

# Introduction to Creating Measures using DAX in Power BI

**Data Analysis Expressions (DAX)** is a programming language that is used throughout Microsoft Power BI for creating calculated columns, measures, and custom tables.

It is a functions, operators, and constants that can be used in a formula, or expressions to calculate and return one or more values

DAX can also be used to solve a number of calculations and data analysis problems which can help you create new information from data that is already in your model

## Use Calculated Columns

DAX allows you to augment the data that you bring in from different data sources by creating a calculated column that didn't originally exist in the data source

### THIS FEATURE SHOULD BE USED SPARINGLY

DAX can be used by

Selecting the **Elipsis (...) button** on the table in the **Fields list**

=> and then, **selecting New column**

=> A new DAX formula should appear in the formula bar underneath the ribbon at the top

An example DAX query to calculate the total price of a table called Sales OrderDetails

Total Price = "Sales OrderDetails"[Quantity] \* "Sales OrderDetails"[Unit Price]

This DAX expression takes the quantity value and multiplies it with the unit price value for each individual row. The result is depicted below

Order ID	Product ID	Quantity	Unit Price
10248	11	12	\$14
10248	42	10	\$9.8
10248	72	5	\$34.8
10249	14	9	\$18.6
10249	51	40	\$42.4
10250	41	10	\$7.7
10250	51	35	\$42.4
10250	65	15	\$16.8
10251	22	6	\$16.8
10251	57	15	\$15.6
10251	65	20	\$16.8
10252	20	40	\$64.8
10252	33	25	\$2
10252	60	40	\$27.2
10253	31	20	\$10

Order ID	Product ID	Quantity	Unit Price	Total Price
10248	11	12	\$14	\$168
10248	42	10	\$9.8	\$98
10248	72	5	\$34.8	\$174
10249	14	9	\$18.6	\$167.4
10249	51	40	\$42.4	\$1,696
10250	41	10	\$7.7	\$77
10250	51	35	\$42.4	\$1,484
10250	65	15	\$16.8	\$252
10251	22	6	\$16.8	\$100.8
10251	57	15	\$15.6	\$234
10251	65	20	\$16.8	\$336
10252	20	40	\$64.8	\$2,592
10252	33	25	\$2	\$50
10252	60	40	\$27.2	\$1,088

Calculated columns are materialized in the .pbix Power BI file extension

Too many calculated columns will slow performance and cause you to reach the maximum Power BI file size sooner

### Create a Custom Column

Three ways to create a custom column in Power BI:

- Create the column in the source query when you get the data, for instance, by adding the calculation to a view in a relational database

An example is pulling data from a relational database, such as SQL

```
CREATE VIEW OrdersWithTotalPrice
AS
SELECT unitprice, qty, unitprice * qty as TotalPrice
FROM sales.salesorders
```

Good way of doing it as it makes the data source do the calculations for you

- Create a custom column in Power Query

Custom column dialog uses the M language to create the new column. M language is out of scope for the purposes of this module

- Create a calculated column by using DAX in Power BI Desktop

You do not need to refresh the dataset to see the columns

Unlike creating a column in the source query  
where you have to refresh the dataset to see updates

DAX calculated columns does not compress well

The earlier you can create a column – the better

It is not considered optimal practice to use DAX for calculations if you can use a different mechanism

### One Way to Avoid Using Calculated Columns

To use one of the X functions

- SUMX
- COUNTX
- MINX

These functions allow you to create measures that are aware of the data in individual rows and calculate totals based on the totals in the row

These functions are called iterator functions

because though they are used in measures,  
they iterate over the individual rows to do their calculations

An X function will perform better and use less disk space than a calculated column

CUSTOM COLUMNS are USEFUL for when you have to OPERATE ROW BY ROW

## Use Measures

A simpler method of calculation

An example scenario:

Consider a situation where **you want an aggregation that operates over the entire dataset and you want the total sales of all rows.**

Furthermore

**you want to slice and dice that data by other criteria like total sales by year, by employee, or by product**

To accomplish these tasks, **you would use a measure** – this can be done through **a quick measure**

## Create a Quick Measure

In order to **create a quick measure** in Power BI desktop

=> **right-click** or **select the elipsis (...) button** next to any item in the **Fields pane**

=> **select New quick measure** from the menu that appears

=> the **Quick measures** screen will appear

In this window you can **select a calculation and the column that you want to operate over.**

Power BI **creates the DAX measure for you** and **displays the DAX.** It can be a **helpful way to learn the DAX syntax**

## Create a Measure

Measures are **some of the most common data analyses**

An example scenario:

You want to **create a measure that totals your new column for the entire dataset.**

Similar to **create columns**, go to the **Fields list**, **click the elipsis (...)** on the selected field

