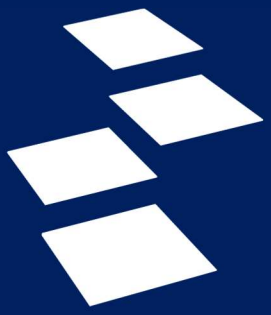
Intermediate DAX

Course Workbook



JOURNEYTEAM

Guide to Workbook

This workbook contains exercises for the JourneyTEAM Intermediate DAX course.

The associated Power BI Desktop (.pbix) and data files with this workbook can be downloaded at http://bit.ly/InDAXcontent.

Save the zipped file *Course Content.zip* to your computer and extract the files.

Once you download and unzip the data you will need to change the data source in Power BI to your local machine. To accomplish this, follow the steps below:

1. In Power BI Desktop, go to *Edit Parameters* in the *Transform Data* dropdown
2. Paste the file path of the folder where you extracted the data
3. Click *OK*
4. Close the window
5. Refresh your data

Each section in this workbook contains step-by-step instruction followed by a solutions subsection called *Check Your Code*. To get the most out of this workbook, wait to look at the *Check Your Code* section until you have tried to write the DAX on your own.

Enjoy!

Advanced Evaluation Context: Modifiers

Below are exercises to test your understanding of the different modifiers that are available to use inside of the CALCULATE function.

Adventure Works needs to know how fast it is fulfilling orders by finding the percentage of sales that are shipped in a given time period compared to the sales that were ordered.

1. Create a measure called *Shipped Sales Amt* that calculates the Sales Amount using the *Shipped Date* instead of the *Order Date* field in the *Sales* table. (Hint: Use USERELATIONSHIP)
2. Create another measure called *Pct Shipped* that calculates the percentage of sales shipped versus ordered in a given time period.

Remember that CALCULATE replaces the filter context from the model with the filter context defined inside of the function. In some cases, we need to use additional DAX code to get the correct numbers.

3. Create a measure called *Accessories Sales* that calculates the sales amount for only the product category accessories. Be sure to use CALCULATE and notice what happens when you place it in the visual.
4. Copy the DAX code from *Accessories Sales* and paste it in a new measure called *Only Accessories Sales* and update the DAX to include KEEPFILTERS inside of CALCULATE. Notice the difference in the behavior.

Sometimes measures return the same value for every value in the visual's row context even when CALCULATE isn't used in the DAX code.

5. Create a measure called *# of Customers* that counts the number of customers in the customer table. Add it to the matrix on the CROSSFILTER page in your report. Notice how the value is the same for each year and month.

Why is this? Looking at the data model reveals the problem. The *Customer* and *Date* tables are both dimension tables with one-to-many relationships to the *Sales* table, our fact table. The filter direction for those one-to-many relationships is unidirectional from the dimension table to the fact table which is best practice for a star schema data model. To get the *Sales*

table to filter the *Customer* table we need to activate a bidirectional filter using DAX. This is done with the modifier CROSSFILTER.

6. Create a new measure called *# of Customers with Sales* and use CROSSFILTER inside of CALCULATE to activate a bidirectional filter between the *Customer* and *Sales* tables.

Adventure Works now wants to know what percentage of customers had a sale in a given time period.

7. Use what you know to create a new measure called *Pct of Customers with Sales* that shows the percentage of customers that had a sale in a given time period.

Bidirectional filters and CROSSFILTER should be used with caution since they can introduce ambiguity into your data model. This exercise has outlined a correct use case of CROSSFILTER with results that make sense and are helpful. Make sure each use of a bidirectional filter or CROSSFILTER is preceded by much thought.

Modifiers inside of CALCULATE alter the filter context defined inside of CALCULATE and allow DAX to activate relationships, simulate many-to-many relationships, and retain the outer filter context.

Check Your Code

Steps 1 & 2

```
Shipped Sales Amt =
CALCULATE (
    [Sales Amount],
    USERELATIONSHIP ( 'Date'[Date], Sales[Ship Date] )
)

Pct Shipped = DIVIDE( [Shipped Sales Amt], [Sales Amount] )
```

Steps 3 & 4

```
Accessories Sales =
CALCULATE(
    [Sales Amount],
    'Product'[Product Category] = "Accessories"
)

Only Accessories Sales =
CALCULATE(
    [Sales Amount],
    KEEPFILTERS( 'Product'[Product Category] = "Accessories" )
)
```

Steps 5, 6, & 7

```
# of Customers = COUNTROWS( Customer )

# of Customers with Sales =
CALCULATE (
    [# of Customers],
    CROSSFILTER ( Sales[Customer ID], Customer[Customer ID], BOTH )
)

Pct of Customers with Sales =
DIVIDE ( [# of Customers with Sales], [# of Customers] )
```

Advanced Evaluation Context: Context Transition

One of the advantages of using CALCULATE and measures in your DAX code is that context transition is enabled automatically. Context transition takes the row context and translates it into a filter context. This is particularly useful for creating calculated columns which only have row context.

