered down to just the rows for Aspen, then the Measure makes its calculation. The Measure does not have to work over the whole fact table, just the Aspen records: this helps the formula to calculate quickly.

Products	SalesTwoStep	
Aspen	253,016,877.15	SalesTwoStep:=SUM(fSales[LineSales])
Beaut	347,103,792.90	SalesTwoStep:=SUM(fSales[LineSales])
Bellen	253,361,736.05	SalesTwoStep:=SUM(fSales[LineSales])
Carlota	253,820,067.55	SalesTwoStep:=SUM(fSales[LineSales])
Yanaki	848,229,570.30	SalesTwoStep:=SUM(fSales[LineSales])
Grand Total	**1,955,532,043.95**	SalesTwoStep:=SUM(fSales[LineSales])

Filter context allows DAX measures that are used in a PivotTable, PivotChart, or Power BI visualization to automatically use the conditions and criteria in the Rows, Columns, or Filters areas to make conditional calculations. This is the context part, where the DAX measure can see the surrounding situation and pick out all the conditions in the context of the report or visualization. In addition, when the behind-the-scenes Data Model engine evaluates the DAX context, it does not use the full fact table but instead uses only rows that match the conditions. This is the filter part, as the underlying tables are filtered down to a smaller size so that the formula has less work to perform and can make the calculation more quickly.

Putting it all together, filter context for DAX measures works as follows:

1. When a DAX measure is dropped into a PivotTable, PivotChart, or Power BI visualization, the measure automatically detects all the external row, column, and filter conditions. For any given cell, the row, column, and any filters determine the measure's current external filter context. If there are conditions or filters internally inside the Measure, the external and internal filters are merged to get the final filter context (conditions for filtering the fact table).
2. The dimension table is filtered down to a smaller size, based on the final filter context.
3. The relationship transfers the filters to the fact table.
4. The fact table is filtered down to a smaller size, based on the conditions from the final filter context.
5. The DAX measure works with the filtered fact table so that the formula has less work to perform and can make the calculation more quickly.

Visual of the Filter Context:

1) Before Measure calculates in "Aspen" both tables have all rows showing

Fact Table = fSales = 2,204,103 Rows **Dimension Table = dProduct =16 Rows**

*** Many** **1**

Date	ProductID	SalesRepID	UnitsSold
12/28/2017	1	2	7
1/18/2017	7	4	95
10/12/2017	12	2	72
12/12/2017	1	1	4
11/9/2017	6	15	6
1/24/2017	7	1	72

ProductID	Products	RetailPrice	Category	Supplier
1	Quad	43.95	Freestyle	Gel Boomerangs
2	Yanaki	25.95	Beginner	Colorado Boomerangs
3	Eagle	19.95	Advanced	Channel Craft
4	Bellen	24.95	Beginner	Gel Boomerangs
5	Aspen	24.95	Beginner	Colorado Boomerangs
6	Carlota	24.95	Freestyle	Gel Boomerangs

2) Row condition "Aspen" flows into Measure and filters the dimenstion table

Fact Table = fSales = 2,204,103 Rows **Dimension Table = dProduct =1 Row**

*** Many** **1**

Date	ProductID	SalesRepID	UnitsSold
12/28/2017	1	2	7
1/18/2017	7	4	95
10/12/2017	12	2	72
12/12/2017	1	1	4

ProductID	Products	RetailPrice	Category	Supplier
5	Aspen	24.95	Beginner	Colorado Boomerangs

3) Row condition "Aspen" flows across the relationship and filters the fact table

Fact Table = fSales = 110,185 Rows **Dimension Table = dProduct =1 Row**

*** Many** **<<<<<<<<<<<<<<<<** **1**

Date	ProductID	SalesRepID	UnitsSold
11/20/2017	5	4	216
12/2/2017	5	15	164
3/20/2017	5	3	8
3/21/2017	5	3	60
5/12/2017	5	21	240

ProductID	Products	RetailPrice	Category	Supplier
5	Aspen	24.95	Beginner	Colorado Boomerangs

COUNTROWS DAX Function is a Super Charged COUNTIFS Function

The next calculation task is to count how many transactions there are for each product; this is a frequency calculation. To do such a calculation using worksheet formulas, you would have to use a formula like COUNTIFS(fTransactions[Product],B2), where you specify the product column and the specific product to count. Then, if you wanted a new formula to count the number of transactions for a different column, you would have to create a new formula based on the new column, such as this formula to count the number of transactions by category: COUNTIFS(fTransactions[Category],B2).

With DAX formulas, you can build a single frequency formula that will work on any set of conditions. To do so, you use the COUNTROWS DAX function, which counts the number of rows in a table based on the current filter context. In a star schema data model, if you use the fact table inside COUNTROWS, the formula will count records in the fact table based on any condition. This makes a formula like COUNTROWS(FactTable) a one-stop-shopping frequency formula that can be used to count based on any set of conditions. This is an example of a DAX formula that is much easier to create and use than worksheet formulas or other tools. Here is an Example from Power BI Desktop and Power Pivot:

Power BI:

```
1 Count = COUNTROWS(fSales)
```

Products	Count
Aspen	110,885
Beaut	104,836
Bellen	110,787
Carlota	110,919
Yanaki	355,217
Total	**792,644**

Power Pivot:

```
fx Count:=COUNTROWS(fSales)
```

Products	Count	
Aspen	110,885	← Fact Table filtered down to 110885 rows, and COUNTROWS counts rows!
Beaut	104,836	← Fact Table filtered down to 104836 rows, and COUNTROWS counts rows!
Bellen	110,787	← Fact Table filtered down to 110787 rows, and COUNTROWS counts rows!
Carlota	110,919	← Fact Table filtered down to 110919 rows, and COUNTROWS counts rows!
Yanaki	355,217	← Fact Table filtered down to 355217 rows, and COUNTROWS counts rows!
Grand Tot	**792,644**	← Fact Table filtered down to 792644 rows, and COUNTROWS counts rows!

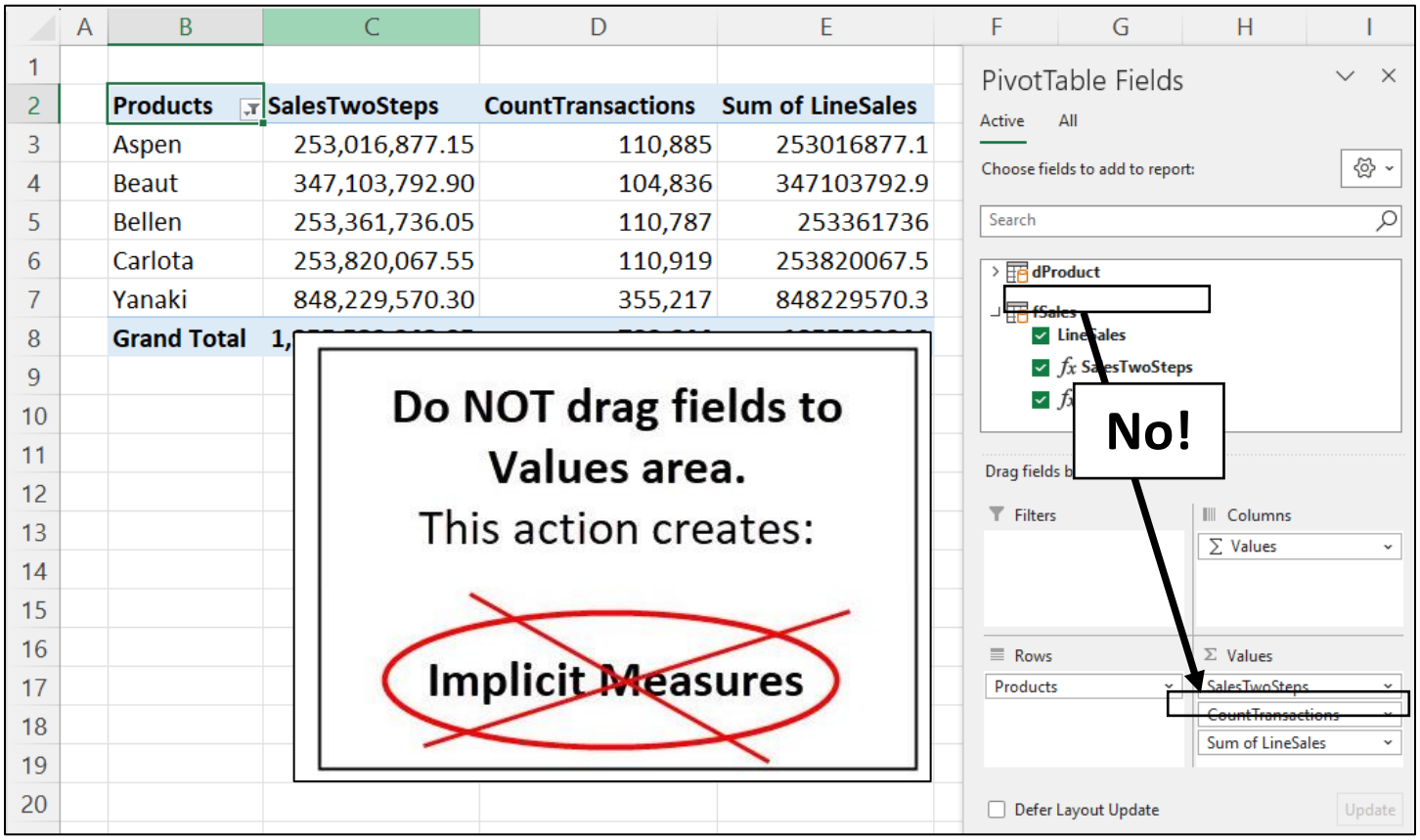
Implicit Measures vs. Explicit Measures

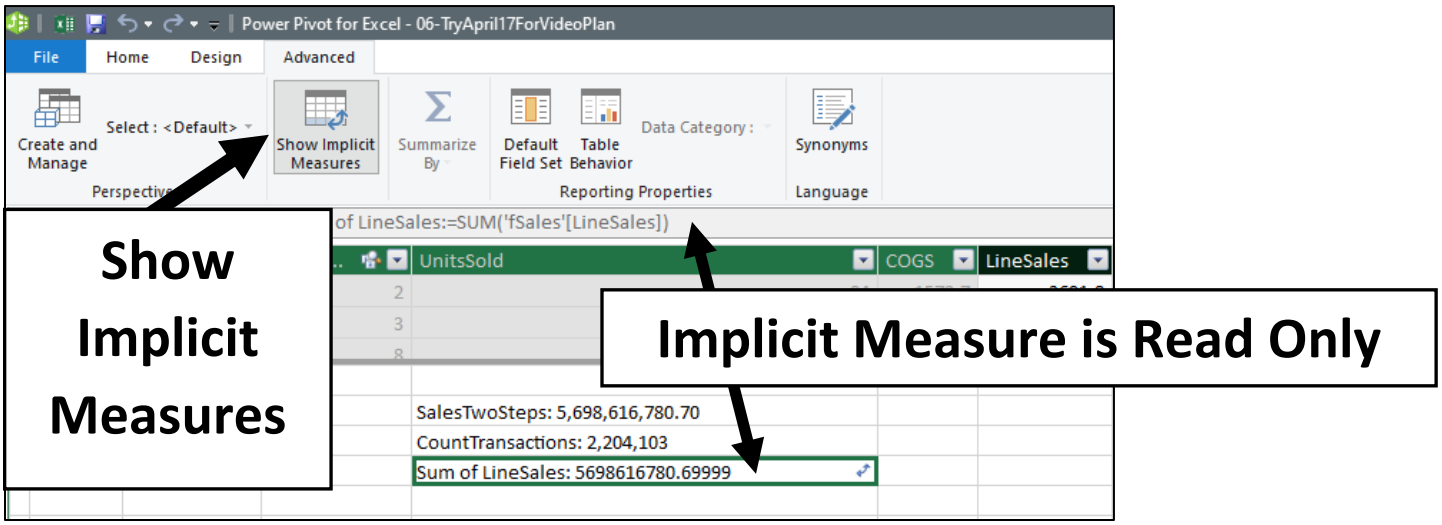
When you drag a field from a table into the Values area of a Data Model PivotTable or a visual in Power BI, a read-only measure is created and stored in a hidden behind-the-scenes location. Microsoft calls this hidden measure an **Implicit Measure**. When you author your own measure, as you did with the Total Sales ($) measure, Microsoft calls that an **Explicit Measure**. Implicit measures can sometimes get the right calculation results, but they do so in an inefficient way. Implicit Measures have several drawbacks, including:

- By default, implicit measures are hidden in the Data Model.
- The measures do not appear in the PivotTable Fields task pane or Power BI Data task pane, and therefore you cannot reuse them in other reports or visuals.
- You cannot edit the name or the formula part of an implicit measure.
- You cannot attach number formatting to an implicit measure. (However, you can manually add number formatting in the report or chart.)
- When you create multiple implicit measures in a data model, duplicate or unnecessary implicit measures may be created.
- This issue has been fixed, but it used to be a problem: If you upload finished Excel or Power BI Desktop file to Power BI Online, Implicit Measures will not show up in the Data Model or any reports or visuals that you created. Now, the implicit Measures seem to show up when you upload an Excel file.

When you create an **Explicit Measure**, you have complete control of the formula in the measure: You can edit the name, edit the formula, add number formatting to the formula, use the formula over and over, and share your data model online.

It is important to know how to look for and delete implicit measures because you may at times accidentally drag fields to the Values area of the PivotTable Fields task pane or inherent a Data Model PivotTable that contains implicit measures. To show implicit Measures in Excel Power Pivot you can use the Show Implicit Measures button in the Advanced tab in the Power Pivot for Excel ribbon, as shown below. In Power BI Desktop, I do not know a way to show Implicit Measures.





Case Where Implicit Measures Might Be OK

If you have some Big Data in a Flat Table (all attribute fields in Fact Table rather than a Dimension Table) that will not fit into worksheet, and your calculations are simple, such as adding, counting or percentage calculations, then it can be fast and easy to use implicit Measures. After loading the table to the Data Model, you can drag and drop raw data fields into Values Area of PivotTable to create implicit Measures. As a second option for this scenario, if you load the Big Data Flat

Table directly to the PivotTable cache and use the Standard PivotTable Calculations (this avoids creating Implicit Measures)

SUMX and Iterator Functions

In the previous example, we used a two-step process to get total sales by building two formulas:

- DAX calculated column: Line Sales =RELATED(dProduct[RetailPrice])*fSales[UnitsSold]
- DAX measure: SalesTwoStep:=SUM(fSales[LineSales])

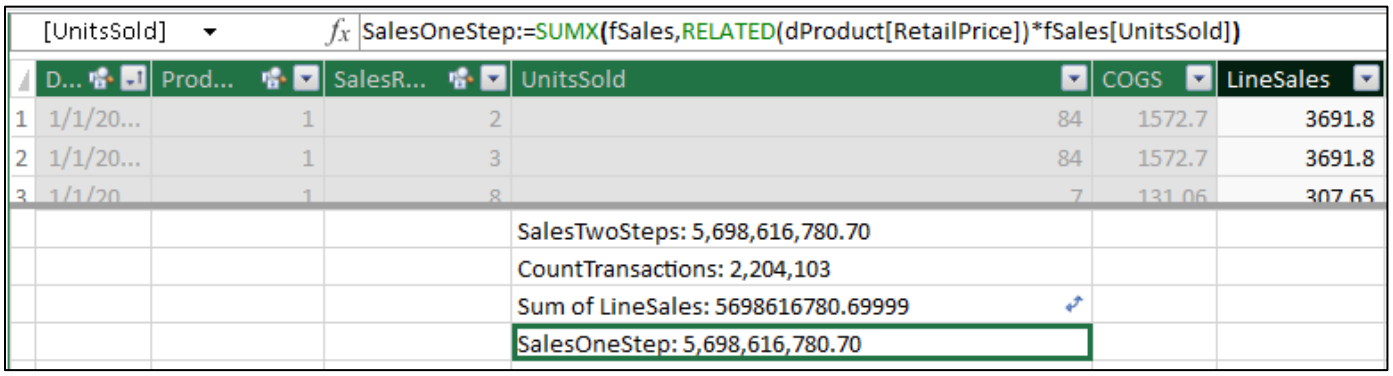
The goal was to calculate the transactional sales amounts for each row in the fact table and then aggregate those numbers into a total. But with the DAX formula language, you can skip over this two-step process and create a single DAX measure that combines the calculated column and the aggregating measure into one DAX measure. You can do this by using one of the amazing DAX X iterator functions: SUMX, AVERAGEX, COUNTX, COUNTAX, MINX, MAXX, CONCATENATEX, PRODUCTX, or RANKX. (There are other types of DAX iterator functions, like FILTER, ADDCOLUMNS and SELECTCOLUMNS too)

To make a calculation in each row of a fact table and then sum those results, you can use the SUMX iterator function, which has the following syntax:

=SUMX(table, expression)

This function simulates a calculated column inside a measure, iterating over each row in the specified table to create the values and then uses the values generated to make the aggregate calculation.