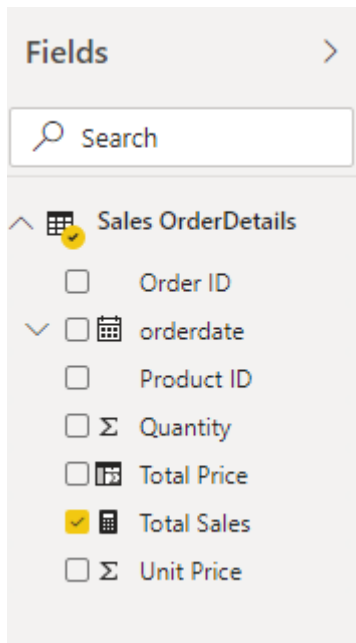
=> Select **New measure**

You can then **enter a DAX that measures a given business need**

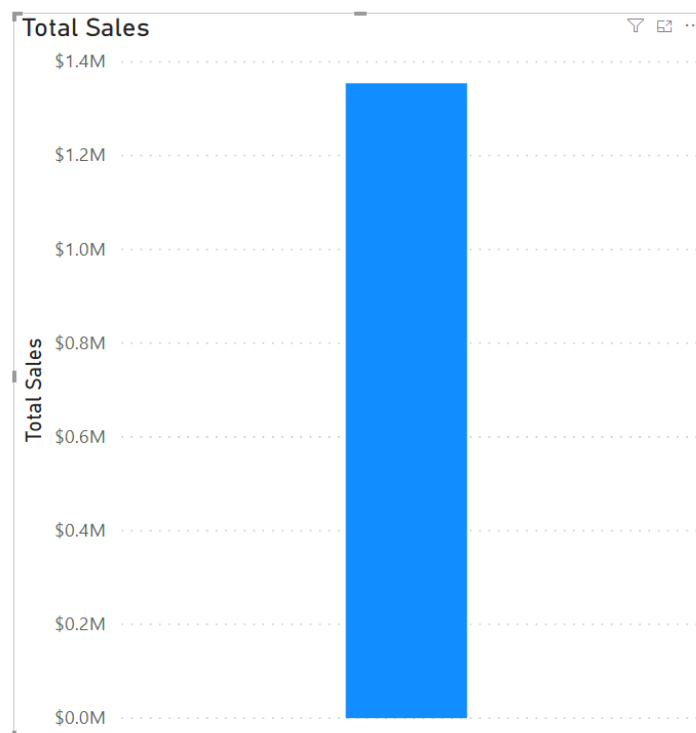
e.g.,

Total Sales = SUM('Sales OrderDetails'[Total Price])

The measure then appears as:



You can then **drag Total Sales** over to the **report design surface** to see the **total sales for the entire organization in a column chart**, as shown below:



## Differences Between a Calculated Column and a Measure

The **MAIN FUNDAMENTAL DIFFERENCE**:

**A calculated column creates a value for each row in a table**

If a table has 1,000 rows then it will have 1,000 values in the calculated column

Calculated column values are stored in the Power BI .pbix file

Each calculated column will increase the space that is used in that file

**Measures are calculated on demand**

Power BI **calculates the correct value when the user requests it**

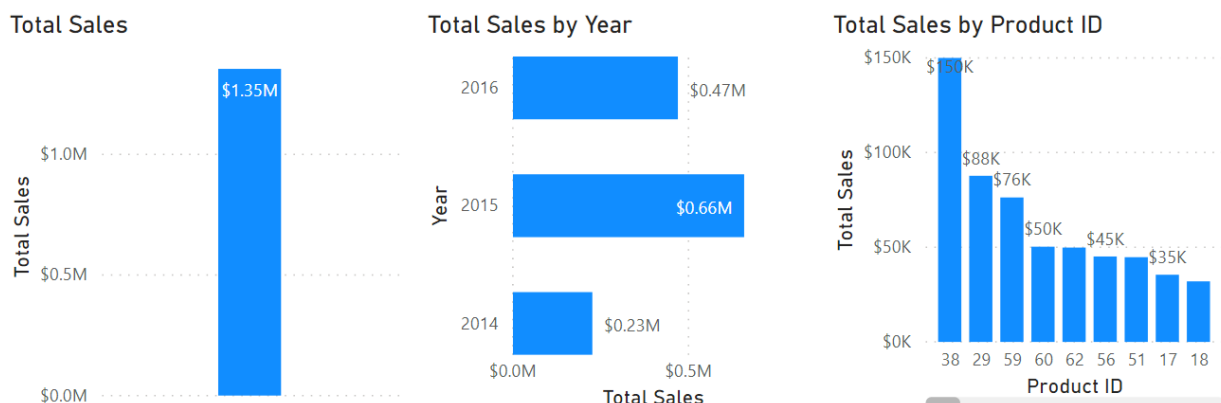
KEY: When you **dragged the Total Sales measure onto the report**, Power BI **calculated the correct total and displayed the visual**

Measures DO NOT add to the overall disk space of the Power BI .pbix file

**Measures are calculated based on the filters that are used by the report user**

These **filters combine to create the filter context**

## Understand Context



**Context is difficult to explain**, the following visual will demonstrate how context affects DAX measures so you can see how they interact together.