Open the *InDAX – 1.30 – Start – Context Transition* Power BI file and notice the two table visualizations with *Product Number* and *Product ID* with the native aggregation on the *Price* column. No DAX was written to obtain this column. We are going to compare our DAX to this so we can see what happens when context transitioning is not activated.

1. Create a calculated column called *Sum Price* that sums the *Product[Price]* column. Note this needs to be done in the *Product* table.

Take a minute to look at the Data view to see what is returned.

2. Create a second calculated column in the Product table called *Calc Sum Price* and add CALCULATE on the outside of the SUM function used in the previous step.

Now in the data view we see the result is quite different. This is because CALCULATE activates context transition. Add the two calculated columns to the table visual with *Product Number*.

The *Context Transition Table* is the Product table with only three columns, *Product ID*, *Color*, and *Price*. Context transition is replaced by the filter context defined inside of CALCULATE and this table will show us a potential issue that arises when we don't have unique rows in the data.

3. Create a calculated column in the *Context Transition Table* called *Sum Price by Product ID* to find the sum of the price column using CALCULATE and adding the modifier ALL on *Product ID*. Then add this calculated column to the table visual with *Product ID*.

Right away we see that the values in the two columns are not equal. Why is this happening? Context transition takes the row context of a table and translates it into a filter context and every column of a table contributes to the row context. Remember that without using CALCULATE the SUM returned the sum of all prices in the Product table. That is because without a filter context the whole *Product* is used for the calculation.

CALCULATE activates context transition, but the ALL modifier then removes the column *Product ID* from the context transition so only *Color*, and *Price* are used to create the filter context and we have products that have the same color and the same price (for example bike frames in different colors). The SUM now is adding the price for each unique color price combination.

Context transition is activated when CALCULATE is used or a measure is referenced in your DAX code. Even when context transition is activated, numbers could be incorrect if the data lacks unique rows.

Check Your Code

Step 1 & 2

```
Sum Price = SUM( 'Product'[Price] )

Calc Sum Price = CALCULATE( SUM( 'Product'[Price] ) )
```

Step 3

```
Sum Price by Product ID =
CALCULATE(
    SUM( 'Context Transition Table'[Price] ),
    ALL( 'Context Transition Table'[Product ID] )
)
```

Hierarchies

Hierarchies can be enhanced with DAX above and beyond dragging and dropping one field on top of another. Standard hierarchies such as Year, Month, Day don't require extra DAX because each column is mutually exclusive. 2012 is not a month or a day and vice versa.

Adventure Works has salespeople who are supervisors, but also have sales themselves. This adds a level of complexity to creating measures that display correctly in visuals. Before we dive into that, let's work with a new function to refine visuals with hierarchies.

1. Create a hierarchy in the *Product* table in this order: category, subcategory, model. Add this hierarchy to the rows in a matrix visual and add *Sales Amount* in the values of the matrix.
2. Create a measure that calculates the percent of total sales the level of the hierarchy is of the total. (Hint: use CALCULATE and ALL)

Notice how this works great. However, Adventure Works wants to see the percent of total as the percent total of the next level in the hierarchy. Category compared to total, subcategory compared to category and so on.

This requires using the function ISINSCOPE in conjunction with several variables. Let us do it.

3. Create a measure called *ISINSCOPE Percent of Total Sales* and create a variable for the Sales Amount, Total Sales Amount, Category Sales Amount, and Subcategory Sales Amount. (Hint: use CALCULATE and ALLSELECTED)

We then need to use nested IF statements since ISINSCOPE returns true if the filter context is in scope and false otherwise.

4. Return an IF statement with ISINSCOPE for the logical statement and work down the hierarchy starting at the bottom of the hierarchy and moving up.

We next move on to working with non-standard hierarchies. Look at the *Salesperson* table in the data view. Each salesperson has an Id. In the *Parent Id* column, the Id of the supervisor of the salesperson is listed. We are going to use some DAX functions to create a hierarchy we can work with.

5. Create a calculated column in the *Salesperson* table called *Parent Path* using the PATH function referencing the *Id* and *Parent Id* columns respectively.

6. Then create another calculated column called *Path Length* also in the *Salesperson* table. This gives us the level of the hierarchy the salesperson is in. Use the PATHLENGTH function referencing the *Parent Path* column.

Now we will create a calculated column for each of the three levels in the hierarchy.

7. Create a calculated column called *Level 1* that looks up the name of the salesperson in the first level of the hierarchy path. Do this using the LOOKUPVALUE and PATHITEM functions.
8. Repeat the previous steps for levels 2 and 3 adding an IF statement to check if the *Parent Path* is greater than or equal to the hierarchy level, returning the next highest level if false. Then create a drag and drop hierarchy in the order of Level 1, Level 2, Level 3.

Time to create our measures. When you add the Sales Hierarchy to a matrix with the Sales Amount measure for the values you will notice that top-level salespeople are repeated. Let us write some DAX to hide those repeats.