You can often simply take the formula you would have used in a calculated column and place it into an X iterator function to create a one-step solution, like this:



[UnitsSold] fx SalesOneStep:=SUMX(fSales,RELATED(dProduct[RetailPrice])*fSales[UnitsSold])

	D...	Prod...	SalesR...	UnitsSold	COGS	LineSales
1	1/1/20...	1	2	84	1572.7	3691.8
2	1/1/20...	1	3	84	1572.7	3691.8
3	1/1/20	1	8	7	131.06	307.65

SalesTwoSteps: 5,698,616,780.70

CountTransactions: 2,204,103

Sum of LineSales: 5698616780.69999

SalesOneStep: 5,698,616,780.70

In the first argument (the table argument), you place the table where you want to make a row-by-row calculation— in this case, fSales. In the second argument (the expression argument), you place the formula that you want to iterate down the table to make a calculation in each row—in this case, RELATED(dProduct[RetailPrice])*fSales[UnitsSold].

Row Context in Iterator functions

The amazing thing about SUMX and the other X iterator functions is that they automatically create row context so that the formula can use the values from each row in the first argument table and so the RELATED function can look up the price for each row in the table. This single SUMX formula is a more compact solution than the Data Model two-step process with a calculated column and a measure, and it is more efficient than the worksheet formula and standard PivotTable method, where you have to create the Excel Table calculated column using the XLOOKUP function and then do a standard PivotTable calculation.

Two Step and One Step Measures

- A Two Step Measure is created when you use a Measure to aggregate the results from a Calculated Column
- A One Step Measure is created when you use an X Iterator function to create the intermediate values that would have been created in a Calculates Column before aggregating with functions like SUMX, COUNTX and AVERAGEX.
- Both the one step and two step methods will yield the same result, as shown here:

Products	SaleOneStep	SalesTwoStep
Aspen	253,016,877.15	253,016,877.15
Beaut	347,103,792.90	347,103,792.90
Bellen	253,361,736.05	253,361,736.05
Carlota	253,820,067.55	253,820,067.55
Yanaki	848,229,570.30	848,229,570.30
Total	**1,955,532,043.95**	**1,955,532,043.95**

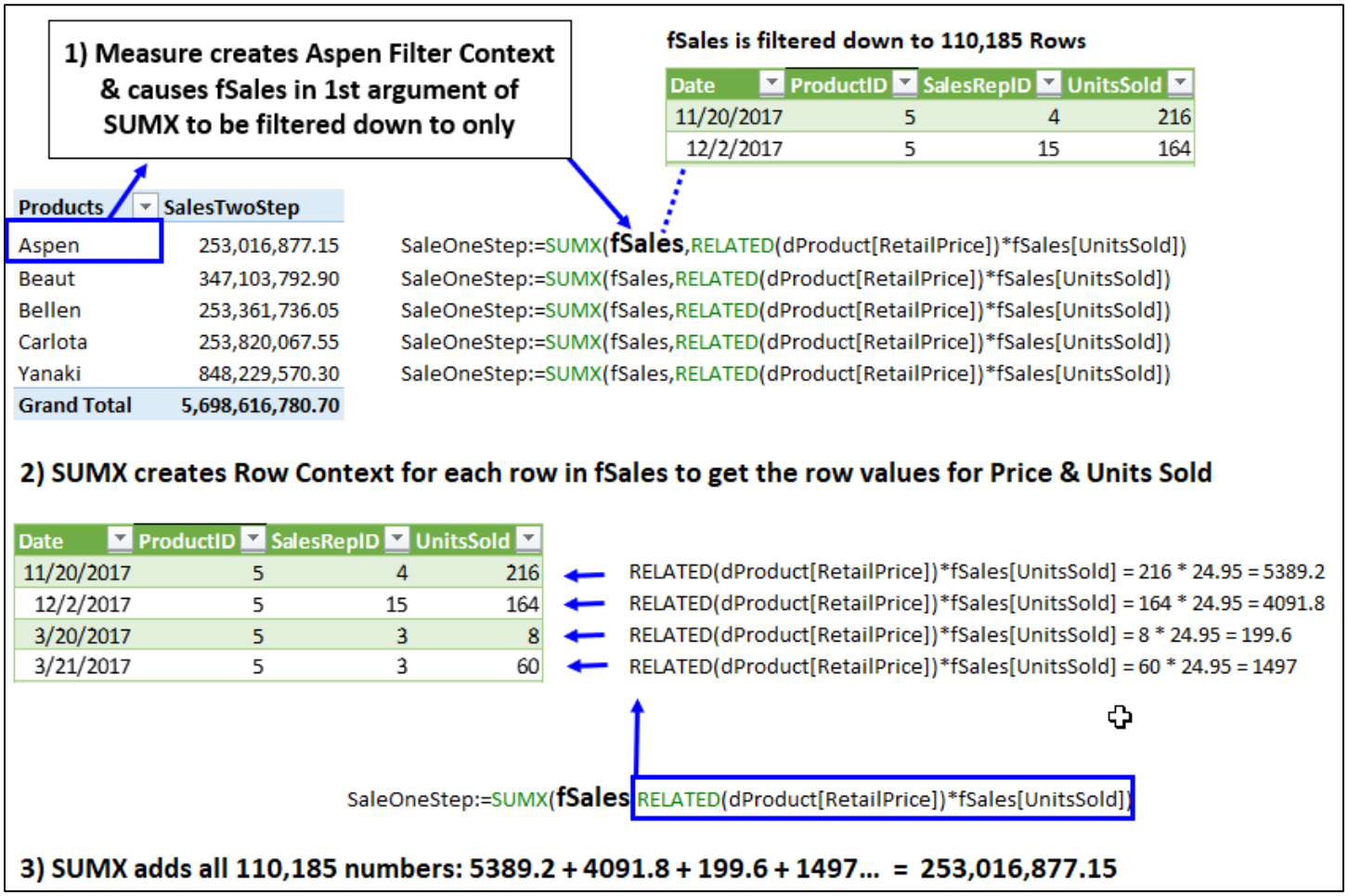
Characteristics of two methods:

- **Two-step method:** With the two-step method, the values generated by the calculated column are stored in the columnar database & become part of what is stored in RAM. This increases file size. When you refresh the table, either in the Power Pivot for Excel window or in Power Query, the calculated column values are recalculated.
    1. DAX calculated column: Line Sales =RELATED(dProduct[RetailPrice])*fSales[UnitsSold]
    2. DAX measure: SalesTwoStep:=SUM(fSales[LineSales])
- **One-step method:** With the one-step method, the values that are generated inside the SUMX function are not stored in RAM. The values in the SUMX function are recalculated each time you drop the measure into the report/visual or when a condition is changed in the Rows, Columns, or Filters area of the report/visual.
    1. SalesOneStep:=SUMX(fSales,RELATED(dProduct[RetailPrice])*fSales[UnitsSold])

When building DAX measures in the Data Model. the convention is to use the one-step method because the measure can be created more quickly, and the in-RAM database does not have to store as much data. As the great DAX formula masters Marco Russo and Alberto Ferrari say, for most models under 100 million rows of data with simple calculations, either method will work fine, and so it becomes a matter of preference whether to use the two-step method or the one-step method. However, my rule of thumb is that if an X iterator Measure calculates slowly every time you drop it into a report/visual, it might be better to move the calculation back to a calculated column (keeping in mind that other issues might cause slowness such as unnecessary content transition over a fact table).

Understanding How Filter Context and Row Context Work Together

When you use an X iterator function such as SUMX in a measure and then drop it into a report or a visual, filter context and row context work together to make the final calculation. As shown in the figure below, when the SUMX measure is in the Aspen cell, the Aspen filter context filters the fUnits table in the first argument of the SUMX function down to 110,185 rows. Then, the SUMX function's row context allows the formula in the second argument to make the calculation row by row, pulling out the correct units sold, looking up the correct product price, and finally multiplying the amounts to get the transaction sales amount for each row. Once SUMX has created all the transactional sales amounts for Aspen, then it adds to get a total of 253,016,877.15 and delivers the result to the Apsen cell in the PivotTable. The Measure does uses this process for each row in the report to get the correct sales amount for each product.



Gross Profit

Gross profit is a metric that assesses how well a company can manage the variable production and labor costs that go into producing a product or service. The formula for the gross profit calculation is:

Gross Profit = Total Sales – Total COGS

Gross profit tells you how much of the total sales is left over after subtracting all the variable production costs, which can then be used to cover fixed costs (such as rent, utilities, and administrative costs) and profit for the company. For a boomerang manufacturing company, total sales would be the revenue brought in from selling the finished boomerang products, and COGS (cost of goods sold) would be the variable costs incurred from producing the boomerangs, such as wood, paint, labor to make the boomerangs, packaging, and other costs that went into producing the boomerangs. The formula for the percentage of gross profit calculation is:

% Gross Profit = Total Gross Profit/Total Sales

The percentage of gross profit expresses the number of pennies for every one dollar of sales that can be used to cover fixed costs and profit. For a manufacturer, this is an important metric that indicates the health of the company. If the percentage of gross profit is increasing over time, it can indicate that the company is managing variable product costs well and that profit may be going up. If this metric goes down over time, it may indicate that the costs of production are increasing and that profits may be lower.

Using Measures in Other DAX Formulas

The conventions for referring to Measures or table columns in formulas is as follows:

- When you use Measures in other DAX Formulas you type square brackets around the Measure name, like: **[MeasureName]**
- When you use a column reference in DAX Formulas you type table name and then in square brackets you type the column name, like: **TableName[ColumnName]**

I created these two Measures:

- TotalSales:=SUMX(fSales,RELATED(dProduct[RetailPrice])*fSales[UnitsSold])
- TotalCOGS:=SUM(fSales[COGS])

To create a Measure for Gross Profit, we can use the two measures in a new Measure, as shown below:

Excel Power Pivot:

[UnitsSold] fx GrossProfit:=[TotalSales]-[TotalCOGS]

	Date	ProductID	SalesRepID	UnitsSold	LineSales	COGS
1	11/10/2017	1	1	84	3691.8	1572.7
2	10/27/2017	1	1	84	3691.8	1572.7
3	5/22/2017	1	1	84	3691.8	1572.7

TotalSales: 5,698,616,780.70

TotalCOGS: 2,396,102,293.04

GrossProfit: 3,302,514,487.66

Power BI Desktop:

```
1 GrossProfit = [TotalSales]-[TotalCOGS]
```

Products	TotalSales	TotalCOGS	GrossProfit
Aspen	253,016,877.15	94524809	158,492,068.60

Here is what the 3 Measures look like a PivotTable and a Matrix:

Excel Power Pivot PivotTable:

Products	TotalSales	TotalCOGS	GrossProfit
Aspen	253,016,877.15	94,524,808.55	158,492,068.60
Beaut	347,103,792.90	129,753,288.23	217,350,504.67
Bellen	253,361,736.05	107,342,954.14	146,018,781.91
Carlota	253,820,067.55	103,613,424.02	150,206,643.53
Yanaki	848,229,570.30	328,821,091.92	519,408,478.38
Grand Total	**1,955,532,043.95**	**764,055,566.86**	**1,191,476,477.09**

Power BI Desktop Matrix Visual:

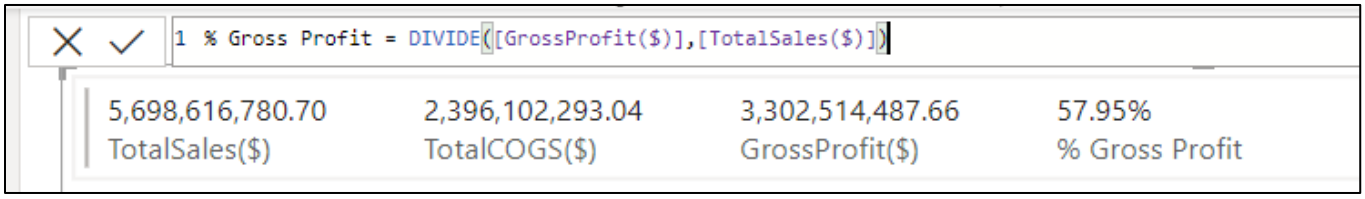
Products	TotalSales	TotalCOGS	GrossProfit
Aspen	253,016,877.15	94,524,808.55	158,492,068.60
Beaut	347,103,792.90	129,753,288.23	217,350,504.67
Bellen	253,361,736.05	107,342,954.14	146,018,781.91
Carlota	253,820,067.55	103,613,424.02	150,206,643.53
Yanaki	848,229,570.30	328,821,091.92	519,408,478.38
Total	**1,955,532,043.95**	**764,055,566.86**	**1,191,476,477.09**

DIVIDE Function & % Gross Profit

In order to create the % Gross Profit measure, you have to perform division. In the DAX formula language, there is a built-in function to do this: DIVIDE. The DIVIDE function, which delivers the quotient of two numbers and allows you to specify an alternative value when the denominator is zero, has the following syntax:

DIVIDE(Numerator, Denominator, [AlternativeResult])

If you omit the third argument, [AlternativeResult], you get a DAX blank value, which is neither an empty cell, as you might get in the Excel worksheet, nor a null value, as you might get in Power Query, but instead it will show nothing in the report. The picture below shows how to use the DIVIDE function to calculate % gross Profit in Power BI Desktop:



CALCULATE Function and Filter Context

- The CALCULATE DAX function allows you to change the external filter context (conditions from the Rows area, Columns area, Filters area, and slicers) for a measure by specifying one or more new internal filters (logical tests and conditions) inside in the function.
- CALCULATE can also convert the row context in a calculated column or a DAX iterator function into filter context with a process called context transition.
- The CALCULATE function arguments are:

  CALCULATE(Expression, Filter1, Filter2…)

- The Expression argument, which is a required argument, contains the scalar formula for which you want to change the filter context.
- The Filter arguments allow you to specify one or more new internal filters by using:
  - A filter modifier function like: ALL, ALLEXCEPT, or ALLSELECTED.
  - A Boolean (True/False) formula like: dProduct[Product]="Quad".
  - A DAX table function that defines a valid list of values as a filter like: DATESINPERIOD or the VALUES DAX function.
- When you enter two or more internal filters into the Filter arguments, the filters are run as an AND logical test.
- If a field is used in both the external and internal filters, the external filter is removed and replaced with the internal filter.
  - For example, if a measure is in the Aspen product row in a report, and the filter inside CALCULATE is dProduct[Product]="Quad", the internal filter dProduct[Product]="Quad" would replace the external filter dProduct[Product]="Aspen", and the measure would calculate an amount for the Quad product.
- The Filter arguments are not required. If you omit these arguments, CALCULATE will perform context transition without an internal filter. If you use the Filter arguments and row context is available, external filters, internal filters, and the transitioned row filters are all merged in an AND logical test.
- When all external and internal filters are evaluated by the CALCULATE function, CALCULATE creates the final filter context by running an AND logical test with all remaining external and internal filters. The final filter context is used to filter the underlying data model tables so the measure can calculate the formula result.

ALL Function

- The ALL function has two uses:
  - It can be used on a single column, multiple columns, or a full table to remove the filter context and return a table.
    - When it is used on a table, it removes all filters and returns the full table.
    - When used on a column or columns, it returns a unique list of records as a table with a single blank row if there are unmatched items in the relationship. Columns must be from the same table.
  - When you use it in the CALCULATE function as a filter, such as ALL(), it removes all filters in the data model. You can also use the ALL function with a columns, columns or a table to remove filters from specific columns or table.

VALUES Function

- The VALUES function "sees" the current filter context and delivers a unique list as a table for a column, columns from the same table or a full table. If there are unmatched items in the relationship, it returns a single blank row to bottom of the unique list.
- If the table returned is a single item, it is returned as a scalar value. We can use VALUES to bring a variable from a table into a DAX Formula.

ALL and VALUES functions compared

VALUES(Column or Table)	ALL(Column or Columns from Same Table or Table)
"Sees" current filter context and delivers a unique list	Removes filters and delivers a unique list or table
Don't want unmatched blank use: DISTINCT	Don't want unmatched blank use: ALLNONBLANKROW

Example of ALL and VALUES in CONCATENATEX DAX Function

This example can be found in the file named "17-VALUES-ALL.xlsx" in the folder named "ExtraSingleExampleFiles".