All three visuals **use the same exact DAX measure: Total Sales**

The first visual

**Total sales measure** for the **entire dataset**

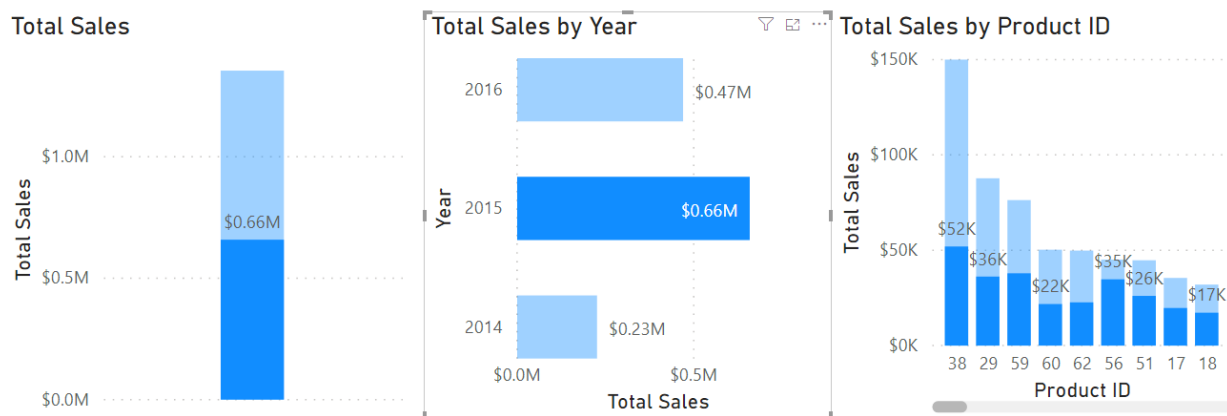
The second visual

**Total sales** is **broken down by year**

The third visual

**Total sales** is **broken down by Product ID**

Hence, we have **one measure** (Total Sales), but **three different contexts of how the measure is used**



**Interactions with a visual** can also be **changed depending on context**

For example, if you **select the 2015 date in the 2<sup>nd</sup> visual**, as shown above.

**The visual displays the measure with the SPECIFIED CONTEXT of 2015**

Many other factors like above (clicking on the 2015 entry in the 2<sup>nd</sup> visual) can **affect how DAX formulas are evaluated**

- Slicers
- Page filters
- And more...

can **affect how a DAX formula is calculated and displayed**

## **Use the Calculate Function**

The **CALCULATE function** in DAX is **one of the most important functions** that a data analyst can learn.

The **CALCULATE function** is your method of **creating a DAX measure** that **will override certain portions of the context** that are **being used to express the correct result**

An example calculate function,

**Always calculate the total sales for 2015, regardless of which year is selected**

Total Sales for 2015 = CALCULATE(SUM('Sales'[Total Price]), YEAR('Sales'[OrderDate]) = 2015)

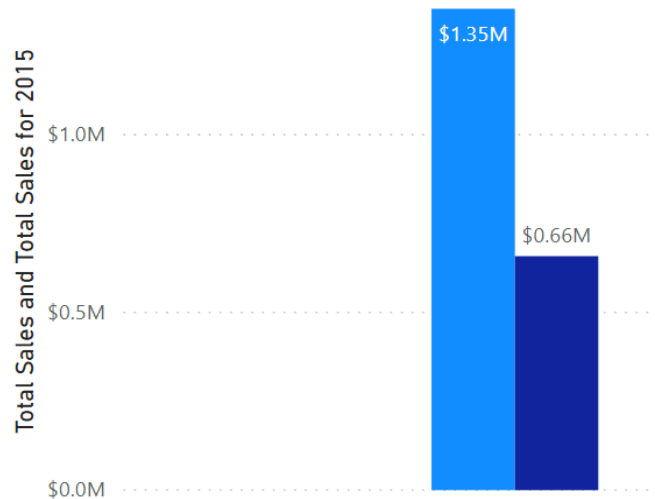
In this case, **CALCULATE** is **aggregating the Total Price column for ONLY the year 2015**.

Hence, when **other filters are applied** (e.g, clicking on the year 2016 in a report), **this specific clause will FILTER the CONTEXT to ONLY 2015**

An example of the measure being added to a report can be seen below:

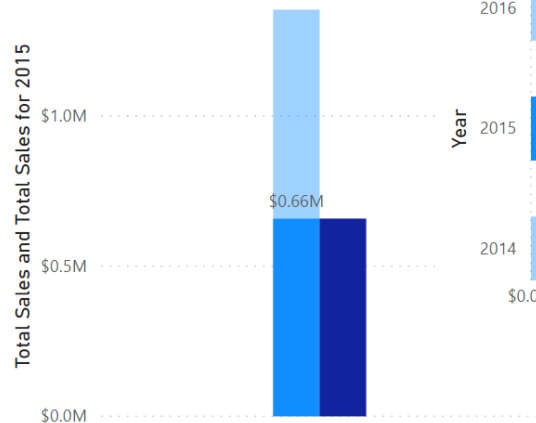
Total Sales and Total Sales for 2015

● Total Sales ● Total Sales for 2015

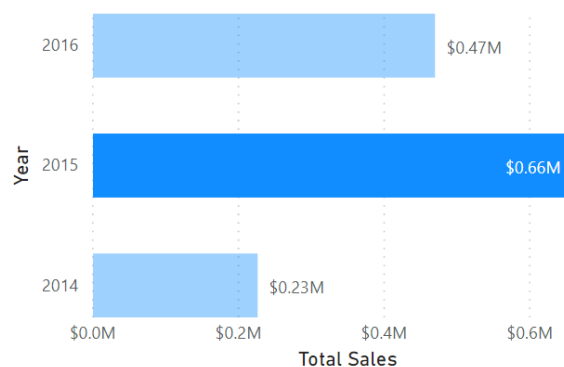


Total Sales and Total Sales for 2015

● Total Sales ● Total Sales for 2015



Total Sales by Year



Hence, when **the year 2015 is selected and the filter applied**. The total measure will still show.