9. Create a measure called *Browse Depth* that is a sum of three ISINSCOPE functions, one for each level of the hierarchy.
10. Create another measure called *Max Parent Path* that is the MAX of the *Parent Path* column we created earlier.
11. Create a final measure called *Sales Amount Hierarchy* that is a simple IF statement that checks if the *Browse Depth* is less than or equal to the *Max Parent Path*. If this is true, then we know that the salesperson is in the correct level of the hierarchy. Otherwise return blank.

Add this new measure to another matrix and compare the differences between the two visuals.

Hierarchies make displaying measures tricky, but using ISINSCOPE and parent child DAX formulas allows the data needed to be displayed as wanted.

Check Your Code

Step 2

```
Percent of Total Sales =
VAR Denominator =
CALCULATE(
    [Sales Amount],
    ALL( 'Product' )
)
RETURN
       DIVIDE( [Sales Amount], Denominator )
```

Step 3 & 4

```
INSCOPE Percent of Total Sales =
VAR SalesAmount = [Sales Amount]
VAR Total =
CALCULATE(
    [Sales Amount],
    ALLSELECTED( 'Product'[Product Category] )
)
VAR Category =
CALCULATE(
    [Sales Amount],
    ALLSELECTED( 'Product'[Product Subcategory] )
)
VAR Subcategory =
CALCULATE(
    [Sales Amount],
    ALLSELECTED( 'Product'[Product Model] )
)
RETURN
IF(
    ISINSCOPE( 'Product'[Product Model] ),
    DIVIDE( SalesAmount, Subcategory ),
    IF(
        ISINSCOPE( 'Product'[Product Subcategory] ),
        DIVIDE( SalesAmount, Category ),
        IF(
            ISINSCOPE( 'Product'[Product Category] ),
            DIVIDE( SalesAmount, Total ),
            BLANK()
        )
    )
)
```

Step 5

```
Parent Path = PATH('Salesperson'[Id], 'Salesperson'[Parent Id] )
```

Step 6

```
Path Length = PATHLENGTH( 'Salesperson'[Parent Path] )
```

Step 7

```
Level 1 =
LOOKUPVALUE(
    'Salesperson'[Name],
    'Salesperson'[Id], PATHITEM( 'Salesperson'[Parent Path], 1, INTEGER))
```

Step 8

```
Level 2(3) =
IF(
    PATHLENGTH( 'Salesperson'[Parent Path] ) >= 2(3),
    LOOKUPVALUE(
        'Salesperson'[Name],
        'Salesperson'[Id],
        PATHITEM( 'Salesperson'[Parent Path], 2(3), INTEGER )
    ),
    'Salesperson'[Level 1(2)]
)
```

Step 9

```
Browse Depth = ISINSCOPE( 'Salesperson'[Level 1] ) + ISINSCOPE(
'Salesperson'[Level 2] ) + ISINSCOPE( 'Salesperson'[Level 3] )
```

Step 10

```
Max Parent Path = MAX( 'Salesperson'[Path Length] )
```

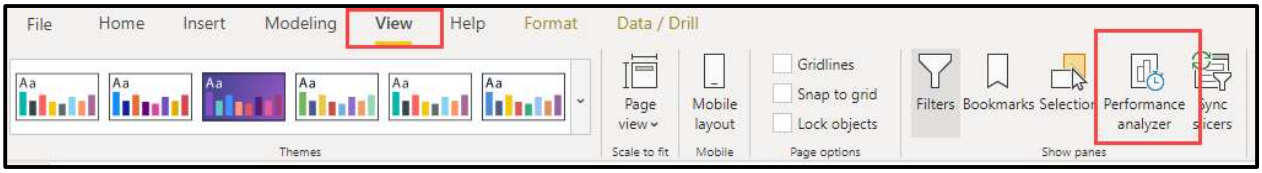
Step 11

```
Sales Amount Hierarchy =
IF(
    [Browse Depth] <= [Max Parent Path],
    [Sales Amount],
    BLANK()
)
```

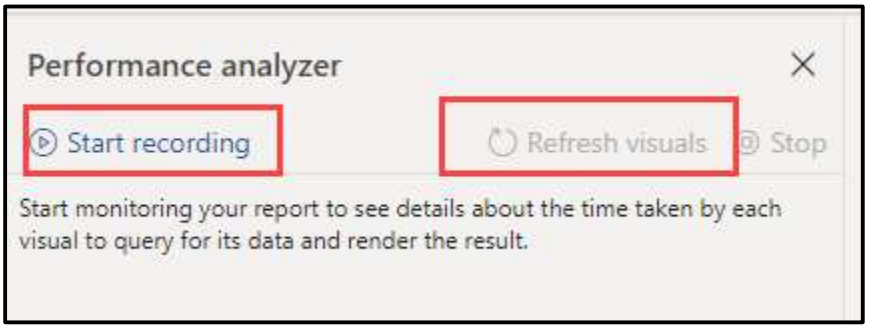
Querying

DAX is a query language, but the code we have been writing is not exactly the query language that runs in the background. Using the performance analyzer in Power BI we can find the DAX query that runs when visuals refresh in a report.