Units	Product	SalesRep
72	MB	Sioux
48	Quad	Shihara
60	MTA	Chantel
60	Quad	Shihara
96	FunRang	Bob
132	Quad	Shihara
36	FunRang	Bob
24	FunRang	Shihara
144	Quad	Chantel
108	MTA	Bob
96	MB	Sioux
96	MTA	Shihara
120	Quad	Chantel
132	Quad	Chantel
24	MB	Sioux
120	MB	Shihara

	ALL Table Function	VALUES Table Function
	Removes filters & delivers a unique list	"Sees" current filter context & delivers a unique list
SalesRep	**ProductsSoldALL**	**ProductsSoldVALUES**
Bob	FunRang, MB, MTA, Quad	FunRang, MTA
Chantel	FunRang, MB, MTA, Quad	MTA, Quad
Shihara	FunRang, MB, MTA, Quad	FunRang, MB, MTA, Quad
Sioux	FunRang, MB, MTA, Quad	MB
Grand Total	**FunRang, MB, MTA, Quad**	**FunRang, MB, MTA, Quad**

```
ProductsSoldALL:=
CONCATENATEX(
ALL(fSales[Product]),fSales[Product],
",",
fSales[Product],ASC)
```

```
ProductsSoldVALUES:=
CONCATENATEX(
VALUES(fSales[Product]),fSales[Product],
",",
fSales[Product],ASC)
```

ALLSELECTED Function to get a Filtered Grand Total

- When you use ALLSELECTED() as a filter modifier it removes the row and column filters in a particular PivotTable, PivotChart or Visual, but retains the other filters in the Data Model. In this way, you can use the ALLSELECTED() filter modifier to show a filtered grand total amount, which is useful when you want to compare a filtered number against a filtered grand total.
- However, if you use a measure with ALLSELECTED() in other measures, such as in iterator functions, your formula will remove the row and column conditions from the table being iterated—rather than removing any row or column conditions from a given report or visual.
- Microsoft Help: the ALLSELECTED function gets the context that represents all rows and columns in the query, while keeping explicit filters and contexts other than row and column filters. This function can be used to obtain visual totals in queries.

ALLEXCEPT Function

ALLEXCEPT allows you to remove filters from a table except for filters on specified columns and then returns a table of unique records. The columns to exclude can be any column in the Data Model. You cannot use table expressions or column expressions (formulas) inside the ALLEXCEPT function: only tables and columns.

FILTER and CALCULATETABLE functions

FILTER(Table, Filter)	**CALCULATETABLE**(Table,Filters1, Filter2…)
Iterates Row-By-Row when filtering	Uses Data Model Filtering mechanism when filtering
Filter columns must be from table in 1st argument of FILTER	Fact Table can be filtered by any columns in the Start Schema Data Model
In CALCULATE a Boolean filter is converted to a FILTER & ALL Function formula construction. For example, the Boolean filter dProducts[Product]="Quad" is converted to FILTER(ALL(dProducts[Product]), dProducts[Product]="Quad")	Performs Context Transition if Row Context is available
There is only one Filter argument in the FILTER function. For an AND Logical Test use Double Ampersand, like: &&. You can use Double Pipe to create an OR Logical Test, like ||.	Multiple Filter arguments that work in an AND Logical Test. You can use Double Pipe to create an OR Logical Test, like ||.

- Examples of CALCULATETABLE and the FILTER function (in file named "17-ExtraDAXTablesExample.pbix"):

```
QC = CALCULATETABLE(fSales,dProducts[Product]="Quad")
```

Date	SRID	ProductID	CustomerID	Units	LineSales
1/21/2027 12:00:00 AM	6	2	1	30	1290

```
QF = FILTER(fSales,fSales[ProductID]=2)
```

Date	SRID	ProductID	CustomerID	Units	LineSales
1/21/2027 12:00:00 AM	6	2	1	30	1290

```
QuadAndSiouxF = FILTER(fSales,fSales[ProductID]=2 && fSales[SRID]=1)
```

Date	SRID	ProductID	CustomerID	Units	LineSales
11/17/2024 12:00:00 AM	1	2	3	37	1591

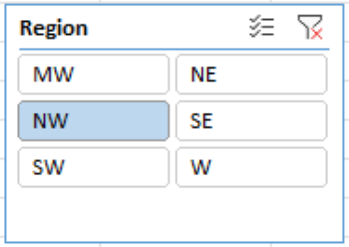
```
QuadOrCarlotaF = FILTER(fSales,fSales[ProductID]=2 || fSales[ProductID] =1)
```

Date	SRID	ProductID	CustomerID	Units	LineSales
1/21/2027 12:00:00 AM	6	2	1	30	1290

Examples of CALCULATE to Change Filter Context for Various % of Total Calculations

Category	Products	TotalSales($)	GrandTotalSales	%OfTotal Sales	FilteredGrand TotalSales	%OfFiltered GrandTotalSales	ParentCategory TotalSalesAE	%ParentCategory TotalSalesAE	ParentCategory TotalSalesASV	%ParentCategory TotalSalesASV
Advanced	Eagle	143,602,055.10	5,698,616,780.70	2.52%	1,248,871,910.45	11.50%	1,090,011,883.95	13.17%	239,510,211.95	59.96%
	Flattop	45,393,738.15	5,698,616,780.70	0.80%	1,248,871,910.45	3.63%	1,090,011,883.95	4.16%	239,510,211.95	18.95%
	Kangaroo	50,514,418.70	5,698,616,780.70	0.89%	1,248,871,910.45	4.04%	1,090,011,883.95	4.63%	239,510,211.95	21.09%
Advanced Total		**239,510,211.95**	**5,698,616,780.70**	**4.20%**	**1,248,871,910.45**	**19.18%**	**1,090,011,883.95**	**21.97%**	**239,510,211.95**	**100.00%**
Beginner	Aspen	54,830,918.40	5,698,616,780.70	0.96%	1,248,871,910.45	4.39%	1,354,608,183.50	4.05%	295,204,213.15	18.57%
	Bellen	55,295,637.10	5,698,616,780.70	0.97%	1,248,871,910.45	4.43%	1,354,608,183.50	4.08%	295,204,213.15	18.73%
	Yanaki	185,077,657.65	5,698,616,780.70	3.25%	1,248,871,910.45	14.82%	1,354,608,183.50	13.66%	295,204,213.15	62.69%
Beginner Total		**295,204,213.15**	**5,698,616,780.70**	**5.18%**	**1,248,871,910.45**	**23.64%**	**1,354,608,183.50**	**21.79%**	**295,204,213.15**	**100.00%**
Competition	NaturalElbow	30,103,775.05	5,698,616,780.70	0.53%	1,248,871,910.45	2.41%	693,391,381.45	4.34%	151,981,953.35	19.81%
	Sunset	57,741,915.90	5,698,616,780.70	1.01%	1,248,871,910.45	4.62%	693,391,381.45	8.33%	151,981,953.35	37.99%
	Sunshine	44,172,930.90	5,698,616,780.70	0.78%	1,248,871,910.45	3.54%	693,391,381.45	6.37%	151,981,953.35	29.06%
	Vrang	19,963,331.50	5,698,616,780.70	0.35%	1,248,871,910.45	1.60%	693,391,381.45	2.88%	151,981,953.35	13.14%
Competition Total		**151,981,953.35**	**5,698,616,780.70**	**2.67%**	**1,248,871,910.45**	**12.17%**	**693,391,381.45**	**21.92%**	**151,981,953.35**	**100.00%**
Freestyle	Carlota	55,119,440.20	5,698,616,780.70	0.97%	1,248,871,910.45	4.41%	1,752,387,854.50	3.15%	383,996,960.35	14.35%
	Quad	316,487,905.50	5,698,616,780.70	5.55%	1,248,871,910.45	25.34%	1,752,387,854.50	18.06%	383,996,960.35	82.42%
	TriFly	12,389,614.65	5,698,616,780.70	0.22%	1,248,871,910.45	0.99%	1,752,387,854.50	0.71%	383,996,960.35	3.23%
Freestyle Total		**383,996,960.35**	**5,698,616,780.70**	**6.74%**	**1,248,871,910.45**	**30.75%**	**1,752,387,854.50**	**21.91%**	**383,996,960.35**	**100.00%**
Long Distance	Beaut	76,828,205.75	5,698,616,780.70	1.35%	1,248,871,910.45	6.15%	808,217,477.30	9.51%	178,178,571.65	43.12%
	Elevate	71,127,614.25	5,698,616,780.70	1.25%	1,248,871,910.45	5.70%	808,217,477.30	8.80%	178,178,571.65	39.92%
	LongRang	30,222,751.65	5,698,616,780.70	0.53%	1,248,871,910.45	2.42%	808,217,477.30	3.74%	178,178,571.65	16.96%
Long Distance Total		**178,178,571.65**	**5,698,616,780.70**	**3.13%**	**1,248,871,910.45**	**14.27%**	**808,217,477.30**	**22.05%**	**178,178,571.65**	**100.00%**
Grand Total		**1,248,871,910.45**	**5,698,616,780.70**	**21.92%**	**1,248,871,910.45**	**100.00%**	**5,698,616,780.70**	**21.92%**	**1,248,871,910.45**	**100.00%**

```
TotalSales($):=SUMX(fSales,RELATED(dProduct[RetailPrice])*fSales[UnitsSold])
GrandTotalSales:=CALCULATE([TotalSales($)],ALL())
%OfTotalSales:=DIVIDE([TotalSales($)],[GrandTotalSales])
FilteredGrandTotalSales:=CALCULATE([TotalSales($)],ALLSELECTED())
%OfFilteredGrandTotalSales:=DIVIDE([TotalSales($)],[FilteredGrandTotalSales])
ParentCategoryTotalSalesAE:=CALCULATE([TotalSales($)],ALLEXCEPT(fSales,dProduct[Category]))
%ParentCategoryTotalSalesAE:=DIVIDE([TotalSales($)],[ParentCategoryTotalSalesAE])
ParentCategoryTotalSalesASV:=CALCULATE([TotalSales($)],ALLSELECTED(),VALUES(dProduct[Category]))
%ParentCategoryTotalSalesASV:=DIVIDE([TotalSales($)],[ParentCategoryTotalSalesASV])
```



Time Intelligence Functions such as SAMEPERIODLASTYEAR

Time Intelligence functions can be used with a date table to change the filter context. Below is a list of some of these functions. The Data Table must have all days for all years that span the minimum and maximum years from the dates in the Fact Table in order for these functions to make correct data calculations.

Function	Description
DATEADD	Returns a table that contains a column of dates, shifted either forward or backward in time by the specified number of intervals from the dates in the current context.
DATESBETWEEN	Returns a table that contains a column of dates that begins with a specified start date and continues until a specified end date.
DATESINPERIOD	Returns a table that contains a column of dates that begins with a specified start date and continues for the specified number and type of date intervals.
DATESMTD	Returns a table that contains a column of the dates for the month to date, in the current context.
DATESQTD	Returns a table that contains a column of the dates for the quarter to date, in the current context.
DATESYTD	Returns a table that contains a column of the dates for the year to date, in the current context.
ENDOFMONTH	Returns the last date of the month in the current context for the specified column of dates.
ENDOFQUARTER	Returns the last date of the quarter in the current context for the specified column of dates.
ENDOFYEAR	Returns the last date of the year in the current context for the specified column of dates.
FIRSTDATE	Returns the first date in the current context for the specified column of dates.
FIRSTNONBLANK	Returns the first value in the column, column, filtered by the current context, where the expression is not blank.
LASTDATE	Returns the last date in the current context for the specified column of dates.
LASTNONBLANK	Returns the last value in the column, column, filtered by the current context, where the expression is not blank.
NEXTDAY	Returns a table that contains a column of all dates from the next day, based on the first date specified in the dates column in the current context.
NEXTMONTH	Returns a table that contains a column of all dates from the next month, based on the first date in the dates column in the current context.
NEXTQUARTER	Returns a table that contains a column of all dates in the next quarter, based on the first date specified in the dates column, in the current context.
NEXTYEAR	Returns a table that contains a column of all dates in the next year, based on the first date in the dates column, in the current context.
PARALLELPERIOD	Returns a table that contains a column of dates that represents a period parallel to the dates in the specified dates column, in the current context, with the dates shifted a number of intervals either forward in time or back in time.
PREVIOUSDAY	Returns a table that contains a column of all dates representing the day that is previous to the first date in the dates column, in the current context.
PREVIOUSMONTH	Returns a table that contains a column of all dates from the previous month, based on the first date in the dates column, in the current context.
PREVIOUSQUARTER	Returns a table that contains a column of all dates from the previous quarter, based on the first date in the dates column, in the current context.
PREVIOUSYEAR	Returns a table that contains a column of all dates from the previous year, given the last date in the dates column, in the current context.
SAMEPERIODLASTYEAR	Returns a table that contains a column of dates shifted one year back in time from the dates in the specified dates column, in the current context.
STARTOFMONTH	Returns the first date of the month in the current context for the specified column of dates.
STARTOFQUARTER	Returns the first date of the quarter in the current context for the specified column of dates.
STARTOFYEAR	Returns the first date of the year in the current context for the specified column of dates.
TOTALMTD	Evaluates the value of the expression for the month to date, in the current context.
TOTALQTD	Evaluates the value of the expression for the dates in the quarter to date, in the current context.
TOTALYTD	Evaluates the year-to-date value of the expression in the current context.

IF function

The IF function is the same as in the Excel worksheet, except that if the third argument is omitted, a DAX Blank is used.

HASONEVALUE function

HASONEVALUE is a Boolean DAX function that returns TRUE when a field in the current filter context contains only one value and FALSE when it contains more than one value. For example, if you use the HASONRVALUE function is a Year/Month Report, in the January 2018 row, the Year field contains only the 2018 value; in the January 2019 row, the Year field contains only the 2019 value; and in the 2019 total row, the Year field contains only the 2019 value. It is only in the grand total cell where the field contains more than one year value: It contains all four years. This function can be used to prevent a formula from executing in the grand total row.

VAR & RETURN to define Variables in DAX Formula

- You can define a variable in any DAX expression by using VAR followed by RETURN. In one or several VAR sections, you individually declare the variables needed to compute the expression; in the RETURN part you provide the expression itself.
- Visual of VAR & RETURN:

Syntax for VAR & RETURN Variables in DAX:

```
VAR VariableName1 = Formula1
VAR VariableName2 = Formula2
VAR VariableName3 = Formula3
RETURN
Formula that uses DAX Formulas and VAR Variables
```

Example of YOY % Change DAX Formula:

```
YOY%Change:=
VAR LastYear = CALCULATE([TotalSales($)],SAMEPERIODLASTYEAR(dDate[Date]))
VAR ThisYear = [TotalSales($)]
RETURN
IF(HASONEVALUE(dDate[Year]),DIVIDE(ThisYear -LastYear,LastYear))
```

YOY % Change DAX Formula and Report from above formula

	A	B	C	D	E	F	G	H
1								
2		Year	Month	TotalSales($)	YOY%Change			
3		2018	Jan	20,034,332.75	-31.65%			
4			Feb	18,293,496.50	-30.48%			
5			Mar	49,971,855.30	-33.13%			
6			Apr	105,595,099.55	-31.70%			
7			May	108,295,145.60	-31.15%			
8			Jun	47,543,848.20	-33.79%			
9			Jul	19,886,442.25	-31.56%			
10			Aug	19,816,735.75	-31.49%			
11			Sep	19,327,985.90	-31.59%			
12			Oct	114,375,373.15	-33.52%			
13			Nov	209,606,879.20	-32.45%			
14			Dec	199,461,489.40	-31.93%			
15		2018 Total		932,208,683.55	-32.23%		Worksheet Formula:	
16		2019	Jan	6,608,339.05	-67.01%		-67.01%	=D16/D3-1
17			Feb	5,822,378.75	-68.17%			
18			Mar	16,864,207.50	-66.25%			

YOY % Change Formula For Partial Year Data:

1. Helper Column in Date Table that asks the question "Is date in date table less than or equal to the last sales date in the fact table that is pushed 12 months back?"

```
ValidDatesForPartialYearSaleYOYCalc = dDate[Date]<=EDATE(MAX(fSales[Date]),-12)
```

Date	MonthNumber	Month	Year	EOMonth	ValidDatesForPartialYearSaleYOYCalc
3/14/2023	3	Mar	2023	3/31/2023	True
3/15/2023	3	Mar	2023	3/31/2023	True
3/16/2023	3	Mar	2023	3/31/2023	False

2. CALCULATE uses the Helper Column from the Date Table as a filter to filter out dates after the last sales date in the Fact Table. This prevents the formula from calculating after the last sales date in the fact table. This is helpful for reports and visuals so that the amount does not show after the last sales date in the Fact Table. For amounts in the first year, because the last year Measure delivers a blank, the DIVIDE function divides by zero and is thus triggered to show a blank in the report or visual.

```
YOY%Change =
VAR LastYear =
CALCULATE([TotalSales($)],
    SAMEPERIODLASTYEAR(dDate[Date]),
    dDate[ValidDatesForPartialYearSaleYOYCalc])
VAR ThisYear = [TotalSales($)]
RETURN
IF(HASONEFILTER(dDate[Year]),DIVIDE(ThisYear - LastYear,LastYear))
```

Boolean Filters in CALCULATE

1. "Boolean Logical Test Filter", or just "Boolean Filter" means that you use a single column, a comparative operator and a condition, like: **dProduct[Product]="Quad"** in the two below examples:

Internal Filter Context = Product "Quad" →	QuadSalesB:= CALCULATE([TotalSales($)], dProduct[Products]="Quad")

External Filter Context = SalesRep = "Alysha Dewitt" →

SalesRep	QuadSalesB
Alysha Dewitt	44,507,461.80
Chantel Zoya	163,281,809.40
Chin Pham	164,438,397.60
Grand Total	**1,452,192,120.30**

Step 1: External and Internal filters are merged in an AND Logical Test:

External Filter Context:	**AND**	**Internal Filter Context:**
dSalesReps[SalesRep] = "Alysha Dewitt"		dProduct[Products]="Quad")

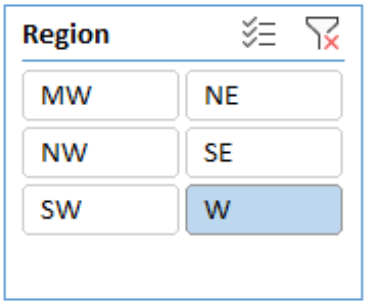
Step 2: Fact Table is filtered down to just rows for SalesRep "Alysha Dewitt" AND Product "Quad", and then the formula calculates total sales to get **44,507,461.80**

Internal Filter Context → "Quad"

```
QuadSalesB:=
CALCULATE( [TotalSales($)] ,
dProduct[Products] = "Quad" )
```

External Filter → Context "Aspen"

Products	TotalSales($)	QuadSalesB
Aspen	32,095,530.30	185,835,235.35
Beaut	44,354,858.35	185,835,235.35
Bellen	32,080,310.80	185,835,235.35
Quad	185,835,235.35	185,835,235.35
Yanaki	109,831,843.95	185,835,235.35
Grand Total	**404,197,778.75**	**185,835,235.35**



2. Why Same "Quad" Sales Number in all cells?
3. Because the CALCULATE functions uses the **overwrite operation** to merge the **external filter** (products in row area of report and "W" (West) from the slicer) with the **internal filter** (Quad) into the **final filter context** that filters the underlying fSales table.