Furthermore, **if the year 2016 is selected and the filter applied**. The **measure will remain as \$0.66M** as that is the date that the calendar was built on

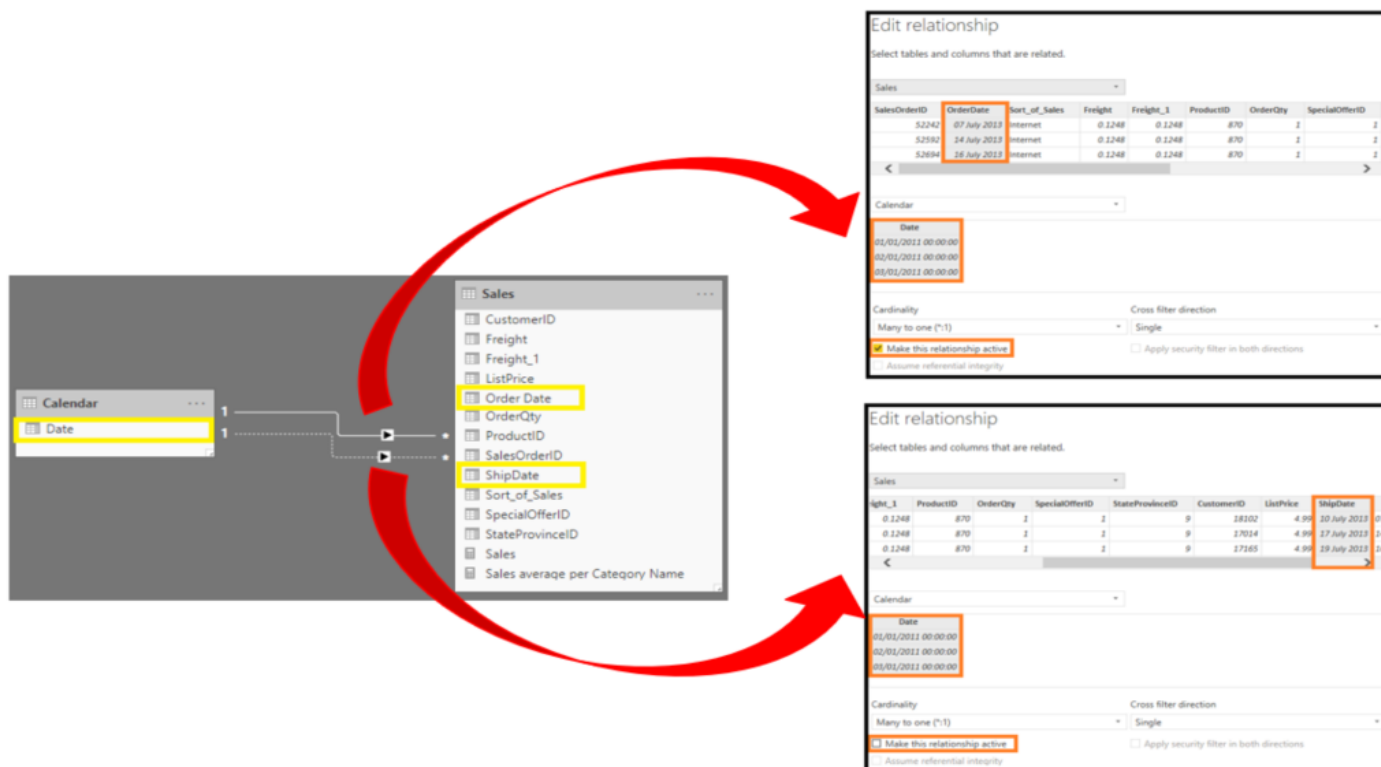
### **Use Relationship Effectively**

Another DAX function that **allows you to override the DEFAULT behaviour is USERELATIONSHIP**

This **function** will **change which relationship is used in a single measure**

Hence, allowing us to **use an inactive relationship in a single measure**

Consider the following scenario,



It **shows an established relationship** between the **Date and OrderDate columns**

There is an **active relationship** between the **Date column and the OrderDate column**

There is an **inactive relationship** between the **Date column and the OrderDate column**