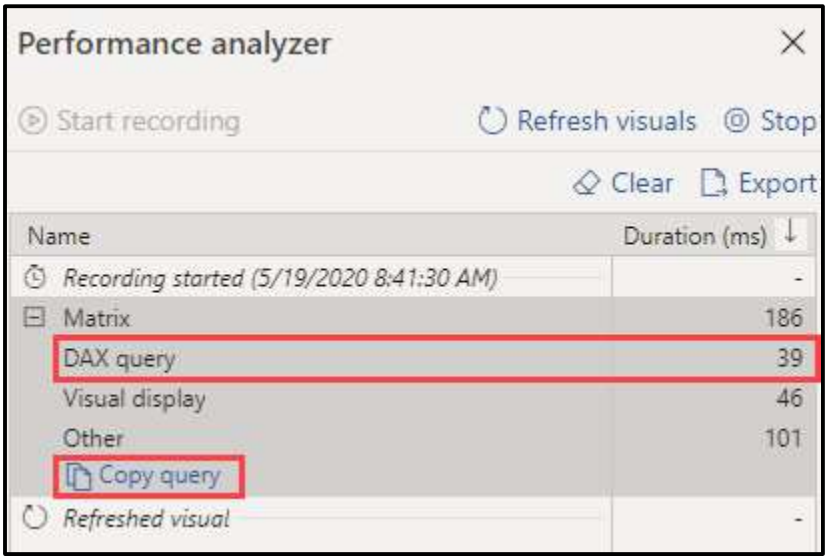
Go to the *View* tab in the top ribbon and click on the *Performance analyzer* icon.



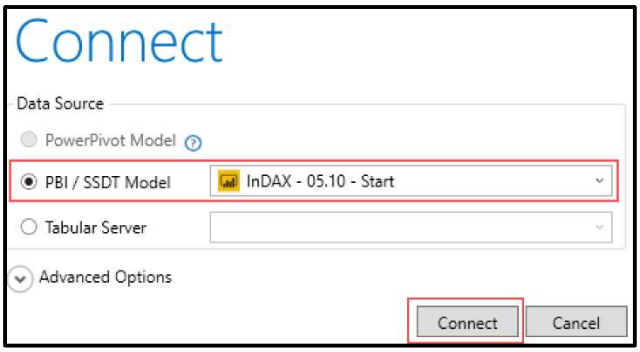
Make sure you are on a page in your report that has visualizations, click *Start recording* and then click *Refresh visuals*.



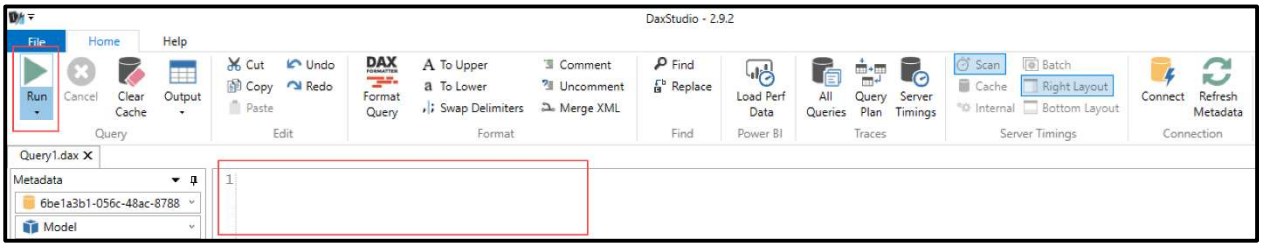
You will then see how long in milliseconds each visual took to refresh and the details about how long the DAX query, visual display, and other items took. Click *Copy query* and paste it in notepad.



Now we are ready to download DAX Studio. Go to https://daxstudio.org/ and download DAX Studio on your computer. Once it is installed, launch DAX Studio and connect to the *InDAX – 5.10 – Start* Power BI file by selecting it from the drop-down menu next to PBI / SSDT Model. Then click *Connect*.



Once you have connected to your Power BI file, copy and paste the DAX query you have in notepad into the query editor and click *Run*.



You will see the table that is generated by the DAX query in the output section below the query editor. This table is the data used in the visual you copied the query from and DAX queries like this one are being run every time your visuals refresh. In the next exercises we are going to learn how to write several DAX queries, so you understand how it differs from the DAX code we have been writing.

*Remember that each step should be run separately from the others. You can do this by highlighting the section of code you want to run and then clicking the Run button.

1. Select the *Sales* table from your model in DAX Studio.
2. Select the *Sales* table sorted by *Order Date* and *Line Total* in DAX Studio
3. Now we need to filter the table we select. Filter the *Sales* table where the product category is Bikes.

FILTER is great, and that is what we should use in DAX, but for querying there is a better function.