<u>Overwrite Operation in CALCULATE function</u>

4. Example of Overwrite Operation for Aspen cell:

Step1: List all filters:

External Filter Context	Internal Filter Context
dProduct[Products] = "Aspen"	dProduct[Products] = "Quad"
AND	
dSalesReps[Region]="West"	

Step2: Remove column filters from External that are also in Internal

External Filter Context	Internal Filter Context
~~dProduct[Products] = "Aspen"~~	dProduct[Products] = "Quad"
AND	
dSalesReps[Region]="West"	

Step3: All remaining filters are run in an AND Logical test to create final filter context

dSalesReps[Region]="West" **AND** dProduct[Products] = "Quad"

<u>Boolean Filter Behinds Scenes Runs as a FILTER and ALL Function Construction</u>

5. When you use a Boolean Filter, behind the scenes it uses the ALL and FILTER functions like this:

```
QuadSalesFA:=
CALCULATE([TotalSales($)],
FILTER(
        ALL(dProduct[Products]),
        dProduct[Products]="Quad"))
```

How it calculates:

1) ALL removes all filters from dProduct[Products] & delivers a 1 column table that lists all product names.
2) FILTER filters the ALL table & delivers a table with just "Quad".
3) The "Quad" table is used by CALCULATE to filter the underlying fSales table.

KEEPFILTERS Function to Perform AND Logical Test Rather Then An Overwrite Operation in CALCULATE

6. If you do not want the Boolean Filter result to have the same amount show up in every cell, you can use the KEEPFILTERS DAX Function around the Boolean Filter as shown below. The KEEPFILTER function prevents the CALCULATE function from using the Overwrite Operation and instead it forces the CALCULATE to run an AND Logical Test. Said a different way: ***KEEPFILTERS merges External Filter Context with the Internal Filter Context with an AND Logical Test rather than with the Overwrite Operation.***

```
QuadSalesKF:=
CALCULATE([TotalSales($)],
KEEPFILTERS(dProduct[Products]="Quad"))
```

Products	TotalSales($)	QuadSalesKF
Aspen	25,694,664.85	
Beaut	349,439,787.95	
Bellen	255,131,689.05	
Quad	1,452,192,120.30	1,452,192,120.30
Yanaki	854,526,623.25	
Grand Total	**3,165,984,885.40**	**1,452,192,120.30**

Or with no Total Sales Measure:

Products	QuadSalesKF
Quad	1,452,192,120.30
Grand Total	**1,452,192,120.30**

Internal Filter		External Filter
Product="Quad"	AND	Product="Aspen"
TRUE	AND	FALSE

FALSE

Final Filter Context =

Empty Filter = No Records in Fact Table

Internal Filter		External Filter
Product="Quad"	AND	Product="Quad"
TRUE	AND	TRUE

TRUE

Final Filter Context =

Product="Quad"

FILTER and VALUES Functions to SIMULATE KEEPFILTER Result

Below formula delivers the same result as KEEPFILTERS because VALUES in the first argument of FILTER can see the External Filter Context.

```
QuadSalesFV:=
CALCULATE([TotalSales($)],
FILTER(
   VALUES(dProduct[Products]),
   dProduct[Products]="Quad"))
```

Products	QuadSalesFV
Quad	1,452,192,120.30
Grand Total	**1,452,192,120.30**

Here are the steps for how this formula calculates its results and shows only the Quad row in the report:

1. In "Aspen" cell, VALUES function can "see" external filter context and so Aspen row condition flows into VALUES.
2. VALUES Delivers a one row table for the Aspen product to the first argument of FILTER.
3. From the internal filter context, FILTER applies the condition "Quad" to the Aspen row by asking: "Aspen" = "Quad"? The FALSE answer causes FILTER to deliver a blank as the condition in the filter argument of CALCULATE.
4. CALCULATE filters the underlying fact table down to no rows and the Measure delivers a blank.
5. The blank causes the report to show no row for Aspen.
6. The "Quad" cell in the report is the only row where "Quad" = "Quad", so it is the only row that appears in the report.

Logical Operators in DAX:

- AND Logical Test uses: & & (Double Ampersand)
- OR Logical Test uses: | | (Double Vertical Bar)
- List of Conditions for OR Logical Test: IN { “Condition1”, “Condition2”, … “ConditionN”}
- NOT: use NOT Function
- Comparative Operators: =, <, <=, >, >=, <>, = = (strictly equal to)
    - The “strictly equal to” operator == returns TRUE when the two arguments have the same value or are both BLANK. A comparison between BLANK and any other value returns FALSE.

AND Logical Test with Two Filter Arguments in CALCULATE function

You can construct an AND Logical Test in the CALCULATE in these ways:

1. Use two filter arguments in CALCULATE, like:
   - QuadSalesBandNW:=
     CALCULATE([TotalSales($)],
     dProduct[Products]="Quad", dSalesReps[Region]="NW")

     Two Arguments in CALCULATE.

2. Use Double Ampersand when the two columns are the same, like:
   - Both are fSales[LineSales], so this works:
     CountSalesBetween0and500:=
     CALCULATE([CountTransactions],
     fSales[LineSales]>0 && fSales[LineSales]<=500)

     Same Two Columns so this works with &&.

   - The two columns are different, so you get an Error:
     QuadSalesBandNWError:=
     CALCULATE([TotalSales($)],
     dProduct[Products]="Quad" && dSalesReps[Region]="NW")

     Different Columns so you get error with &&.

3. The AND function if the columns are from the same table, like:
   - CountSalesBetween0and500AND:=
     CALCULATE([CountTransactions],
     AND(fSales[LineSales]>0,fSales[LineSales]<=500))

     AND function requires that both columns are from same table.

OR Logical Test in CALCULATE function

Because the filter arguments in the CALCULATE function do not work as an OR Logical Test (they work as an AND Logical Test), you can use the OR function with columns from the same table or you can use Double Vertical Bars. Examples here:

- FreeStyleBoomsSales:=
  CALCULATE([TotalSales($)],
  KEEPFILTERS(
  dProduct[Products]="Quad" || dProduct[Products]="Carlota"))

  Double Vertical Bar for an OR Logical Test: | |.

- FreeStyleBoomsSalesOR:=
  CALCULATE([TotalSales($)],
  KEEPFILTERS(
  OR(dProduct[Products]="Quad", dProduct[Products]="Carlota")))

  OR Function for Logical Test.

KEEPFILTERS to Create Filtered Reports

This KEEPFILTERS and OR Logical Tests Construction is a method of creating a report with only certain items with no need to use a filter or slicer, as shown here:

```
FreeStyleBoomsSales:=
CALCULATE([TotalSales($)],
KEEPFILTERS(
dProduct[Products]="Quad" || dProduct[Products]="Carlota"))
```

Products	FreeStyleBoomsSales
Carlota	255,555,938.85
Quad	1,452,192,120.30
Grand Total	**1,707,748,059.15**

OR Logical Test for List of Items using IN Operator in CALCULATE function

```
AussieRoundBoomSales:=
CALCULATE([TotalSales($)],
KEEPFILTERS(
   dProduct[Products] IN { "Flattop", "Sunshine", "Sunset", "Beaut" }))
```

Products	AussieRoundBoomSales
Beaut	349,439,787.95
Flattop	206,121,057.60
Sunset	264,198,377.25
Sunshine	204,735,059.55
Grand Total	**1,024,494,282.35**

NOT Logical Test in CALCULATE Function to Filter to Items NOT IN List

```
NOTAussieRoundBoomSales:=
CALCULATE([TotalSales($)],
KEEPFILTERS(
NOT(dProduct[Products] IN { "Flattop", "Sunshine", "Sunset", "Beaut" })))
```

Products	NOTAussieRoundBoomSales
Aspen	254,694,664.85
Bellen	255,121,689.05

CALCULATE to perform Context Transition

- CALCULATE and CALCULATETABLE DAX functions can do these two things:
  1) Change the Filter Context.
  2) **Perform Context Transition**, which takes all available Rows Contexts and merges them with an AND Logical Test and then converts them to Filter Context. When you invoke Context Transition on a table, the table must have a unique set of records or a primary key to avoid the double count error.
- Examples of Context Transition:
  1) In a Calculated Column there is no Filter Context, and so an aggregate calculation like SUM can not "see" the ProductID Row Context to calculate the Product Sales for each row, like:

[SUMSales] fx =SUM(fSales[LineSales])

	Prod...	Products	RetailPrice	Category	Supplier	SUMSales
1	1	Quad	43.95	Freestyle	Gel Boom...	5,740,887,136.55
2	2	Yanaki	25.95	Beginner	Colorado B...	5,740,887,136.55

  2) If we put SUM function inside CALCULATE, because CALCULATE is programmed to convert all available Row Context into Filter Context, the ProductID Row Context in each row is converted to Filter Context, then the Fact Table is filtered down to just the rows for that product, and the Calculated Column formula can deliver the correct Total Sales for Each Product, as shown below. Context Transition works because the table being iterated contains a unique set of records.

fx =CALCULATE(SUM(fSales[LineSales]))

Products	RetailPrice	Category	Supplier	SUMSales	CALCULATESUM
Quad	43.95	Freestyle	Gel Boom...	5,740,887,136.55	1,452,192,120.30
Yanaki	25.95	Beginner	Colorado B...	5,740,887,136.55	854,526,623.25

  3) In the DAX Formula language, all Measures have a hidden CALCULATE Function wrapped around it. This means that whenever you use a Measure in a Calculated Column or an Iterator function, it will convert the available Row Context into Filter Context, like:

fx =[TotalSales($)]

RetailPrice	Category	Supplier	SUMSales	CALCULATESUM	HiddenCalculate
43.95	Freestyle	Gel Boom...	5,740,887,136.55	1,452,192,120.30	1,452,192,120.30
25.95	Beginner	Colorado B...	5,740,887,136.55	854,526,623.25	854,526,623.25

  4) Here is a Formula that calculates the % of Total Sales for each product in a Calculated Column using the aggregate SUM function with no Filter Context and the Total Sales Measure with Filter Context:

[%OfTotalPro... fx =[TotalSales($)]/SUM(fSales[LineSales])

	Prod...	Products	RetailPrice	Category	Supplier	%OfTotalProductSales
1	1	Quad	43.95	Freestyle	Gel Boom...	25.30%
2	2	Yanaki	25.95	Beginner	Colorado B...	14.88%

  5) The #1 Problem to watch for when invoking Context Transition is: the double count problem when you invoke context transition over a table with duplicate records. Because many fact tables have duplicate records, this is a common mistake. In the below picture, when the Measure invokes Context Transition, the Row Context is Converted to Filter Context and the Fact Table is filtered for each row being iterated, but for rows with duplicate records, the table is not filtered down to just one row, but instead it is filtered down to all matching rows, which is not correct. For example, in the picture below, the two records are identical and so for each of the records the Filter Context will deliver two rows and thus double count the Line Sales Amount.

7	*	43.95	=	307.65	=307.65+307.65=615.3
7	*	43.95	=	307.65	(Double Counts)

fx =[TotalSales($)]

	SalesR...	UnitsS...	COGS	LineSales	LineSalesMeasure	Add Column
1	3	7	131.06	307.65	615.3	
1	3	7	131.06	307.65	615.3	

7	*	43.95	=	307.65	=307.65+307.65=615.3
7	*	43.95	=	307.65	(Double Counts)

6) The real power of Context Transition can be seen in a Measure like Average Monthly Sales, which we can use in a Product Report to calculate Average Monthly Sales By Product with only a single formula, rather than a multiple step approach and seen in the bottom part of the picture below.

AverageMonthlySalesHC:=AVERAGEX(VALUES(dDate[EOMonth]),[TotalSales($)])

Products	TotalSales($)	AverageMonthlySalesHC
Aspen	254,694,664.85	2,927,524.88
Beaut	349,439,787.95	4,016,549.29
Bellen	255,131,689.05	2,932,548.15
Carlota	255,555,938.85	2,937,424.58
Eagle	658,302,299.55	7,566,693.10
Elevate	324,187,645.15	3,726,294.77
Flattop	206,121,057.60	2,369,207.56
Yanaki	854,526,623.25	9,822,145.09
Grand Total	**5,740,887,136.55**	**65,987,208.47**

Steps in Calculation:

1) Aspen filter flows into [TotalSales($)] Measure and filters Fact Table down to just rows for Aspen.

2) VALUES function delivers a unique list of EOMONTH values (no duplicates) from Date Table to the first argument of AVERAGEX.

3) The Measure iterates the EOMONTH table.

4) For each row in the EOMONTH table the Hidden CALCULATE function wrapped around the Measure performs **Context Transition** and converts the EOMONTH Row Context into Filter Context so that the Fact Table is filtered down to just rows for Aspen in the Given Month.

5) The Iterator function calculates the correct Aspen Sales Amount for Each Month.

6) The Aspen Sales for each month are then used to calculate an Average: Average Monthly Sales By Product "Aspen".

Products	Aspen

EOMonth	TotalSales($)
1/31/2017	1,307,205.35
2/28/2017	1,167,385.55
2/29/2024	664,917.50
3/31/2024	343,336.95
Grand Total	**254,694,664.85**

Hidden Rows

Average:	2,927,524.88

<<== Multiple Step Method:

1) Create End of Month Total Sales Report.
2) Add an Aspen Filter.
3) Create Worksheet Formula.

Context Transition in AVERAGEX involves few steps & is easy to deploy:

AverageMonthlySalesHC:= AVERAGEX(VALUES(dDate[EOMonth]),[TotalSales($)])

When Context Transition Causes Trouble: Context Transition for a Measure Over a Table with Duplicate Records

Looking at Formula #3, when you invoke Context Transition over a table with duplicate records or no primary key, for the rows that are duplicates, the table will be filtered down to show all matching records, and thus the formula will calculate an amount that uses all matching records (a value that is too big) rather than make the calculation based on the single row (shown in picture at bottom of page). In this case, rather than invoke Context Transition with the hidden calculate in a Measure, use a formula rather than a Measure (Formula #2).

	1) Use AVERAGE Function If you have column of transactional sales in fact table.	2) Use AVERAGEX with a formula if you don't have transactional sales field in fact table.	3) Do NOT use AVERAGEX to iterate over a table with duplicate records when the second argument contains a Measure.
	AveTransactionalSaleC:= AVERAGE(fSales[LineSales])	AveTransactionalSaleF:= AVERAGEX(fSales, RELATED(dProduct[RetailPrice])*fSales[UnitsSold])	AveTransactionalSaleM:= AVERAGEX(fSales, [TotalSales($)])
Supplier	**AveTransactionalSaleC**	**AveTransactionalSaleF**	**AveTransactionalSaleM**
Channel Craft	2,086.77	2,086.77	2,602.70
Colorado Boomerangs	2,519.78	2,519.78	3,088.37
Darnell Booms	1,761.69	1,761.69	1,856.58
Gel Boomerangs	3,058.47	3,058.47	3,764.53
Grand Total	**2,586.22**	**2,586.22**	**3,164.08**

Problems with Measure that iterates a table with duplicate records:

1) Context Transiation converts the Row Context to Filter Context, and for the duplicate records, the calculated row amount will be too high (double count).
2) The 3rd formula above unnecessarily iterates the Fact Table twice: once in 1st argument of AVERAGEX and another time inside the Measure.
3) The 3rd formula above does not need context transition to work because the formula was already iterating over the Fact Table.

SalesR...	UnitsSold	COGS	LineSales	LineSalesM
3	7	131.06	307.65	615.3
3	7	131.06	307.65	615.3

This shows a duplicate record in fact table. The LineSales column shows the correct amount of 307.65 for each of the duplicate records. But the LineSalesM column shows that the Context Transition for the duplicate rows double counts the amount because Row Context is converted to Filter Context and for each duplicate record the fact table is filtered down to two rows, and the incorrect double amount of 615.3 is the result for both duplicate records.