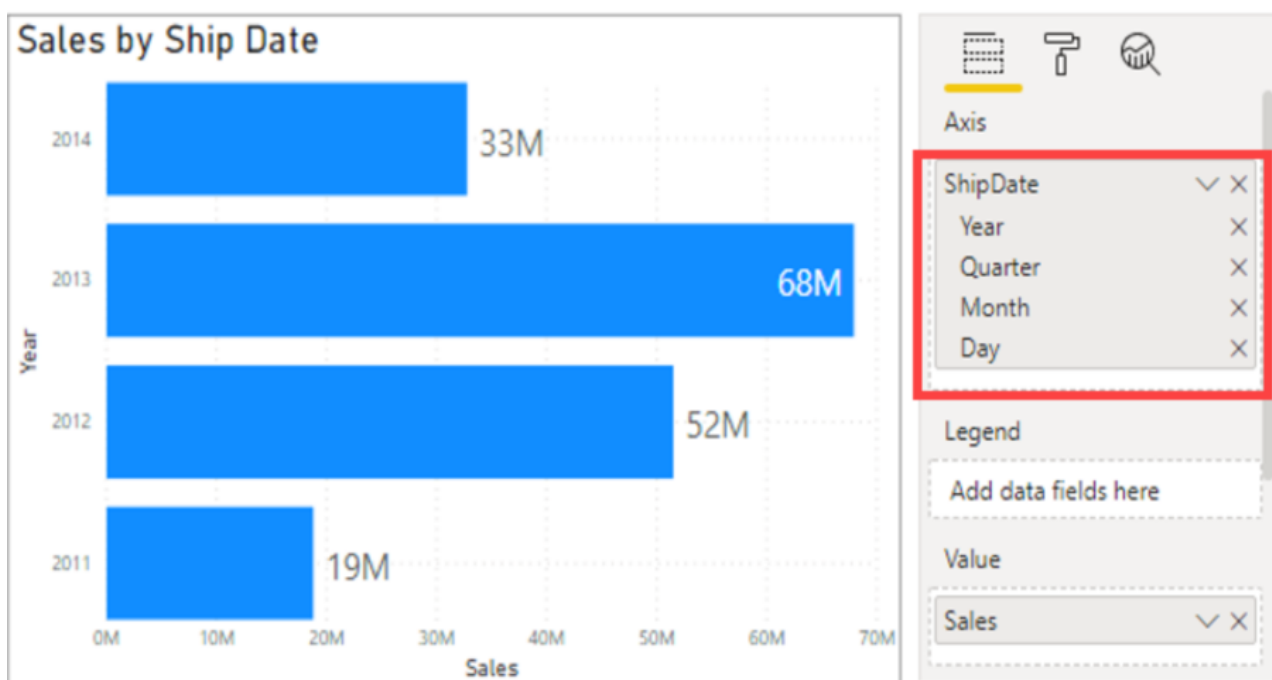
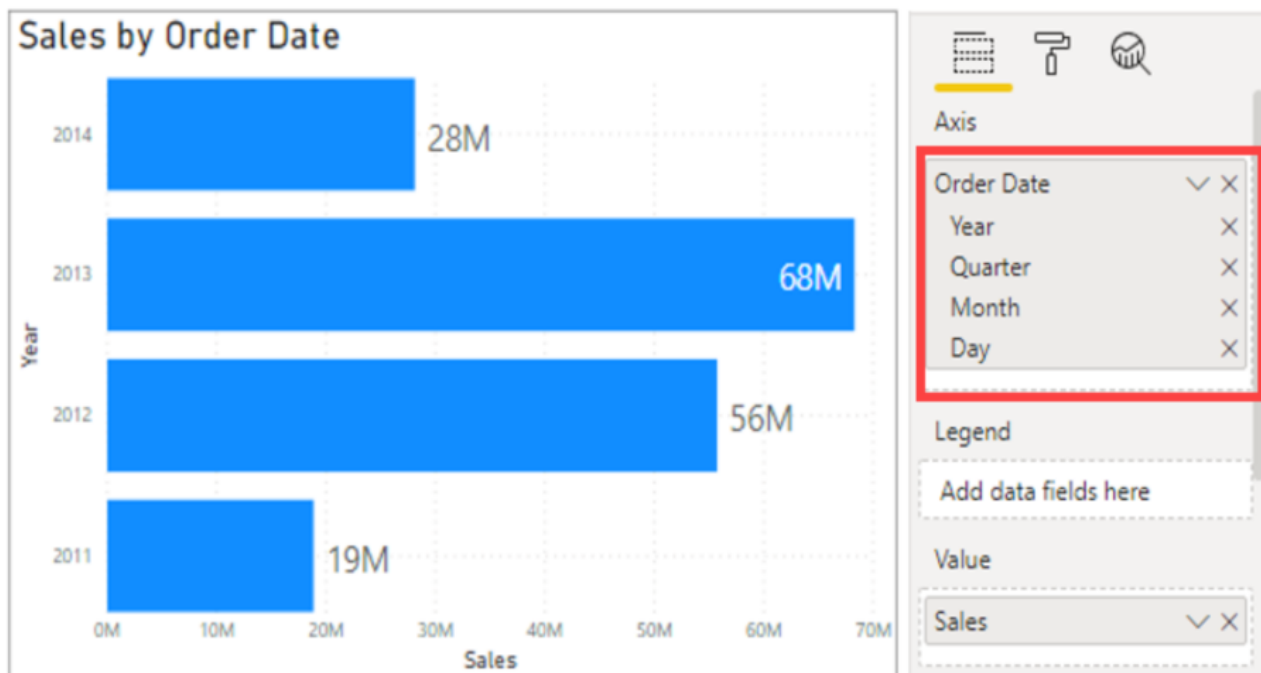
Will never be used **UNLESS EXPLICITLY STATED IN A MEASURE**