4. Write the same query using CALCULATETABLE instead of FILTER (Hint: We don't need to use RELATED).

CALCULATETABLE does have limitations. It has no context so measures or related columns cannot be used as a filter. In this case, the filter function needs to be used.

5. Write a filter on the *Product* table that selects only the products that have sales over $500.
6. Now we will select columns from a table and add expressions to our columns. Select *Order Date*, *Quantity*, *Unit Price*, *Revenue*, *Discount*, and *Line Total* from the *Sales* table. Remember to use the SELECTCOLUMNS function.
7. We can also add columns with expressions to a table. Add the *Product Category* to the *Sales* table.
8. Now we will select columns from a table and add expressions to our columns. Select the *Order Date*, sum of *Quantity*, and sum of *Line Total* from the *Sales* table. Remember to use the SUMMARIZE function.

We can also use SUMMARIZECOLUMNS, but it is more complex and not as commonly used. We will review what it can do here, but don't worry about memorizing this syntax.

9. Create the same table as in Step 8 using SUMMARIZECOLUMNS instead of SUMMARIZE.
10. We can also include a table as a filter in SUMMARIZECOLUMNS. Add an argument before the *Quantity* column that filters *Sales[Quantity]* to be greater than 5 products. Then update the function from Step 9 to include that filter.

DAX queries run in the background of your model and return the tables that each visualization uses. The more DAX queries a visualization requires, the longer it will take your visualization to render.

Check Your Code

All solutions for this section can be found in the text file in the Querying solutions folder.

05 – Querying\Solutions\InDAX – 5.10 - Solutions

Table Functions: Table Variables

Now that we have learned new DAX functions and theory let us put it into practice by adding to the Table Functions we learned in Fundamentals of DAX.

First, we will review using variables in DAX to clean up the code. Variables can be tables and the same rules apply to table variables as single value variables; constant once defined, can access the outer row context, can reference predefined variables within single DAX code.

We will be combining time intelligence with table variables to create two measures so we can compare sales in the last 30 days to sales from previous 31 to 60 days. Let's get started.

1. Create a calculated table called *MaxSelectedTable Table* that filters the *Date* table down to one row and one column equal to the measure *Max Selected Date*. (Hint: Use a variable to define the *Max Selected Date* measure)

Check your data view to make sure you have a one row one column table with today's date.

2. Create another calculated table called *DatesIncluded Table* that take the code you wrote in step one and adds another variable that returns a one column table with a row for every day in the last month. (Hint: Use the DATESINPERIOD function)

Now that we have these table created, we can incorporate them into a measure. Note that we created these tables only to see what is returned. We would not create these tables in our data model otherwise.

3. Create a measure called *Selected Month Sales* that takes the code from step 2 and instead of returning the variable *DatesIndluded* uses that variable inside of CALCULATE to get the sales for the last month.

Advanced Exercise:

4. Create a measure that calculates the sales for the last 31 to 60 days. (Hint: Start with the code from step 3 and adjust it)

Variables can be tables. This is especially valuable because tables can be used in the filter arguments of CALCULATE.

Check Your Code

Step 1

```
MaxSelectedTable Table =
VAR MaxSelectedDate = [Max Selected Date]
VAR MaxSelectedTable =
      FILTER( VALUES( 'Date'[Date] ), 'Date'[Date] = MaxSelectedDate )
RETURN
      MaxSelectedTable
```

Step 2

```
DatesIncluded Table =
VAR MaxSelectedDate = [Max Selected Date]
VAR MaxSelectedTable =
      FILTER( VALUES( 'Date'[Date] ), 'Date'[Date] = MaxSelectedDate )
VAR DatesIncluded =
      DATESINPERIOD( 'Date'[Date], MaxSelectedTable, -1, MONTH )
RETURN
      DatesIncluded
```

Step 3

```
Selected Month Sales =
VAR MaxSelectedDate = [Max Selected Date]
VAR MaxSelectedTable =
      FILTER( VALUES( 'Date'[Date] ), 'Date'[Date] = MaxSelectedDate )
VAR DatesIncluded =
      DATESINPERIOD( 'Date'[Date], MaxSelectedTable, -1, MONTH )
RETURN
CALCULATE(
    [Sales Amount],
    DatesIncluded
)
```

Step 4

```
Previous Month Sales =
VAR MaxSelectedDate = [Max Selected Date]
VAR MaxSelectedTable = FILTER( VALUES( 'Date'[Date] ), 'Date'[Date] = MaxSelectedDate )
VAR EndDay = DATEADD( MaxSelectedTable, -1, MONTH)
VAR StartDay = DATEADD( MaxSelectedTable, -2, MONTH)
VAR DatesInlcuded = DATESBETWEEN( 'Date'[Date], StartDay, EndDay )
RETURN
CALCULATE(
    [Sales Amount],
    DatesInlcuded
)
```

Table Functions: ISEMPTY & HASONEVALUE

Now that we have used tables as variables, we can add DAX to check if a table is empty and return different values based on the result.