7	*	43.95	=	307.65	=307.65+307.65=615.3
7	*	43.95	=	307.65	(Double Counts)

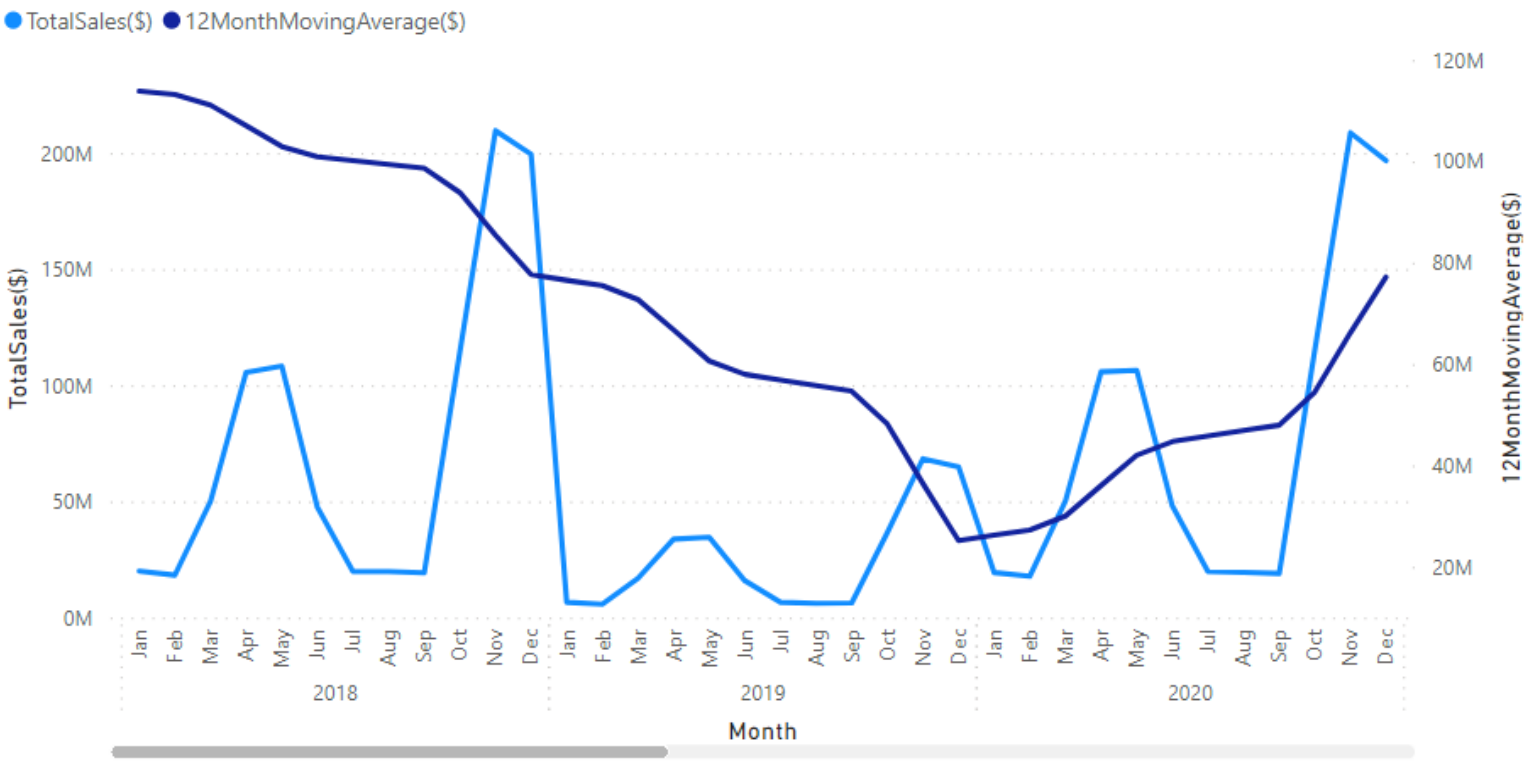
Rule for When to Use Context Transition:

- When you iterate over table with duplicate records no primary key: Use formula.
- When you iterate over table with a unique set of records or has a primary key: Measure that invokes Context Transition won't cause miscalculation.

12 Month Moving Average DAX Formula and Report

When you have volatile sales and you would like a metric to help see the over all trend, you can use a 12-Month Moving Average as shown in the figure below.

```
12MonthMovingAverage($) =
CALCULATE(
    AVERAGEX(VALUES(dDate[EOMonth]),[TotalSales($)]),
    DATESINPERIOD(dDate[Date],MAX(dDate[Date]),-12,MONTH))
```



Moving 12 Month Average Formula For Partial Year Data:

The IF function logical test asks the question "in the current filter context, is there a fact table sales date?" When there is no sales date, this prevents the Measure result from showing up after the last sales date in the fact table.

```
12MonthMovingAverage =
VAR Move12MonthAve =
CALCULATE(
    AVERAGEX(VALUES(dDate[EOMonth]),[TotalSales($)]),
    DATESINPERIOD(dDate[Date],MAX(dDate[Date]),-12,MONTH))
RETURN
IF(MAX(fSales[Date]),Move12MonthAve)
```

Complex Filter

- **Complex Filter** is a filter that involves two or more columns and uses a combination of AND Logical Tests and OR Logical Tests, such as:
    - Year = 2017 **AND** Month = Nov
      OR
      Year = 2017 **AND** Month = Dec
      OR
      Year = 2018 **AND** Month = Jan
      OR
      Year = 2018 **AND** Month = Feb
- **Complex Filter Reduction Error** can happen when:
    - We have a complex filter on two or more columns in a PivotTable or Power BI Visualization.
    - In a Measure, we have an Iterator function that is iterating over one or more of the columns involved in the external complex filter.
    - Context Transition (Row into Filter Context) is occurring in the 2nd argument in the iterator.
    - The Overwrite process in CALCULATE replaces the External Column/s with the Internal Columns/s and leads to the incorrect number of rows in the table being iterated.
    - The KEEPFILTERS function can help to solve this error by instructing CALCULATE to use an AND Logical test rather than the Overwrite Operation. But it is MUCH better to build a data model solution by adding an EOMONTH column in the Data Table and use that column to iterate over.

CROSSJOIN DAX Table Function

- **CROSSJOIN**(Table,Table) = Cartesian product of two or more tables that returns a table, # rows = product of the # of rows from all tables, # columns = the sum of the # of columns in all tables.
- Example of CROSSJOIN in first argument of AVERAGEX function is on next page

A Complex Filter is created because we are using two separate columns (Year & Month) and filtering to create a combination of AND and OR Logical Tests:

2017 AND Nov
OR
2017 AND Dec
OR
2018 AND Jan
OR
2018 AND Feb

```
AverageMonthlySalesHC:=
AVERAGEX(
    VALUES(dDate[EOMonth]),
    [TotalSales($)])
```

```
AverageMonthlySalesCJError:=
AVERAGEX(
    CROSSJOIN(
        VALUES(dDate[Year]),
        VALUES(dDate[Month])),
    [TotalSales($)])
```

Year	Month	TotalSales($)	AverageMonthlySalesHC	AverageMonthlySalesCJError
⊟ 2017	Nov	310,305,980.45	310,305,980.45	310,305,980.45
	Dec	293,014,483.95	293,014,483.95	293,014,483.95
2017 Total		**603,320,464.40**	**301,660,232.20**	**301,660,232.20**
⊟ 2018	Jan	20,034,332.75	20,034,332.75	20,034,332.75
	Feb	18,293,496.50	18,293,496.50	18,293,496.50
2018 Total		**38,327,829.25**	**19,163,914.62**	**19,163,914.62**
Grand Total		**641,648,293.65**	**160,412,073.41**	**138,292,353.04**

Using EOMONTH Helper Column, We Get Correct Average Monthly Sales:

Year	Month	Monthly Sales
2017	Nov	310,305,980.45
2017	Dec	293,014,483.95
2018	Jan	20,034,332.75
2018	Feb	18,293,496.50
Average:		160,412,073.41

Using CROSSJOIN, We Get INCORRECT Ave. Monthly Sales:

Year	Month	Monthly Sales
2017	Nov	29,309,926.50
2017	Dec	26,312,235.60
2017	Jan	310,305,980.45
2017	Feb	293,014,483.95
2018	Nov	20,034,332.75
2018	Dec	18,293,496.50
2018	Jan	209,606,879.20
2018	Feb	199,461,489.40
Average:		138,292,353.04

When you have two columns in an external filter context (created by a hierarchy) that are locked in an AND and OR Logical Test, and both fields are being iterated inside the Measure, the Overwrite Process will use the incorrect eight rows to iterate rather than the correct four

External Complex Filter:

Year	Month
2017	Nov
2017	Dec
2018	Jan
2018	Feb

** 4 Rows to Iterate

Year Column	CROSSJOIN Cartesian Product	Month Column
2017		Nov
2018		Dec
		Jan
		Feb

= **Cartesian Product Process:**

2017	2017	Nov
	2017	Dec
	2017	Jan
	2017	Feb
2018	2018	Nov
	2018	Dec
	2018	Jan
	2018	Feb

=

Year	Month
2017	Nov
2017	Dec
2017	Jan
2017	Feb
2018	Nov
2018	Dec
2018	Jan
2018	Feb

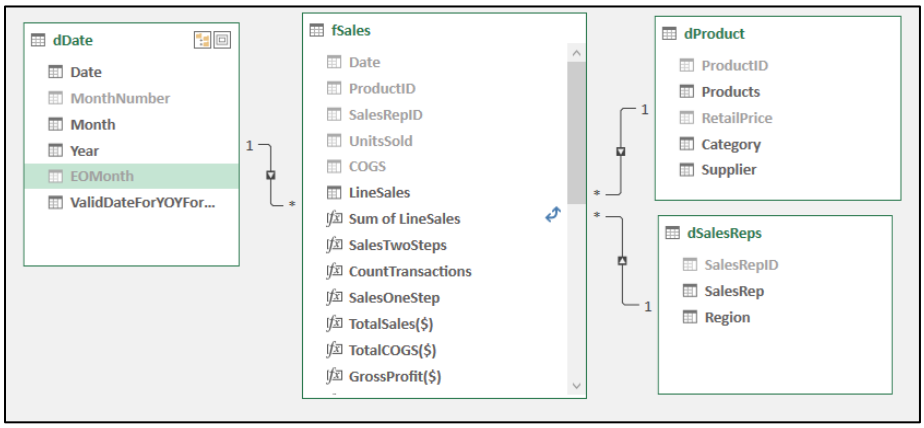
** 8 Rows to Iterate

* This formula would fix it:

```
AverageMonthlySalesKF:=
AVERAGEX(
KEEPFILTERS(CROSSJOIN(VALUES(dDate[Year]),VALUES(dDate[Month]))), [TotalSales($)])
```

***But use EOMONTH Helper Column formula instead!!!*

Star Schema Data Model and Expanded Table Diagram:



Color Coding:

Native Columns are columns in the table
Expanded Columns and columns that flow into a table through a one-to-many relationship

Expanded Table Diagram Key Concept: When there are one-to-many relationships between the Fact Table and the Dimension Table, because filters on Dimension Tables flow to Fact Table, it is as if the Fact Table contains all columns: Native and Expanded.

Fields/Tables	dProduct	dSalesRep	dDate	fSales
ProductID				
Products				
RetailPrice				
Category				
Supplier				
SalesRepID				
SalesRep				
Region				
Date				
MonthNumber				
MonthNumber				
Year				
EOM				
ValidDateForYOYCalc				
Date				
ProductID				
SalesRepID				
UnitsSold				
COGS				
LineSales				

Expanded Table Columns Helps Answer These Questions:

1) Which Tables will a "Column Filter" affect?

ALL(fSales[UnitsSold]) In ALL, this column filter removes filters from fSales only

ALL(dProduct[Product]) In ALL, this column filter removes filters from dProduct and fSales

2) Which Columns will a "Table Filter" affect?

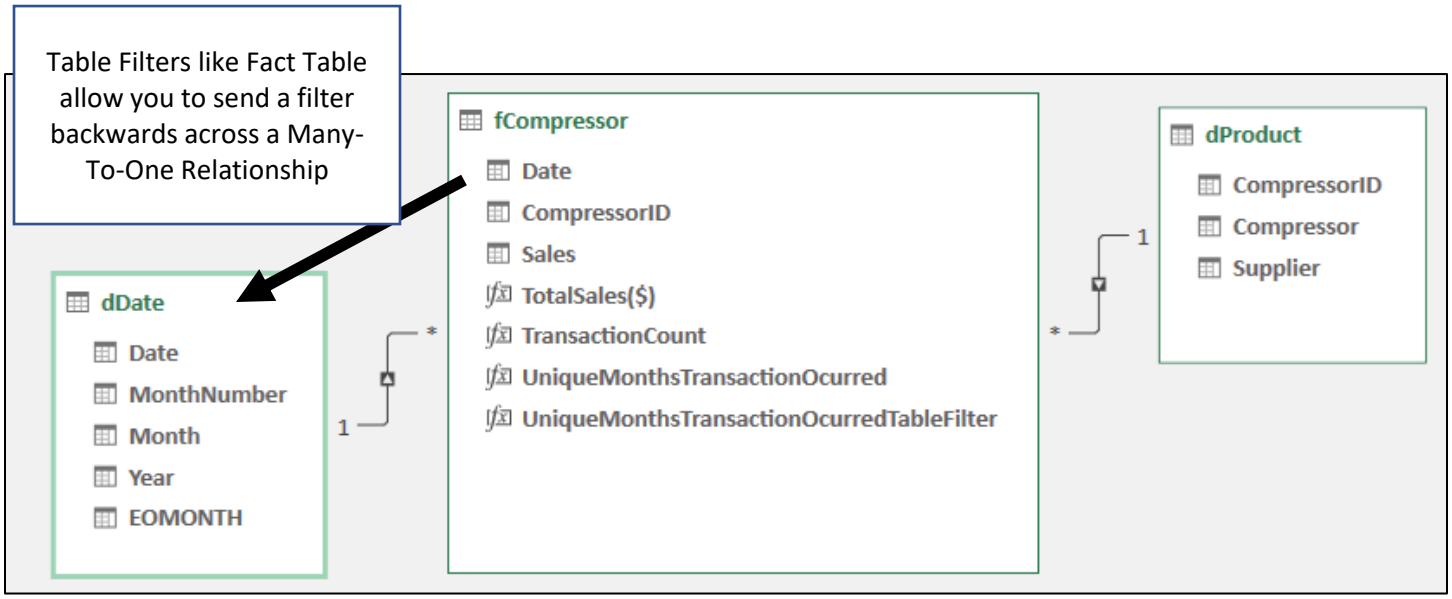
ALL(fSales) removes filters from all tables

3) Which Expanded Table columns are in play with any "Table Filter"?

ALLEXCEPT(fSales,dProduct[Category]) uses a field from dProduct but filters the fSales table.

Table Filter To Go Backwards Across Many-To-One Relationship

This example can be found in the file named “17-TableFilter.xlsx”.



Example of “Unique Months Transactions Occurred” Count for each Supplier:

1. The below COUNTROWS / VALUES formulas (UniqueMonthsTransactionOcurred Measure) is counting filtered rows on the EOMONTH Field. But it returns a 48 count of all possible months because the filter from the dProduct that is filtering the fact table cannot move across the Many-To-One Relationship from the Fact Table (fCompressor) to the Date Table (dDate).
2. But when we add a table filter of fCompressor as a filter in CALCULATE (UniqueMonthsTransactionOcurredTableFilter Measure), because the expanded table has all fields from the data model, including the dDate table, the dDate table is filtered down to just the EOMONTH unique dates that the VALUES function is delivering for the current filter context.

UniqueMonthsTransactionOcurred:= COUNTROWS(VALUES(dDate[EOMONTH]))	UniqueMonths TransactionOcurred TableFilter:= CALCULATE(COUNTROWS(VALUES(dDate[EOMONTH])), fCompressor)

Supplier	Compressor	TotalSales($)	Transaction Count	UniqueMonths TransactionOcurred	UniqueMonths TransactionOcurred TableFilter
⊞ Bay Air Services		225,870.00	41	48	30
⊞ Dawalt		80,340.00	10	48	8
⊞ DeVilbiss		57,215.00	27	48	22
⊞ Ingersol		185,721.00	26	48	18
Grand Total		**549,146.00**	**104**	**48**	**44**

DAX Formula Evaluation Context Summary

i. There are Two Evaluation Contexts:
    1. Row Context = allows a formula in a Calculated Column or an Iterator Function or in a PivotTable/Power BI Visualization to see the row and use the values from the row to make a Row-By-Row Calculation.
    2. Filter Context = all the Filters / Conditions / Criteria that filter the underlying tables in the Data Model to provide the final values for the Measure to use to calculate the final answer.
ii. CALCULATE and CALCULATETABLE DAX functions can do these two things:
    1. Change the Filter Context.
    2. Perform Context Transition, which takes all available Rows Contexts and merges them with an AND Logical Test and then converts them to Filter Context.
iii. All Measures have a hidden CALCULATE function wrapped around it.
iv. There are two types of Filter Contexts that are used to determine the Final Filter Context under which the Measure makes its final calculation:
    1. External Filter Context = Filters / Conditions / Criteria from Excel PivotTables or Power BI Visualizations.
    2. Internal Filter Context = Filters / Conditions / Criteria from inside the CALCULATE function.
v. How Final Filter Context is determined:
    1. Filters / Conditions / Criteria from Excel PivotTables or Power BI Visualizations flow into a Measure.
    2. Inside the Measure the internal and external filters are merged into the Final Filter Context using the operators:
        i. And Logical Test (Intersect)
        ii. Overwrite
        iii. Remove
vi. When the ALL functions is used in a CALCULATE Filter argument, all the filters for the column, columns or table are removed and become an empty filter.
vii. When Complex Filters exist in the External Filter Context and the same columns are used in the first argument of an Iterator function, then you can use KEEPFILTERS to perform an AND Logical Test rather than Overwrite.
viii. ALLSELECTED() DAX function, with no tables or columns added as arguments, serves as a filter modifier that will remove the row and column filters from the report or visual and leave the filters that are external to the report or visual intact.
ix. Column filters work on just the column.
x. Table filters work on Expanded Table and can go backwards across One-To-Many Relationship.