Hence, **any slicing on the data** will be **along the OrderDate column**

Sometimes you **WANT TO OVERRIDE THE DEFAULT CONTEXT**

This is so that you can **create measures that behave according to your intentions,**  
**REGARDLESS OF WHAT THE USER SELECTS**





Hence, we can **build the first visual relatively easily** as **there is already an active relationship between it and the date column**

We can do it through dragging the OrderDate onto the Y-axis slot for the visual

This is **not possible for the 2<sup>nd</sup> visual** unless we add the following DAX query **and use the USERELATIONSHIP expression**:

Sales by Ship Date = CALCULATE(SUM(Sales[TotalPrice]),  
USERELATIONSHIP(Sales[OrderDate], 'Calendar'[Date]))

This **allows developers to make additional calculations on inactive relationships** by **overriding the default active relationship** between the two tables in a DAX expression,

Hence, allowing us to **create the 2<sup>nd</sup> visual**

## **Create Semi-Additive Measures**

In situations where **you don't want the standard evaluation behaviour** in Power BI, you can use the **CALCULATE and/or USERELATIONSHIP** functions

One of those situations **is when you have a semi-additive problems to resolve**.

**Standard measures** are **simple concepts**.

Where they might use **SUM, AVERAGE, MIN, and MAX functions**

Thus far, you've been **using SUM** for the **Total Sales measure**

Occasionally, summing a measure doesn't make sense, e.g.,

If you have 100 mountain bikes, and on Tuesday you have 125 mountain bikes, you wouldn't want to add those together to indicate 225 mountain bikes between those two days.

In this case, **if you want to know the stock levels for March**, you would need to tell Power BI **not to add the measure but instead take the last value for the month of March and assign it to a visual**

The DAX query could look like this:

Last Inventory Count =

```
CALCULATE (  
    SUM('Warehouse'[Inventory Count]),  
    LASTDATE('Date'[Date]))
```

This approach will **stop the SUM from crossing all dates**. Instead you will only **use the SUM function on the last date of the time period**

## **Work with Time Intelligence**

**All data analysts will have to deal with time.**

**HIGHLY RECOMMENDED TO:**

- Create or import a dates table

This approach will **help make date and time calculations much simpler in DAX**

While some **time calculations are SIMPLE in DAX**, others are **MORE DIFFICULT**

An example of a time intelligence measure:

**a running total that increments for each month but resets when the year change**, as shown below

Month	2014	2015	2016
January		\$66,692.8	\$100,854.72
February		\$107,900	\$205,416.67
March		\$147,879.9	\$315,242.12
April		\$203,579.29	\$449,872.68
May		\$260,402.99	\$469,771.34
June		\$299,490.99	\$469,771.34
July	\$30,192.1	\$354,955.92	\$469,771.34
August	\$56,801.5	\$404,937.61	\$469,771.34
September	\$84,437.5	\$464,670.63	\$469,771.34
October	\$125,641.1	\$534,999.13	\$469,771.34
November	\$175,345.1	\$580,912.49	\$469,771.34
December	\$226,298.5	\$658,388.75	\$469,771.34
<b>Total</b>	<b>\$226,298.5</b>	<b>\$658,388.75</b>	<b>\$469,771.34</b>

This can be coded in DAX through the following expression:

```
YTD Total Sales = TOTALYTD
(
    SUM('Sales OrderDetails'[Total Price])
    , Dates[Date]
)
```

The **TOTALYTD()** function

**Takes an argument for the type of calculations**

In the case above, the SUM of the Total Price column is used as the first argument

The **second argument that you want to operate over is the Dates field**

You can use **all functions with YTD, MTD, QTD in a similar fashion**

Another example of **a time intelligence calculation/work** is:

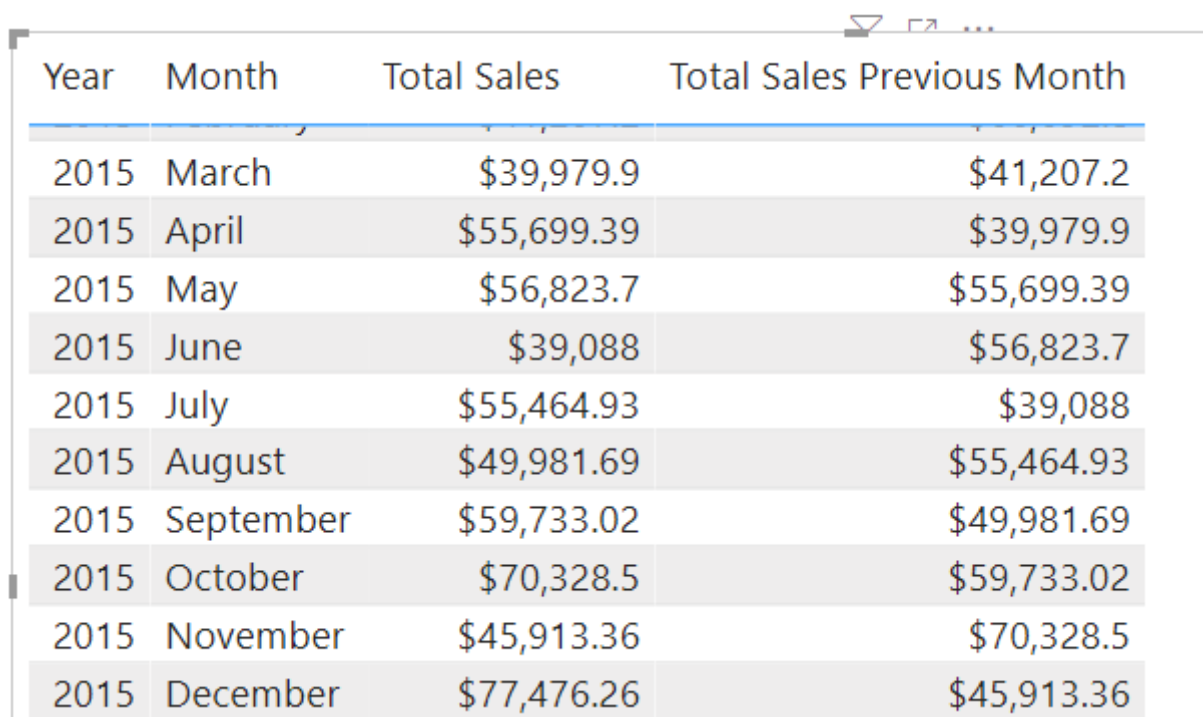
**Comparing your current sales with the sales of a previous time period.**

e.g., See the **total sales of the month next to the total sales of the prior month**

```
Total Sales Previous Month = CALCULATE
(
    SUM('Sales OrderDetails'[Total Price])
    , PREVIOUSMONTH('Dates'[Date])
)
```

This measure **uses the CALCULATE function**, indicating that you're **overriding the context to evaluate this expression the way you want to**

The second argument uses the **PREVIOUSMONTH()** function for the override  
This function **tells Power BI that, no matter what month is the default, the system should override it to be the previous month**



Year	Month	Total Sales	Total Sales Previous Month
2015	March	\$39,979.9	\$41,207.2
2015	April	\$55,699.39	\$39,979.9
2015	May	\$56,823.7	\$55,699.39
2015	June	\$39,088	\$56,823.7
2015	July	\$55,464.93	\$39,088
2015	August	\$49,981.69	\$55,464.93
2015	September	\$59,733.02	\$49,981.69
2015	October	\$70,328.5	\$59,733.02
2015	November	\$45,913.36	\$70,328.5
2015	December	\$77,476.26	\$45,913.36

## **Data Analysis Expressions (DAX) - Code**

- Quantity of Orders YTD = TOTALYTD(SUM('Sales'[OrderQty]), 'Sales'[OrderDate])
- CALENDER or CALENDERAUTO()
  - Calendar Table
  - Can be used to visualize data from each table using a single visual
  - Creates a dimension table that contains a list of dates - one for every day of the calendar of fiscal years
- CALENDERAUTO([FiscalYearEndMonth])
  - bases the START and END date on the MINIMUM and MAXIMUM dates across the entire model

EXCEPT for DATES THAT ARE CALCULATED BY GENERATED COLUMNS OR TABLES OR MEASURES

returns a table with a column of dates - one for each date in the date range

ALWAYS starts with the FIRST DAY OF THE CALENDAR/FISCAL YEAR (e.g., 1/1/2020)  
ENDS on the LAST DAY OF THE CALENDAR/FISCAL YEAR (e.g., 31/12/2020)

Takes a single OPTIONAL ARGUMENT that is - the last month number of a year  
When OMITTED, the value is 12 - December is the last month of the year  
When ENTERED, e.g., 6 - June is the last month of the year

- The columns for just the year

Year = Year(Dates[Date])

- The month number

Month Number = MONTH(Dates[Date])

- The week of the year

Week Number = WEEKNUM(Dates[Date])

- The day of the week

DayOfTheWeek = FORMAT(Dates[Date], "DDDD")

-- A list of DAX date-based queries

- Another date-based DAX

Fiscal Year =

"FY" & YEAR('Date'[Date]) + IF(DATE('Date'[Date]) > 6, 1)

The formula uses the date's year value but adds one to the year value when the month is after June

-- Printing FY 2018 Q1 based off the above query - where the July is THE START OF Q1

Quarter =

'Date'[Year] & " Q"

& IF (

MONTH('Date'[Date]) <= 3,

1,

IF (

MONTH('Date'[Date]) <= 6,

2,

IF (

MONTH('Date'[Date]) <= 9,

3,

4

)

)

)

-- Printing 2017 Jul based off the above two queries

Month = FORMAT('Date'[Date], "yyyy MMM")

-- Computing a numeric value (can also be used as a key) for each year/month combination  
MonthKey = (YEAR('Date'[Date]) \* 100) + MONTH('Date'[Date])