1. Create a measure called *Readable Sales* that returns "This customer has bought X products" where X is the number of products the customer has purchased. If the customer hasn't purchased any products have X equal 0.

Customers 30119 and 30120 should have no products. The ISEMPTY function requires a table as an input. ISBLANK should be used for checking columns for empty or blank values. Note that ISEMPTY only returns TRUE if the input has no rows or columns, not if the table only contains a blank value.

Our next exercise builds on creating an interactive slicer to change the calculation of the Sales Amount.

2. Add a slicer to the page with the *Value* column from the *Discount* table.
3. Create a measure called *Discounted Sales 1* that returns the discounted sales amount when a discount is selected in the slicer. Use the HASONEVALUE function.

This code works beautifully, but it is a little long. Recall from the Fundamentals course that we can use SELECTEDVALUE to simplify our code.

4. Create another measure called *Discounted Sales 2* that does the same as step 3 but uses SELECTEDVALUE instead.

Notice that the DAX code is much simpler, and the functionality is the same. The performance analyzer shows that each DAX query takes the same time.

Use ISEMPTY to check if a table contains no rows or columns. ISBLANK is to check if a value in a column is blank or empty.

Check Your Code

Step 1

```
Readable Sales =
VAR ProductCount = VALUES( Sales[Product ID] )
VAR Result =
IF(
    ISEMPTY( ProductCount ),
    "This customer bought no products",
    "This customer bought " & COUNTROWS( ProductCount ) & " products"
)
RETURN
    Result
```

Step 3

```
Discounted Sales 1 =
VAR OneValue = HASONEVALUE( Discount[Value] )
VAR DiscountValue = 1 - MAX( Discount[Value] )
VAR Result =
    IF(
        OneValue,
        [Sales Amount] * DiscountValue,
        [Sales Amount]
    )
RETURN
    Result
```

Step 4

```
Discounted Sales 2 =
VAR DiscountValue = SELECTEDVALUE( Discount[Value], 0 )
VAR Result = [Sales Amount] * ( 1 - DiscountValue )
RETURN
    Result
```

Table Functions: Optimization

Table functions take tables as inputs and accordingly many users use a single table reference in their DAX. This practice will often return the correct numbers, but it could cause your DAX to run slowly.

Functions like VALUES return a subset of the data that can speed up your DAX queries as we will see now.

1. Create a measure called *Order 74403*, I know; creative, that calculates the sales amount for order 74403. Include the table reference to the *Sales* table in this measure.
2. Create a second measure called *Order 74403 Opt* that is the same code as step 1 but adds the VALUES function around the *Sales* table reference. You will also need to include a column reference. Which column do you think should be inside of VALUES?
3. Open the performance analyzer, start recording, and refresh your visuals. Notice that the optimized code is faster. Not by much because of the small dataset, but still faster.

Including an entire table in a table function requires DAX to scan the entire table which can slow down your queries. Use table functions like VALUES to optimize your DAX.

Check Your Code

Step 1

```
Order 74403 =
SUMX (
    FILTER (
        Sales,
        Sales[Order ID] = 74403
    ),
    [Sales Amount]
)
```

Step 2

```
Order 74403 Opt =
SUMX (
    FILTER (
        VALUES ( Sales[Order ID] ),
        Sales[Order ID] = 74403
    ),
    [Sales Amount]
)
```

Iterators: RANKX

RANKX was introduced in the Fundamentals course and we are going to take it a step farther here with the additional functions we have been introduced to.

1. First, create a *Sales Rank Base* measure that ranks the sales people by their sales.

When we add this measure to the visual notice that the total has rank 1. While that technically is true, it isn't useful information. Let's use what we have learned to get rid of the value in the totals area.

2. Create another measure called *Sales Ranking* that uses INSCOPE to omit the rank for the totals area. (Hint: Use an IF statement)

Ranks and percentages are the most common use cases for this scenario but get creative.

Advanced Exercise:

3. Create a rank measure that ranks the salespeople by sales that had 10 or more products. Make sure the measure doesn't return a value for the total.

ISINSCOPE can be used to remove values from your visuals that are meaningless. This helps avoid confusion for your end users.

Check Your Code

Step 1

```
Sales Rank Base =
RANKX(
    ALL( 'Sales Person' ),
    [Sales Amount]
)
```

Step 2

```
Sales Ranking =
IF (
    ISINSCOPE ( 'Sales Person'[Name] ),
    RANKX ( ALL ( 'Sales Person' ), [Sales Amount] )
)
```

Step 3

```
Large Sales Ranking =
IF (
    ISINSCOPE ( 'Sales Person'[Name] ),
    RANKX (
        ALL ( 'Sales Person'[Name] ),
        CALCULATE ( [Sales Amount], Sales[Quantity] >= 10 )
    )
)
```

Iterators: Context Transition

Earlier we created calculated columns to visualize what happens when context transition is not activated. Now we will do it using a measure.

1. First, create a *Net Sales* measure that sums the *Line Total* column in the *Sales* table.
2. Then create a *Sales Last Quarter* measure that calculates the net sales for the last quarter using the SUM function of *Line Total*. Use what we reviewed in the table variables section for this measure.