Calculating Averages at Different Grains and with Different Formulas

There are many different types of averages, such as mean, median, mode, and geometric mean. The most common average is the arithmetic mean, also called just the mean. This metric is commonly known as an **average**, and I will refer to it as such. The average calculation involves adding up a set of numbers and dividing by the count of that set of numbers. This metric is helpful because it gives you a single number that represents all the data points and can be used to gauge the typical performance for a given set of numbers.

In analytics, you are usually given a fact table with a certain **grain** (which refers to the size of the number in each row). The fact table in this project has a transactional grain, where each row in the table represents a sale of a product, by a specified SalesRep, on a specified date. If you average the sales amounts in all the rows of the fact table, because the grain of each number is at the transactional level, you are calculating the average transaction sales. If you add the transactional sales amounts to get the daily sales total amounts and then use those numbers to calculate an average, because the grain of each number is now at the day level, you are calculating the average daily sales. If you add the

transactional sales amounts to get the monthly sales total amounts and then use those numbers to calculate an average, because the grain of each number is now at the month level, you are calculating the average monthly sales. Each of these metrics communicates the typical sales amount at the given grain.

We will make these three average calculations:

- Average transactional sales by product
- Average daily sales by product
- Average monthly sales by product

If the goal is to calculate the average transactional sales, you can just use the Line Sales field from the fact table inside the AVERAGE DAX function, which works the same as the Excel worksheet AVERAGE function. That formula uses the fact table row line sales numbers as a set of numbers; it adds them up and divides by the count.

However, you often need to make aggregate calculations, such as averages, with a grain that is larger than the grain in the fact table. For example, to calculate the average daily sales, the grain of the numbers needed in the formula is larger than the grain of the numbers in the fact table. Luckily, DAX formulas can deal with such grain disparities easily; in fact, this ability is one of the main benefits of DAX formulas.

For the average daily sales calculation, there are two useful approaches to building the DAX formula:

1. The first approach is to pre-aggregate the daily sales amounts and then, once you have the daily sales totals, average those numbers. The pre-aggregation is necessary because there are many records in the fact table for any given day. You must add up the sales for each day and then, once you have that set of daily sales numbers at the correct grain, you can average them. For this approach, you can use the AVERAGEX DAX function.

   Note: If you needed to calculate the average daily sales with only the standard PivotTable tool and worksheet formulas, because there is no way to pre-aggregate numbers with a standard PivotTable calculation, you would be forced to create an intermediate table in the worksheet with the total sales for each date and then make a standard PivotTable from that intermediate table. This approach was common before the Data Model and DAX, but it was time-consuming, did not work well with large datasets, and could become very complex.

2. The second approach to calculating average daily sales is to just add up all sales and then divide by the unique count of dates in the Date field in the fact table. This approach is more straightforward than the first approach, but it is possible only because there is a field in the fact table that allows you to create a unique list of dates. For some calculations, such as average monthly sales, there is usually not a field in the fact table that allows you to get a unique count of months for the denominator, and therefore you cannot use this second approach (though the first approach will work). When you have an attribute field in the fact table, you can use the DIVIDE and DISTINCTCOUNT DAX functions.

   Note: If you needed to calculate the average daily sales with only the standard PivotTable tool and worksheet formulas, because there is no unique count calculation in the standard PivotTable, you would once again be stuck with a more inefficient worksheet solution if you wanted to use this second approach.

Examples on Next Page →

Average Transactional / Daily / Monthly Sales By Product DAX Formulas:

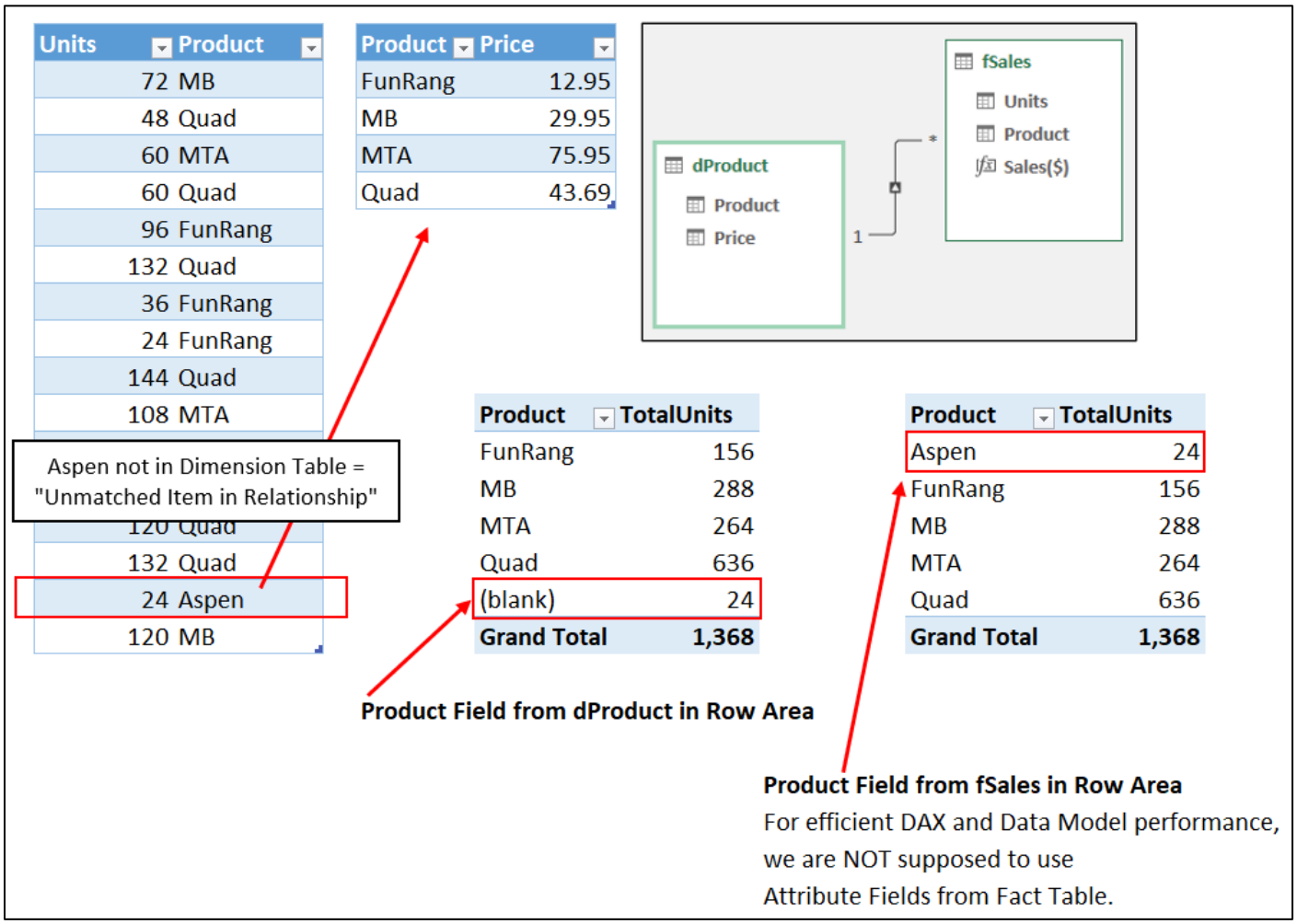
	A	B	C	D	E	F
2			In AVERAGEX we do NOT want Context Transition (duplicate fact table records, iterate fact table 2 times).	Table in 1st argument of AVERAGEX has no dups. In 2nd argument, we want Context Transition to bring Row Context Condition into formula and filter the fact table.	Table in 1st argument of AVERAGEX has no dups. In 2nd argument, we want Context Transition to bring Row Context Condition into formula and filter the fact table.	Because we have a date attribute field in fact table, we can use this formula which will tend to calculate more quickly than an AVERAGEX formula.
3			AveTransactionalSale:= AVERAGEX(fSales, RELATED(dProduct[RetailPrice])*fSales[UnitsSol	AveDailySales:= AVERAGEX(dDate, [TotalSales($)])	AveMonthlySales:= AVERAGEX(VALUES(dDate[EOMonth]), [TotalSales($)])	AveDailySalesOverFactTableAttributeColumn:= DIVIDE([TotalSales($)], DISTINCTCOUNT(fSales[Date]))
5		**Products**	**AveTransactionalSale**	**AveDailySales**	**AveMonthlySales**	**AveDailySalesOverFactTableAttri**
6		Aspen	2,282.48	96,952.67	2,927,524.88	96,952.67
7		Beaut	3,311.41	133,018.57	4,016,549.29	133,018.57
8		Bellen	2,287.77	97,119.03	2,932,548.15	97,119.03
9		Carlota	2,289.07	97,206.52	2,937,424.58	97,206.52
10		Eagle	1,839.06	250,209.92	7,566,693.10	250,209.92
11		Elevate	3,331.12	123,594.22	3,726,294.77	123,594.22
12		Flattop	2,400.75	78,732.26	2,369,207.56	78,732.26
13		Kangaroo	2,300.16	88,927.15	2,686,213.07	88,927.15
14		LongRang	3,979.08	74,568.31	2,235,865.75	74,568.31
15		NaturalElbo	3,417.89	73,563.14	2,210,397.16	73,563.14
16		Quad	4,055.98	551,954.44	16,691,863.45	551,954.44
17		Sunset	2,382.20	100,532.11	3,036,762.96	100,532.11
18		Sunshine	1,833.25	77,905.27	2,353,276.55	77,905.27
19		TriFly	1,234.46	30,189.04	909,504.81	30,189.04
20		Vrang	1,194.99	34,666.65	1,042,390.45	34,666.65
21		Yanaki	2,388.85	324,791.57	9,822,145.09	324,791.57
22		**Grand Total**	**2,586.22**	**2,182,017.16**	**65,987,208.47**	**2,182,017.16**
23						
24						
25		Products	Aspen			
26						
27		**Date**	**TotalSales($)**			
28		1/1/2017	60,753.25			
29		1/2/2017	34,705.45			
30		1/3/2017	49,625.55			
2654		3/15/2024	22,554.80			
2655		**Grand Total**	**254,694,664.85**			
2656						
2657		**Average Daily Aspen Sales:**		96,952.67		
2658				=AVERAGE(C28:C2654)		
2659						
2660		Products	Aspen			
2661						
2662		**EOMonth**	**TotalSales($)**			
2663		1/31/2017	1,307,205.35			
2747		1/31/2024	669,533.25			
2748		2/29/2024	664,917.50			
2749		3/31/2024	343,336.95			
2750		**Grand Total**	**254,694,664.85**			
2751						
2752		**Average Monthly Aspen Sales:**		2,927,524.88		
2753				=AVERAGE(C2663:C2749)		

One DAX Measure compared to:
a calculation process that requires multiple steps

Unmatched Items in a Relationship

This example can be found in the file named “17-UnmatchedItems.xlsx”.

As shown in the picture below, when you have a one-to-many relationship from a Dimension Table to a Fact Table, if there items in the fact table (many side) that are not in the Dimension Table (one-side) then when you use an attribute field (foreign key) from the fact table in a report or visual, all items will show, but if you use the primary key or other fields from the dimension table in a report or visual, you will show one blank cell that accumulates all missing items.



Date Table with the DAX functions GENERATE and ROW

These examples can be found in the file named “17-DAX-DateTableExamples.pbix”.

- The **ROW DAX table function** creates a one-row table with field names and values that you specify.
- The **GENERATE DAX table function** takes two or more tables and performs a cross-join (which is a Cartesian product in set theory) between the two table functions to generate a new third table. A cross-join simply matches up each row from the first table with a row from the second table. In the below example, because the first table is a single column of dates, and the second table is a single row with date attribute DAX formulas, each date in the first table will have a single row of data attributes added to it to create a full date dimension table.

```
dDateGENERATE =
GENERATE(
    CALENDAR(
        DATE(YEAR(MIN(fSales[Date])),1,1),
        DATE(YEAR(MAX(fSales[Date])),12,31)),
    ROW(
        "MonthNumber", MONTH([Date]),
        "Month", FORMAT([Date],"mmm"),
        "Year", YEAR([Date]),
        "EOMonth", EOMONTH([Date],0))
```

Date	MonthNumber	Month	Year	EOMonth
1/1/2024 12:00:00 AM	1	Jan	2024	1/31/2024 12:00:00 AM
1/2/2024 12:00:00 AM	1	Jan	2024	1/31/2024 12:00:00 AM
1/3/2024 12:00:00 AM	1	Jan	2024	1/31/2024 12:00:00 AM
1/4/2024 12:00:00 AM	1	Jan	2024	1/31/2024 12:00:00 AM
1/5/2024 12:00:00 AM	1	Jan	2024	1/31/2024 12:00:00 AM

```
DateTableAdvancedWithGENERATEandROW =
VAR DateColumn =
    CALENDAR(DATE(YEAR(MIN(fSales[Date])),1,1),DATE(YEAR(MAX(fSales[Date])),12,31))
RETURN
    GENERATE(
        DateColumn,
        VAR BaseDate = [Date]
        VAR MonthNumber = MONTH(BaseDate)
        VAR MonthName = FORMAT(BaseDate,"mmm")
        VAR WeekDayNumber = WEEKDAY(BaseDate,2)
        VAR DayOfWeek = FORMAT(BaseDate,"dddd")
        VAR YearStandard = YEAR(BaseDate)
        VAR EOMonthDate = EOMONTH(BaseDate,0)
        VAR StandardQuarter = ROUNDUP(MonthNumber/3,0)
        // April 1 is first day in Fiscal Period
        VAR FiscalQuarter = "Q-"& IF(StandardQuarter=1,4,StandardQuarter-1)
        VAR FiscalYear = IF(StandardQuarter=1,YearStandard-1,YearStandard)
        VAR FiscalPeriod = FiscalYear&" "&FiscalQuarter
        RETURN
        ROW(
            "Month Number", MonthNumber,
            "Month", MonthName,
            "Year", YearStandard,
            "EOMonth", EOMonthDate,
            "Quarter", StandardQuarter,
            "Fiscal Quarter", FiscalQuarter,
            "Fiscal Year", FiscalYear,
            "Fiscal Period", FiscalPeriod,
            "Weekday Number", WeekDayNumber,
            "Day Of Week",DayOfWeek

        )
    )
```

Only 1 input: Fact Table Date Field

Date	Month Number	Month	Year	EOMonth	Quarter	Fiscal Quarter	Fiscal Year	Fiscal
07/01/2024	7	Jul	2024	07/31/2024	3	Q-2	2024	2024 Q

Using the DISTINCTCOUNT and DIVIDE DAX Functions for faster calculating average

These examples can be found in the file named "17-21MillionRowTable.xlsx".

The DISTINCTCOUNT DAX function counts the number of unique values in a column. This function is particularly fast at calculating an answer because it communicates with the Data Model columnar database, which is programmed to store all original full table fields as unique list columns. Thanks to this columnar database characteristic and because the fact table has a date field that correctly marks each row with the date attribute, you can create an alternative average daily sales formula by using the DISTINCTCOUNT function, and the resulting formula will have a faster calculation time than the AVERAGEX formula. Examples shown here:

Table iterated is 21 million rows.	Table iterated in 7 rows and reduces calculation time over fact table formula by about 35%.
AveTransactionalSalesFact:= AVERAGEX(fSales, RELATED(dProduct[Price])*fSales[Units])	AveTransactionalSalesAlternative:= DIVIDE(SUMX(VALUES(dProduct[Price]), dProduct[Price]*CALCULATE(SUM(fSales[Units]))), COUNTROWS(fSales))

AveTransactionalSalesFact	AveTransactionalSalesAlternative
62.16	62.16

Cardinality of Tables in Iterator Functions

These examples can be found in the file named "17-21MillionRowTable.xlsx".