This creates a column containing numeric values of year and month

This can be used to SORT and DEPICT THE DATE in a report while not starting at a specific year/month

-- DISTINCTCOUNT() function

e.g.,

Orders = DISTINCTCOUNT('Sales'[SalesOrderNumber])

The DISTINCTCOUNT function will COUNT ORDERS only ONCE (ignoring duplicates)

-- COUNTROWS() function

e.g.,

Order Lines = COUNTROWS(Sales)

The COUNTROWS function used in the Order Lines measure operates over a table

While the COUNTROWS() is simply the number of table rows (each row is a line of order)

-- HASONEVALUE() function

tests whether a single value in the Salesperson column is filtered.

e.g.,

Target =

IF (

HASONEVALUE('Salesperson (Performance)'[Salesperson]),

SUM('Targets'[TargetAmount])

)

When, true, the expression returns the sum of target amounts (for just that salesperson).

When, false, BLANK is returned

-- CALCULATE() function

used to manipulate the filter context

the first argument takes an expression or a measure (a named expression)

subsequent arguments allow modifying the filter context

think of it as an expression based on a stated conditional

-- an e.g., - always calculate the total sales for 2015, regardless of which year is selected

Sales2015 = CALCULATE (  
SUM('Sales'[Total Price]),  
YEAR('Sales'[Order Date]) = 2015  
)

-- another e.g., - always calculate the total sales for 2020, regardless of which year and sales made in the Asia region

```
Sales2020Asia = CALCULATE (  
    SUM('Sales'[Total Price]),  
    YEAR('Sales'[Order Date]) = 2015,  
    'Region'[Region] = "Asia"  
)
```

-- REMOVEFILTERS() function  
removes active filters

It can take either no arguments, or a table, a column, or multiple columns as its argument.

-- ISINSCOPE() function  
used to test whether the region column is the level in a hierarchy of levels.

basically used to test whether a given column is inside a hierarchy

an example function is displayed below

-- IF() function  
Checks an expression  
When true, it returns the second argument  
When false, it returns the third argument

- RELATED() and RELATEDTABLE() functions  
- RELATEDTABLE()  
a shortcut for CALCULATETABLE() function with no logical expressions  
- CALCULATETABLE()  
evaluates a table expression in a modified filter context and  
returns a table of values

-- TOTALYTD() function  
evaluates the specified expression over the interval which begins on the first day of the year and ends with the last date in the specified date column after applying specified filters

e.g. TOTALYTD function  
Sales YTD = TOTALYTD (  
 SUM('Sales'[Sales]),  
 'Date'[Date],  
 "6-30"  
)

e.g., TOTALYTD function that ends in November 31st  
Sales YtoNovember = TOTALYTD (  
 SUM('Sales'[Sales]),  
 'Date'[Date],  
 "11-31"  
)

e.g., TOTALYTD function that ends in February 28th

```
Sales YtoFebruary = TOTALYTD (
    SUM('Sales'[Sales]),
    'Date'[Date],
    "2-28"
```

- e.g.,

```
Sales YTD = TOTALYTD(SUM('Sales'[Sales]), 'Date'[Date], "6-30")
    in this case the last date of the year is June 30th
```

```
TOTALYTD(Expression, Dates, [Filter], [YearEndDate])
```

hence, the function can take a THIRD OPTIONAL argument representing the last date of a year

An absence here indicates that December 31 is the last date of the year

-- Example Sales Formulas

-- Sales by a Location

```
-- Sales % by All Region = DIVIDE (
    SUM('Sales'[Sales]),
    CALCULATE (
        SUM('Sales'[Sales]),
        REMOVEFILTERS('Region'[Region])
    )
)
```

```
-- Sales % by Country = IF (
    ISINSCOPE('Region'[Region]),
    DIVIDE (
        SUM('Sales'[Sales]),
        CALCULATE (
            SUM('Sales'[Sales]),
            REMOVEFILTERS('Region'[Region])
        )
    )
)
```

-- NOTE: This one is confusing - did not get it to work in the lab

```
-- Sales % by Group = IF (
    ISINSCOPE(Region[Region])
    ISINSCOPE(Region[Country]),
    DIVIDE (
        SUM('Sales'[Sales]),
        CALCULATE (
            SUM('Sales'[Sales]),
            REMOVEFILTERS (
                'Region'[Region],
                'Region'[Country]
            )
        )
    )
)
```



```

    )
)

-- Sales YoY Growth =
VAR SalesPriorYear =
    CALCULATE (
        SUM(Sales[Sales]),
        PARALLELPERIOD (
            'Date'[Date],
            -12,
            MONTH
        )
    )
RETURN
    DIVIDE (
        (SUM('Sales'[Sales]) - SalesPriorYear),
        SalesPriorYear
    )

```

SalesPriorYear is assigned an expression which calculates the sum of the Sales column in a modified context that uses the:

-- PARALLELPERIOD() function to SHIFT THE 12 MONTHS BACK from  
each date in the filter context

this formula DECLARES A VARIABLE

Variables are useful for SIMPLIFYING the formula logic, and more efficient when an expression needs to be evaluated multiple times within the formula

Variables are declared by a unique name

The measure expression must then be output AFTER the RETURN keyword.