Check your code to make sure it matches the solution.

3. Create another measure that is the same as step 2, but include the *Net Sales* measure this time.
4. Place both measures in the matrix visual.

Notice how the numbers are different for each measure and the only difference is the use of a measure inside of the DAX. Remember that measures are all surrounded by an invisible CALCULATE so context transition is activated, and the correct numbers are returned.

Best practice is to create basic measures and then reference those measures in other more complicated measures that require filtering by time or other dimensions.

Check Your Code

Step 1

```
Net Sales = SUM( Sales[Line Total] )
```

Step 2

```
Sales Last Quarter =
VAR Dates = DATESINPERIOD( 'Date'[Date], MAX( Sales[Order Date] ), -1, QUARTER )
VAR Result =
CALCULATE (
    SUMX( Sales, SUM( Sales[Line Total] ) ),
    Dates
)
RETURN
    Result
```

Step 3

```
Sales Last Quarter M =
VAR Dates = DATESINPERIOD( 'Date'[Date], MAX( Sales[Order Date] ), -1, QUARTER )
VAR Result =
CALCULATE (
    SUMX( Sales, [Net Sales] ),
    Dates
)
RETURN
    Result
```

Iterators: CONCATENATEX

Adventure Works has come to you with a request to see the top three companies they sell to by year, month, and day with the percent of total sales that company had. They want to also have the company names displayed next to the percent of total sales.

How would we go about doing this? We know that we need to return a string that combines the company name with the percent of total, but what function(s) can do that? CONCATENATE is great for combining values into a single string, but it concatenates all values entered. We need to return the top three companies, which requires iterating over all companies. Luckily, we have an iterator version of CONCATENATE called, you guessed it, CONCATENATEX.

This measure takes some work. We will take it step by step.

1. Create a measure called *Top 3 Companies* and add a variable that is equal to the *Sales Amount* measure. This will give us the total Sales Amount in the context of the visual which is what we need.
2. Now we need to create a table to iterate over, that contains the values we want to return. We will use ADDCOLUMNS to add columns to a table of unique companies, and VALUES will do the trick. Add a column for Total Sales, Sales, Perc, and Result.

During these steps, please reference the DAX in the Check Your Code section below.

3. We now are ready to write the CONCATENATX inside of an IF statement. We use the IF statement to only show data if we have positive sales. We want to select the top 3 from the table variable we defined in step 2 and include the Result and Sales columns so we can display the Result and sort by the Sales.

Check your code.

4. Add this measure to the matrix visual and see what it returns. Take a moment to change the sort order to see how the visual changes.

Stakeholders often have unique requirements for how they want data displayed. CONCATENATEX can get you to where you need to go.

Check Your Code

Step 1

```
Top 3 Companies =
VAR Totalsales = [Sales Amount]
```

Step 2

```
Top 3 Companies =
VAR Totalsales = [Sales Amount]
VAR CompanySales =
    ADDCOLUMNS(
        VALUES( Customer[Company Name] ),
        "Total Sales", Totalsales,
        "Sales", [Sales Amount],
        "Perc", DIVIDE( [Sales Amount], Totalsales ),
        "Result", Customer[Company Name] & " ( " & FORMAT( DIVIDE( [Sales Amount],
Totalsales ), "0.00%" ) & " )"
    )
RETURN
```

Step 3

```
Top 3 Companies =
VAR Totalsales = [Sales Amount]
VAR CompanySales =
    ADDCOLUMNS(
        VALUES( Customer[Company Name] ),
        "Total Sales", Totalsales,
        "Sales", [Sales Amount],
        "Perc", DIVIDE( [Sales Amount], Totalsales ),
        "Result", Customer[Company Name] & " ( " & FORMAT( DIVIDE( [Sales Amount],
Totalsales ), "0.00%" ) & " )"
    )
RETURN
    IF (
        Totalsales > 0,
        CONCATENATEX(
            SELECTCOLUMNS(
                TOPN( 3, CompanySales, [Sales], DESC ),
                "Result", [Result],
                "Sales", [Sales]
            ),
            [Result],
            CONCATENATE ( ",", UNICHAR ( 10 ) ),
            [Sales],
            DESC
        )
    )
```

Time Intelligence: FIRSTDATE

Time intelligence functions allow manipulation of data using date periods. For the most part, these functions allow you to manipulate the filter context of the data to allow for periodic insight.

Adventure Works wants to do some analysis on their customers and how often they purchase. They want to know the date of each customer's first and last purchase along with the amounts for the first and last purchase.

1. Create a measure called *First Purchase* that finds the sales amount of the first purchase a customer made.
2. Create another measure called *Last Purchase* that finds the sales amount of the last purchase a customer made.

Notice the values and the total. Most customers have purchased more in their last purchase which is good for business. We can identify which customers have purchased less and focus marketing efforts on them.