- Cardinality = number of items in a set (table or array) or number of iterations.
- Cardinality matters for Big Data Calculations because, in general, the smaller the cardinality or the fewer the number of iterations, the faster the formulas will calculate.
- Examples of Total Sales Calculations from Video:

Table iterated is 21 million rows.	Table iterated in 630 rows and reduces calculation time over fact table formula by about 20%	Table iterated in 7 rows and reduces calculation time over fact table formula by about 32%
TotalSalesIterateFact:= SUMX(fSales, RELATED(dProduct[Price])* fSales[Units])	TotalSalesIterateProduct:= SUMX(VALUES(dProduct), dProduct[Price]* CALCULATE(SUM(fSales[Units])))	TotalSalesIteratePrice:= SUMX(VALUES(dProduct[Price]), dProduct[Price]* CALCULATE(SUM(fSales[Units])))

TotalSalesIterateFact	TotalSalesIterateProduct	TotalSalesIteratePrice
1,310,909,098.70	1,310,909,098.70	1,310,909,098.70

Approximate Match Lookup with DAX:

These examples can be found in the file named "17-DAX-ApproximateMatchLookup.xlsx".

1. Approximate Match Lookup DAX Formula to lookup the correct Discount Based On Units Purchased (both fields in lookup table must be sorted smallest to biggest):

```
=CALCULATE(MAX(disDiscount[Discount]),FILTER(disDiscount,disDiscount[Units]<=fSales[Units]))
```

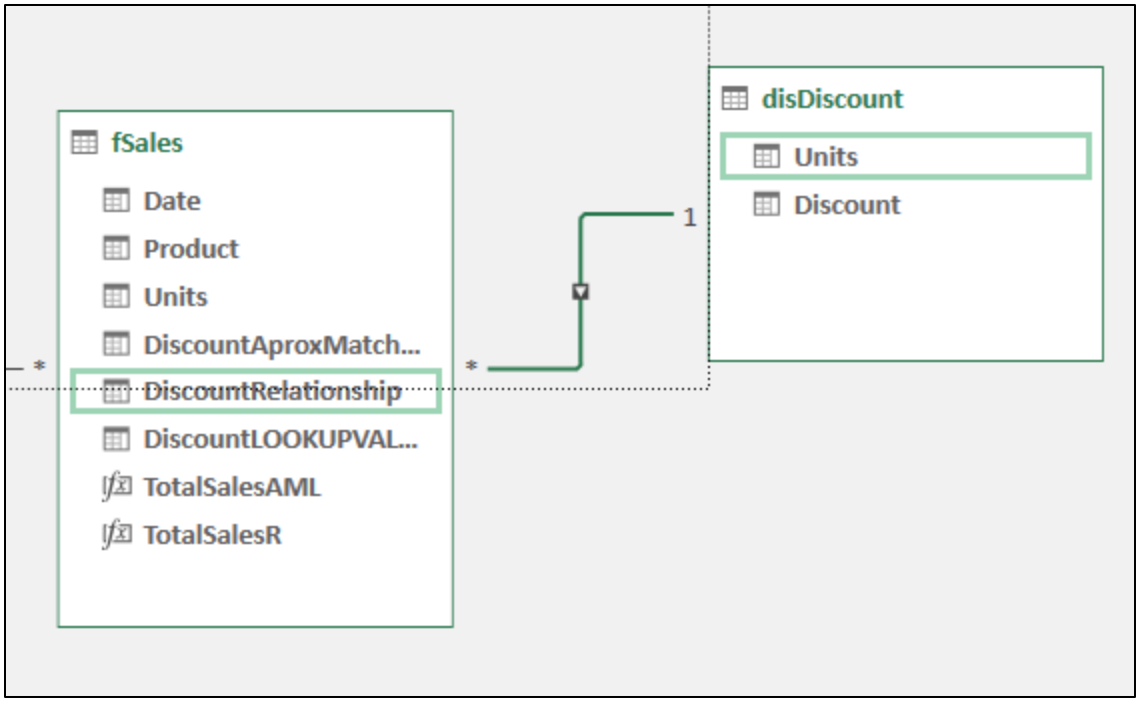
Product	Units	DiscountAproxMatchLookup	DiscountRelationship	DiscountLOOKUPVALUE
Quad	160	0.35	96	0.35
Carlota	11	0	0	0

2. Approximate Match Lookup DAX Formula to look create Foreign Key Column in Fact Table based on Units Field so that a relationship can be used to do Approximate Match Lookup:

```
=CALCULATE(MAX(disDiscount[Units]),FILTER(disDiscount,disDiscount[Units]<=fSales[Units]))
```

Product	Units	DiscountAproxMatchLookup	DiscountRelationship	DiscountLOOKUPVALUE
Quad	160	0.35	96	0.35
Carlota	11	0	0	0
Quad	36	0.2	12	0.2

3. Relationship can now be used with REALTED function to lookup correct Discount:



4. This is a stand-alone Approximate Match Lookup Formula that allows the Discount column to not require sorting from smallest to largest (Units Field does require a sort from smallest to biggest):

```
=LOOKUPVALUE(disDiscount[Discount],disDiscount[Units],CALCULATE(MAX(disDiscount[Units]),FILTER(disDiscount,disDiscount[Units]<=fSales[Units])))
```

Product	Units	DiscountAproxMatchLookup	DiscountRelationship	DiscountLOOKUPVALUE	Add Column
Quad	160	0.35	96	0.35	
Carlota	11	0	0	0	

ADDCOLUMNS and SELECTCOLUMNS DAX Table Functions

- **ADDCOLUMNS**(Table,"Name New Column", Expression) = Adds new column/s to a table. ADDCOLUMNS iterates Row-by-Row over the table in the first argument.
- Example of ADDCOLUMNS to add COGS to the fSales table:

```
QuadCOGSFILTERADDCOLUMN =
    ADDCOLUMNS(
        CALCULATETABLE(fSales,dProducts[Product]="Quad"),
        "COGS",RELATED(dProducts[Cost])*fSales[Units])
```

Date	SRID	ProductID	CustomerID	Units	LineSales	COGS
1/21/2027 12:00:00 AM	6	2	1	30	1290	596.25
4/15/2024 12:00:00 AM	6	2	1	98	4214	1947.75

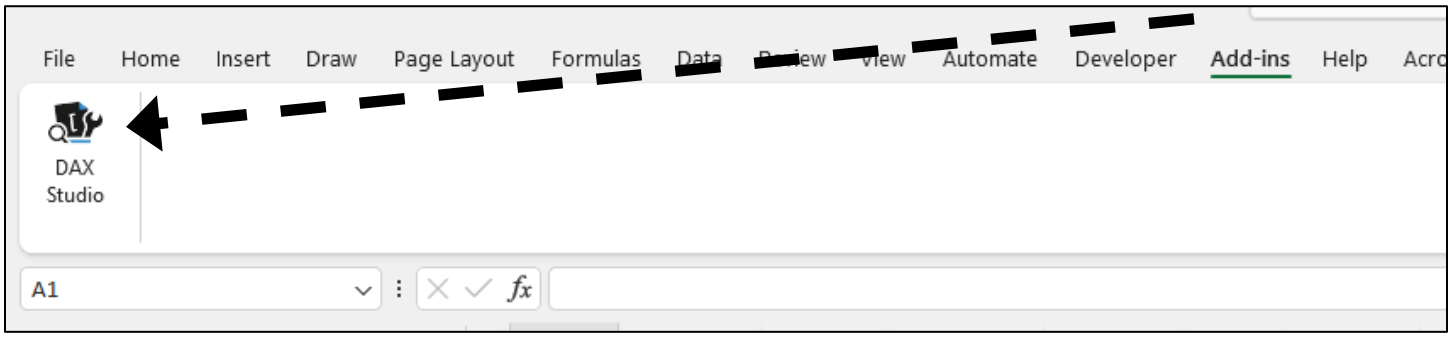
- **SELECTCOLUMNS**(Table,"Name New Column", Expression) Has the same signature as ADDCOLUMNS, and has the same behavior except that instead of starting with the <Table> specified, SELECTCOLUMNS starts with an empty table before adding columns. SELECTCOLUMNS iterates Row-by-Row over the table in the first argument.
- Example of SELECTCOLUMNS to select the "Units" and "Sales" columns from the fSales table and to create the two calculated columns "Product" and "COGS".

```
QuadCOGSCALCULTETABLESELECTCOLUMNS =
    SELECTCOLUMNS(
        CALCULATETABLE(fSales,dProducts[Product]="Quad"),
        "Product",RELATED(dProducts[Product]),
        "Units",fSales[Units],
        "Sales",fSales[LineSales],
        "COGS",RELATED(dProducts[Cost])*fSales[Units])
```

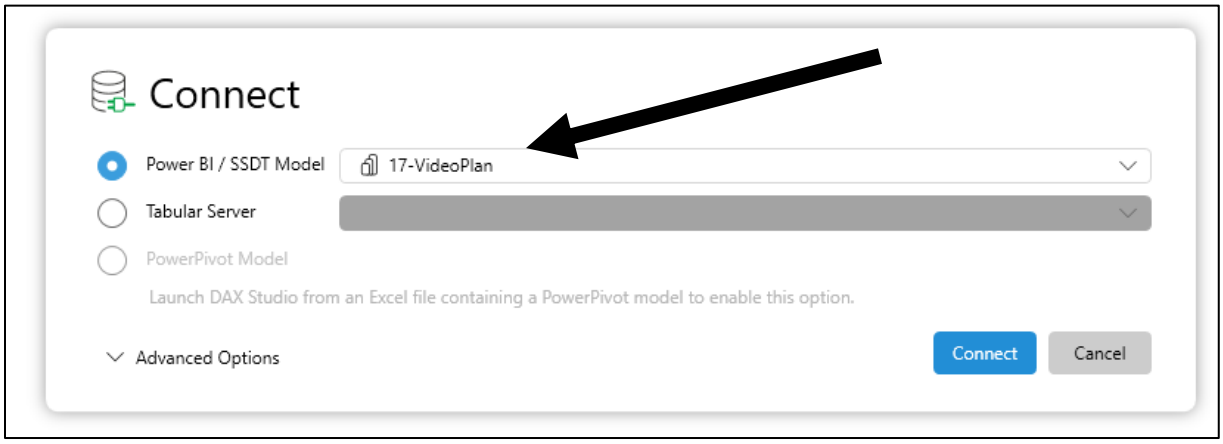
Product	Units	Sales	COGS
Quad	103	4429	2047.125
Quad	186	7998	3696.75
Quad	107	4601	2126.625

DAX Studio

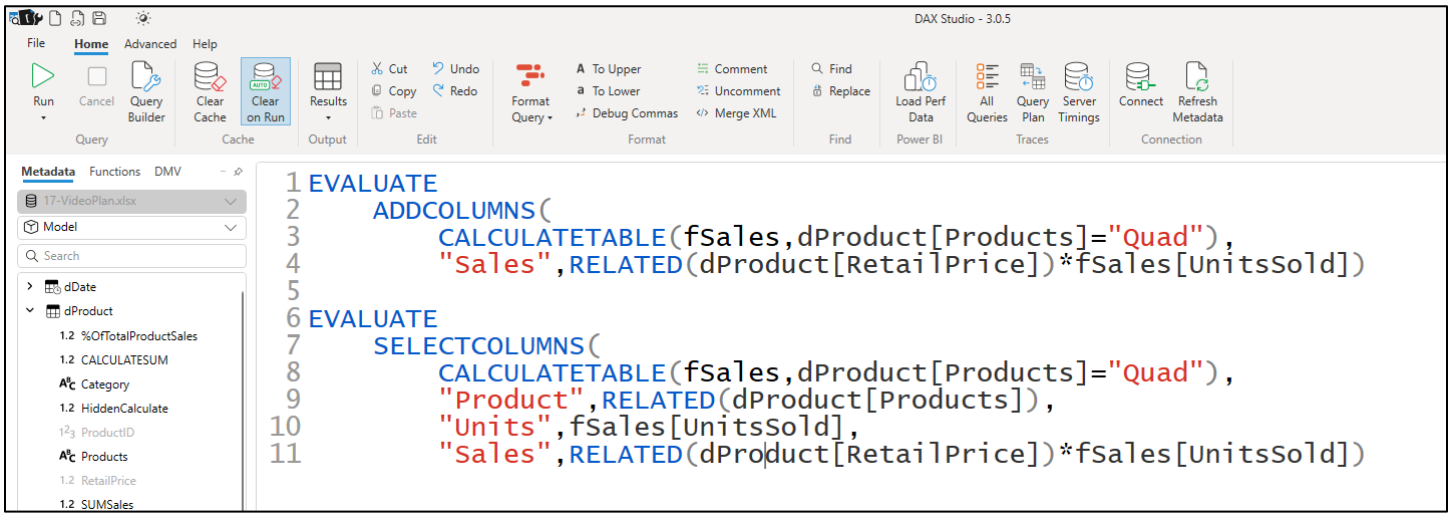
1. DAX Studio is a program that allows you to build DAX Formulas and time the speed of the formulas.
2. You can search for and download the program DAX Studio.
3. In Excel it appears in the Add-in Tab, as shown here:



4. To use DAX Studio for a Power BI Desktop file, you must open DAX Studio and in the first step select the Power BI Desktop file that you want to examine, as shown here:



5. DAX Studio only delivers table results (called a Query Result), and the DAX Code must always be preceded by the EVALUTAE command, as shown here:



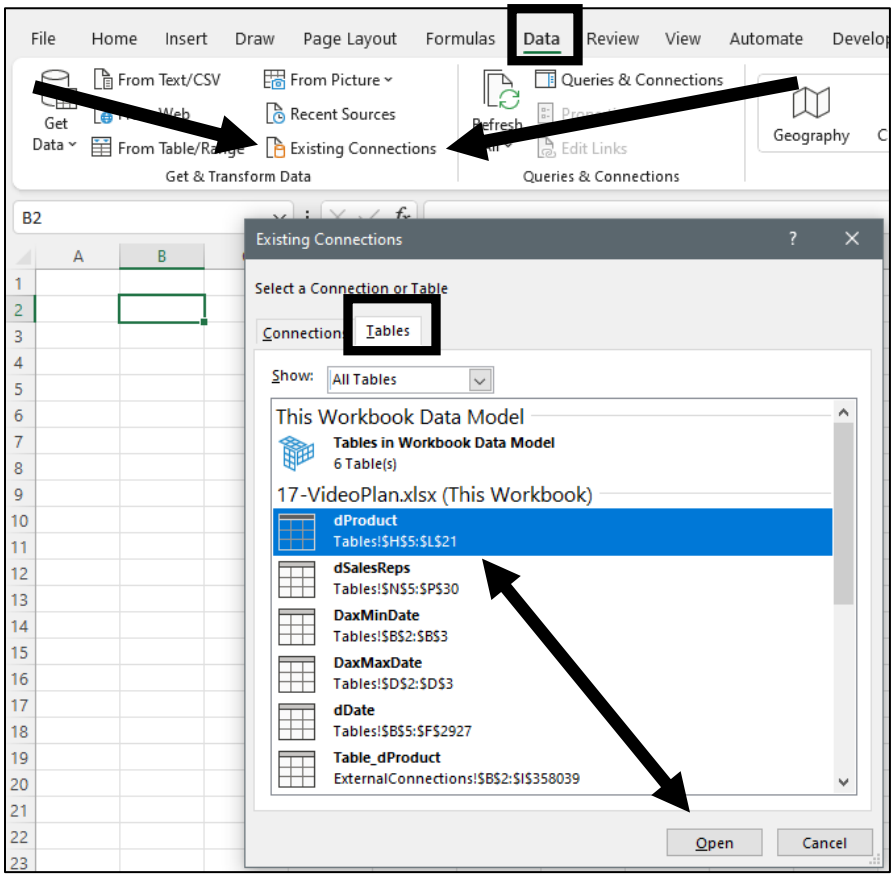
6. To create or time Measures (scalar values), you must house Measure in the ROW function and create a one row table, like:

```
EVALUATE
ROW("Fact",[TotalSalesIterateFact])

EVALUATE
ROW("Product",[TotalSalesIterateProduct])

EVALUATE
ROW("Price",[TotalSalesIteratePrice])

EVALUATE
ROW("Ave",[AveTransactionalSalesFact])

EVALUATE
ROW("AveAlternative",[AveTransactionalSalesAlternative])
```

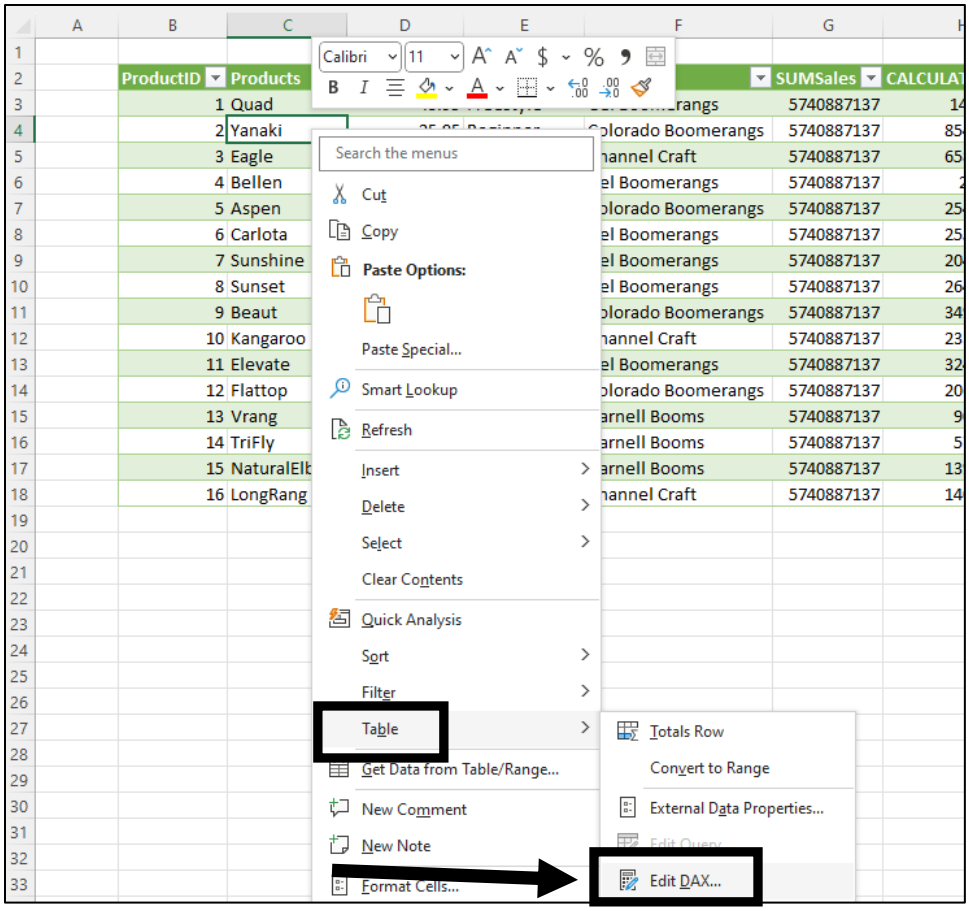
Using DAX with Existing Connections feature to extract Data From Data Model into Worksheet

1. This example is in the file named “17-M365ExcelClassFinished.xlsx”.
2. The Existing Connections feature in Excel allows us to extract data from the data model and load it to the worksheet.
3. This feature is very “clunky” and “primitive”.
4. Here are steps:

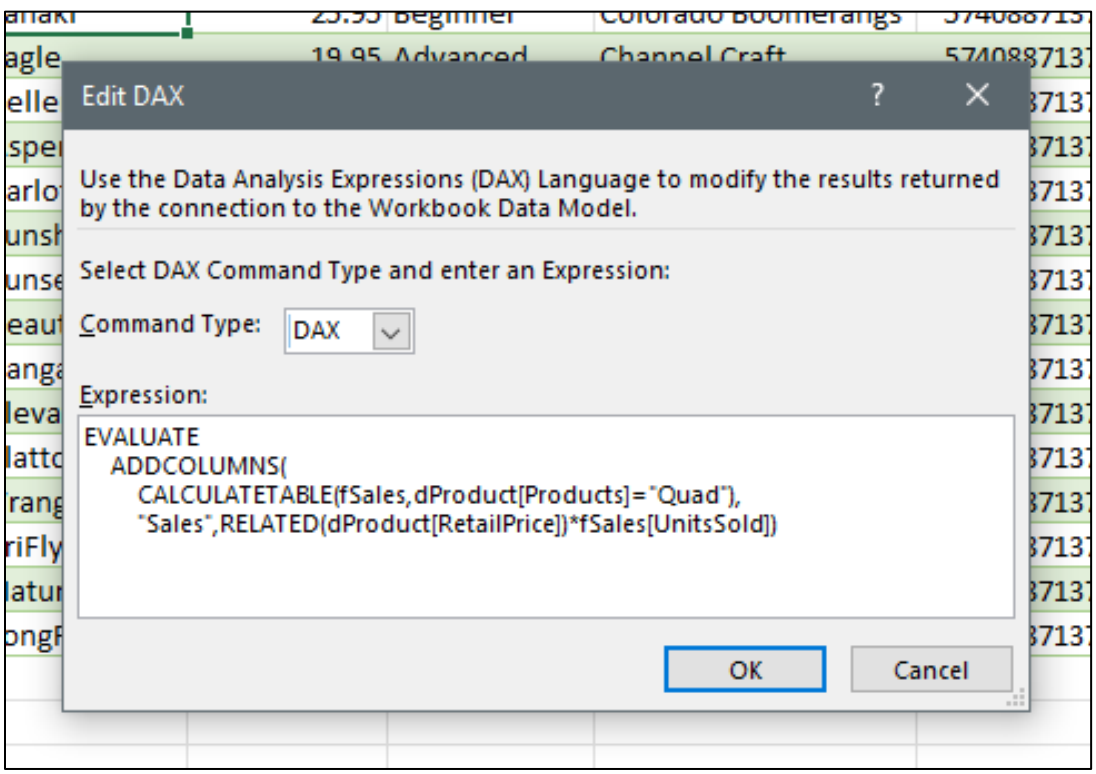
1. As shown below, Select cell in worksheet, click Existing Connections button in Get & Transform group in Data Ribbon tab, then in then in the Existing Connections dialog box select the Tables tab, then select a table from the Data Model and then click Open, then in the Import Data dialog box, click OK.



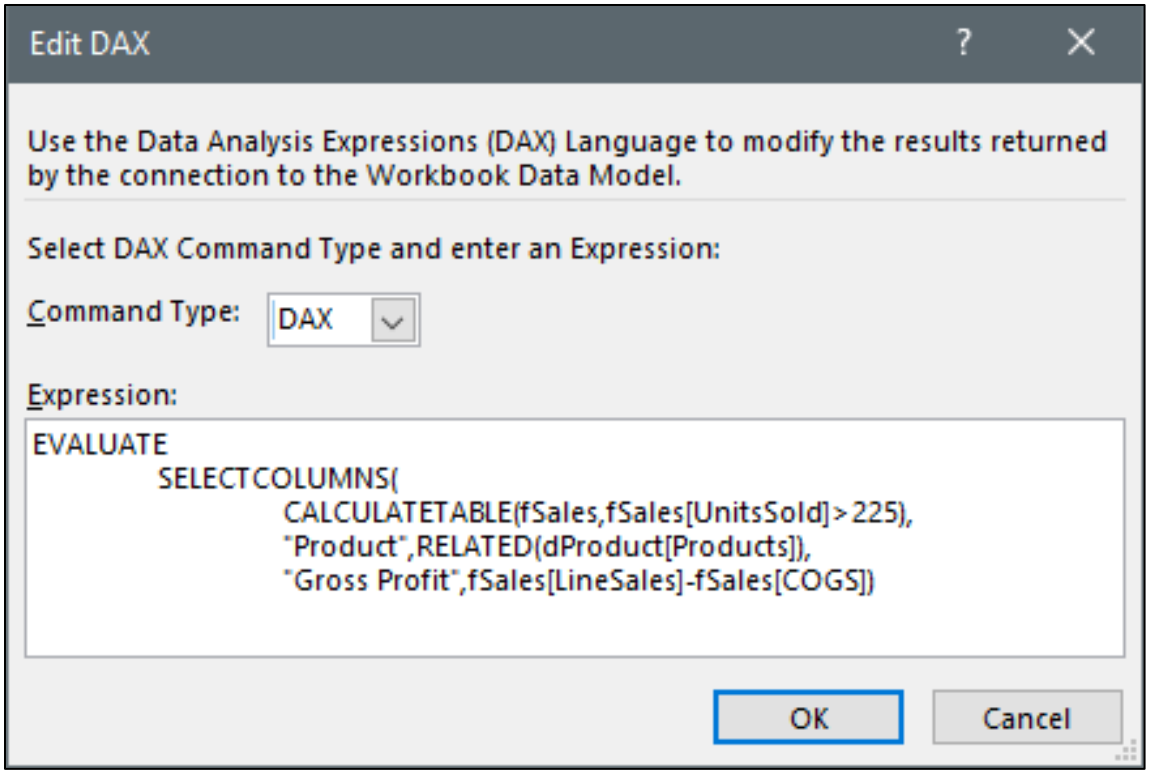
2. In a cell in the imported table, right click, point to Table, then click on Edit Dax, as shown here:



3. In the Edit DAX dialog box, select DAX from the command type dropdown arrow, then copy and paste the code from DAX Studio into the Expression area, as shown here:



4. Here is a second DAX Expression:



5. Result looks like this:

Product	Gross Profit
Quad	5978.89
Quad	6054.57
Quad	6155.48
Quad	5802.3
Quad	5751.85
Quad	5751.85
Quad	5751.85
Quad	6357.3
Quad	6306.85
Quad	6205.94
Quad	6054.57
Quad	6332.08
Quad	6357.3
Quad	5751.85

DAX Query View in Power BI Desktop

Just like with Excel's Existing Connections, we can use the new DAX Query View in Power BI Desktop to run a DAX query and return a table. This is a great tool to extract data into a table and to create and test new DAX formulas.

Keywords:

EVALUATE, ORDER BY, START AT, DEFINE, MEASURE, VAR, TABLE, COLUMN

Each keyword defines a statement used for the duration of the query:

EVALUATE is written before the DAX table statement. At least one EVALUATE statement is required, but you can have many EVALUATE statements.

DAX:

```
EVALUATE <table>
```

table = Table expression

ORDER BY (optional) defines one or more expressions that are evaluated in each row of the table and is used to sort query results.

DAX:

```
EVALUATE <table>
[ORDER BY {<expression> [{ASC | DESC}]}[, …]]
```

expression = DAX expression that returns a single scalar value for each row in the table.

ASC = (default) Ascending

DESC = Descending

START AT (optional) works with the ORDER BY keyword to determine where the query starts

DAX:

```
EVALUATE <table>
[ORDER BY {<expression> [{ASC | DESC}]}[, …]
[START AT {<value>|<parameter>} [, …]]]
```

value = Constant value. Cannot be an expression

parameter = Name of a parameter in an XMLA statement prefixed with an @ character

DEFINE, MEASURE, VAR, TABLE, COLUMN are all used together to define DAX elements that can be used throughout the query. Definitions precede the EVALUATE statement and are valid for all EVALUATE statements in the query. You can have only one definition

DAX:

```
[DEFINE
  (
   (MEASURE <table name>[<measure name>] = <scalar expression>) |
   (VAR <var name> = <table or scalar expression>) |
   (TABLE <table name> = <table expression>) |
   (COLUMN <table name>[<column name>] = <scalar expression>) |
  ) +
]


(EVALUATE <table expression>) +
```

Entity = MEASURE, VAR, TABLE[1], or COLUMN[1]

name = The name of a measure, var, table, or column definition. It cannot be an expression. The name does not have to be unique. The name exists only for the duration of the query.

expression = Any DAX expression that returns a table or scalar value. The expression can use any of the defined entities. If there is a need to convert a scalar expression into a table expression, wrap the expression inside a table constructor with curly braces {}, or use the ROW() function to return a single row table.

> **Caution:** Query scoped TABLE[1] and COLUMN[1] definitions are meant for internal use only. While you can define TABLE and COLUMN expressions for a query without syntax error, they may produce runtime errors and are not recommended.

DAX **VAR** note:

```
VAR <name> = <expression>
```

name — The name of the variable (identifier).
Delimiters are not supported. For example, 'varName' or [varName] will result in an error.
Supported character set: a-z, A-Z, 0-9.
0-9 are not valid as first character.
__ (double underscore) is allowed as a prefix to the identifier name.
No other special characters are supported.
Reserved keywords not allowed.
Names of existing tables are not allowed.
Empty spaces are not allowed.

expression — A DAX expression which returns a scalar or table value.

Video Examples:

Run

```
DEFINE
Update model: Add new measure
MEASURE fSales[TotalSales] = SUMX(fSales,RELATED(dProduct[RetailPrice])*fSales[UnitsSold])
VAR GTSales = [TotalSales]

EVALUATE
SELECTCOLUMNS(
    dProduct,
    dProduct[Products],
    "%OfTotalSales",DIVIDE([TotalSales],GTSales))
ORDER BY dProduct[Products]
START AT "NaturalElbow"
```

Results | Result 1 of 1 | Copy

	dProduct[Products]	[%OfTotalSales]
1	NaturalElbow	0.02
2	Quad	0.25
3	Sunset	0.05

DAX queries will be saved to your model They won't be visible when published in the F

Run

```
DEFINE
VAR LastSalesDateOneYearBack = EDATE(MAX(fSales[Date]),-12)
VAR X = ROW("X",LastSalesDateOneYearBack)

EVALUATE
SELECTCOLUMNS(
    dDate,
    dDate[Date],
    "TestDateHelper",dDate[Date]<=LastSalesDateOneYearBack)
```

Results | Result 1 of 1 | Copy

	dDate[Date]	[TestDateHelper]
1	1/1/2017 12:00:00 AM	True
2	1/2/2017 12:00:00 AM	True
3	1/3/2017 12:00:00 AM	True

Run

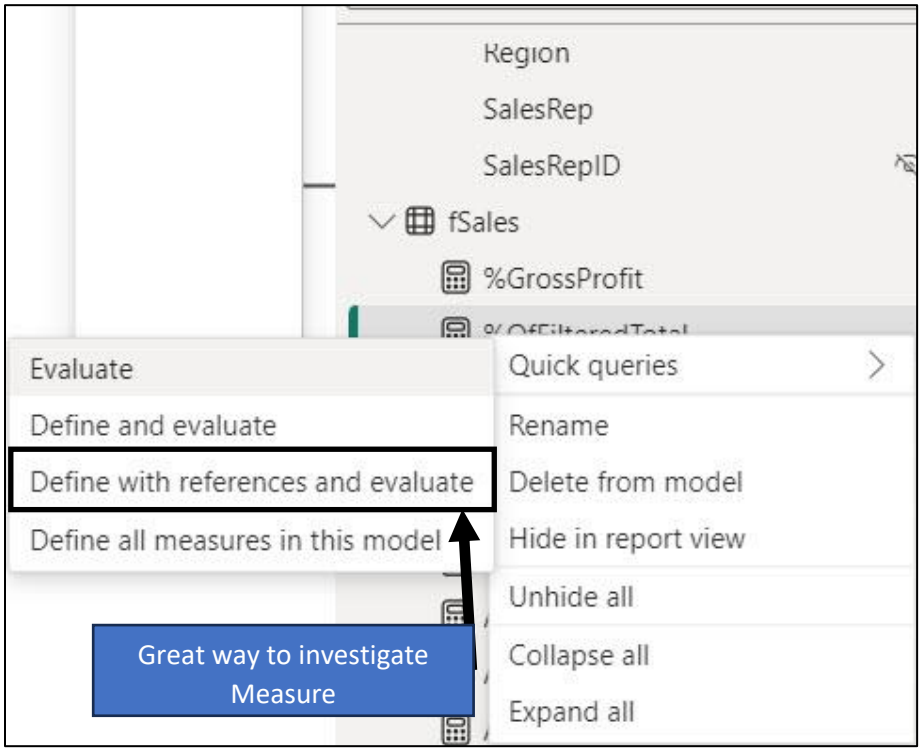
```
EVALUATE
SELECTCOLUMNS(
    dProduct,
    "TotalSales",[TotalSales($)],
    "TS",SUMX(
        VALUES(dProduct[RetailPrice]),
        dProduct[RetailPrice]*CALCULATE(SUM(fSales[UnitsSold]))))
```

Results | Result 1 of 1 | Copy

	[TotalSales]	[TS]
1	1452192120.3	1452192120.3
2	854526623.25	854526623.25
3	658302299.55	658302299.55

Define with References and evaluate:

- Right-click Measure and choose one of three options:
  - Evaluate = Shows measure and result of measure
  - Define and evaluate = Shows measure as a defined variable and result of measure
  - Define with references and evaluate = shows all formulas and sub formulas as variables and shows results for each (see example below).
  - Define all measures in model = allows you to check result for each measure.



Run

Define with references and evaluate = shows all formulas and sub formulas as variables and shows results for each

```
DEFINE
    Update model: Overwrite measure
    MEASURE 'fSales'[TotalSales($)] = SUMX(
            fSales,
            RELATED(dProduct[RetailPrice]) * fSales[UnitsSold]
        )
    Update model: Overwrite measure
    MEASURE 'fSales'[FilteredGrandTotal] = CALCULATE(
            [TotalSales($)],
            ALLSELECTED()
        )
    Update model: Overwrite measure
    MEASURE 'fSales'[%OfFilteredTotal] = DIVIDE(
            [TotalSales($)],
            [FilteredGrandTotal]
        )

EVALUATE
    SUMMARIZECOLUMNS(
        "TotalSales($)", 'fSales'[TotalSales($)],
        "FilteredGrandTotal", 'fSales'[FilteredGrandTotal],
        "%OfFilteredTotal", 'fSales'[%OfFilteredTotal]
    )
```

Results | Result 1 of 1 | Copy

	[TotalSales($)]	[FilteredGrandTotal]	[%OfFilteredTotal]
1	5740887136.55	5740887136.55	1