3. Now create two measures to calculate the date of the first and last purchase for each customer.
4. Add another measure called *Days Since Last Purchase* that finds how many days it has been since the customer made a purchase.

For those customers who have not purchased anything for 30 or more days we can send them a promo code to incentivize a purchase.

Time intelligence functions allow for a broad range of analysis. While a function doesn't exist for every scenario, many functions are available to streamline and simplify your DAX measures.

Check Your Code

Step 1

```
First Purchase =
CALCULATE(
    [Sales Amount],
    FIRSTDATE( Sales[Order Date] )
)
```

Step 2

```
First Purchase =
CALCULATE(
    [Sales Amount],
    LASTDATE( Sales[Order Date] )
)
```

Step 3

```
First Purchase Date = FIRSTDATE( Sales[Order Date] )

Last Purchase Date = LASTDATE( Sales[Order Date] )
```

Step 4

```
Days Since Last Purchase =
DATEDIFF(
    [Last Purchase Date],
    TODAY(),
    DAY
)
```

Time Intelligence: SAMEPERIODLASTYEAR

Year-over-year calculations are a common analysis method and DAX makes these calculations extremely easy with the time intelligence function SAMEPERIODLASTYEAR. Note that this function only works for year-over-year calculations and not month-over-month or quarter-over-quarter.

Your manager wants to see how sales have changed year-over-year. Let's see how easy you could do it with DAX.

1. Create a measure that calculates the sales amount for the same period last year.

Verify that your measure is doing what is expected by adding it to the matrix visual.

2. Create another measure called *YOY %* that calculates the year-over-year change in sales. (Hint: the formula is (current year – last year) / last year)

You can shift the dates a measure uses back one year by adding SAMEPERIODLASTYEAR as a filter argument in CALCULATE. It is best practice to use a date table to unify your data model and DAX.

Check Your Code

Step 1

```
Sales Last Year =
CALCULATE ( Sales[Sales Amount], SAMEPERIODLASTYEAR ( 'Date'[Date] ) )
```

Step 2

```
YOY% =
VAR ThisYSales = Sales[Sales Amount]
VAR LastYSales = Sales[Sales Last Year]
VAR Difference = ThisYSales - LastYSales
RETURN
DIVIDE( Difference, LastYSales, 0 )
```

Time Intelligence: TOTALYTD

DAX also has time intelligence functions for year-to-date calculations. We can also combine SAMPERIODLASTYEAR with our year-to-date calculations to enhance our report.

We will also work with a fiscal calendar in this section to show how the DAX changes when using a fiscal calendar.

1. Create a measure called *Total YTD Bike Sales* that calculates the sales year-to-date in the bike category. (Hint: You will want to use TOTALYTD)
2. Create another measure called *Last YTD Bike Sales* that calculates the measure from step 1 but for the previous year.
3. Then create a measure called *YOY YTD %* that calculates the percent change in the year-to-date sales year-over-year.
4. Add all measures to the matrix visual, in addition to the *Bike Sales* measure, to verify your DAX is calculating correctly.

The *Fiscal Date* table is a standard date table with a few key differences. It starts on July 1st and ends on June 30th. The hierarchy contains a fiscal year and fiscal month level. This is important for display purposes. Measure that are created using fiscal year arguments will display incorrectly if used in a visual with a standard date hierarchy.

5. Create a measure called *Total FYTD Bike Sales* that calculates the year-to-date sales amount for the fiscal year. (Hint: You will need to use CALCULATE and DATESYTD with a fiscal year argument)

Remember, time intelligence functions can be combined to enhance your analysis and fiscal years require additional columns to display measures correctly.

Check Your Code

Step 1

```
Total YTD Bike Sales =
TOTALYTD (
    [Sales Amount],
    'Date'[Date],
    'Product'[Product Category] = "Bikes"
)
```

Step 2

```
Last YTD Bike Sales =
CALCULATE(
    [Total YTD Bike Sales],
    SAMEPERIODLASTYEAR( 'Date'[Date] )
)
```

Step 3

```
YOY YTD % =
VAR LastYearsSales = [Last YTD Bike Sales]
VAR ThisYearSales = [Total YTD Bike Sales]
VAR Difference = ThisYearSales - LastYearsSales
RETURN
    DIVIDE( Difference, LastYearsSales )
```

Step 5

```
Total FYTD Bike Sales =
CALCULATE (
    [Sales Amount],
    DATESYTD ( 'Fiscal Date'[Date], "06/30" ),
    'Product'[Product Category] = "Bikes"
)
```

Notes

Other Classes Taught by JourneyTEAM

Power BI

- Fundamentals of DAX
- Intermediate DAX
- Advanced DAX
- Dashboard in a Day
- Advanced Dashboard in a Day
- Admin in a Day
- Advanced Visualizations

Power Automate

- Flow in a Day
- Advanced Flow in a Day

Power Apps

- Power Apps in a Day
- Advanced Power Apps in a Day

Power Virtual Agent

- Virtual Agent in a Day


JOURNEYTEAM