Advanced DAX

Course Workbook

JOURNEYTEAM

Guide to Workbook

This workbook contains exercises for the JourneyTEAM Advanced DAX course.

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

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!

Querying in DAX

Here we will add onto the DAX querying we learned in the intermediate DAX course. Again, the steps here are all outlined in the *AdDAX – 01.10 – Solution – Querying.txt* file for your convenience. I would resist copying and pasting the code though because as you type it you will become more familiar with it. And as always, please try to write the code on your own first and feel free to use online resources like dax.guide for syntax questions.

1. Use CROSSJOIN to find the sales amount and number of sales for each product category and color category. (Hint: Use ADDCOLUMS)
2. Select the top 3 colors by country by sales amount. (Hint: Use GENERATE and TOPN)

   We now will look at UNION, INTERSECT, and EXCEPT. We first need to create a few table variables to compare. Here is the code:

```
EVALUATE
        VAR RedBlack =
                CALCULATETABLE(
                        VALUES( 'Product'[Color] ),
                        'Product'[Color] IN { "Red", "Black" }
                )
        VAR WhiteBlack =
                CALCULATETABLE(
                        VALUES( 'Product'[Color] ),
                        'Product'[Color] IN { "White", "Black" }
                )
        VAR RedBlackDAX = { "Red", "Black" }
```

3. Use ADDCOLUMNS to create three tables using UNION, INTERSECT, and EXCEPT with the RedBlack and WhiteBlack table variables as inputs and sales in the additional column.
   a. Once you have done that, switch the RedBlack table variable for the RedBlackDAX table variable and see what happens.

Let’s move on. SUMMARIZE and SUMMARIZECOLUMNS have trouble when you reference columns from a table variable. Context transition is turned off and no data lineage exists. We have another function called GROUPBY that will allow us to aggregate our data and preserve context transition from a table variable. We need to first create a few table variables:

```
EVALUATE
      VAR AgeTable =
            ADDCOLUMNS(
                  VALUES( Customer[BirthDate] ),
                  "Age", INT( ( TODAY() - Customer[BirthDate] ) / 365.25 )
                  )
      VAR AgeGroups =
            ADDCOLUMNS(
                  AgeTable,
                  "Age Group",
                        SWITCH(
                              TRUE(),
                              [Age] <= 25, "18 - 25",
                              [Age] <= 45, "26 - 45",
                              [Age] <= 65, "46 - 65",
                              "66+"
                        ),
                  "Sales", [Sales Amount]
            )
      RETURN
            AgeGroups
```

4. We now are ready to summarize sales by AgeGroups. Give it a go with ADDCOLUMNS and SUMMARIZE. You will get an error. Then use GROUPBY.

It is time to combine writing DAX queries in DAX Studio with the actual queries our model uses. This includes troubleshooting measures within DAX Studio.

DAX Studio allows you to define measures that only live within the query and are not part of the Power BI data model. DEFINE and MEASURE are used to retrieve and create measures for our query. Find the [Sales Amount] measure and right click it, then click *Define measure*. Notice what is returned is the DAX for your sales amount measure.

Now we need to get the DAX query for the visual in our report. You can use the performance analyzer to copy the query, or just use the code below:

```
// DAX Query

   EVALUATE

         TOPN(

               502,

               SUMMARIZECOLUMNS(

                     ROLLUPADDISSUBTOTAL( 'Product Category'[Product Category],
                     "IsGrandTotalRowTotal" ),

                     "Sales_Amount", 'Measures Table'[Sales Amount],

                     "Percent_of_Total", 'Measures Table'[Percent of Total]

               ),

               [IsGrandTotalRowTotal],

               0,

               'Product Category'[Product Category],

               1

         )


         ORDER BY

               [IsGrandTotalRowTotal] DESC, 'Product Category'[Product Category]
```

5. Now let us define the measures used in this visual by searching for [Percent of Total] and right clicking then selecting *Define dependent measures*. Notice that the measures that Percent of Total depends on are also defined. Then change the defined measure [Sales Amount] to multiply by the discount and see how the numbers change.

Now measures can be modified within the DAX query, perfected, and then transferred back to Power BI. When you see numbers that don't match what you expect, use DAX Studio to troubleshoot.

Querying in DAX is a great way to troubleshoot measures and see how changes affect the results. You can then copy the code from DAX Studio straight into a measure in your Power BI report.

Check Your Code

```
Step 1 - 5
See file AdDAX – 01.10 – Solution – Querying.txt for the DAX code
```

Data Lineage

Stay in DAX Studio as we explore more about data lineage. Remember, data lineage is the connection to the data model inherited by referencing a column or table in the model. While a column contains string, or numbers, it is more than just the data type because the reference to a column retains a connection to the data model. Most of the time.

1. Use ADDCOLUMNS to create a table with *Product Categories* and *Sales Amount* and give the *Product Categories* column a new name like "New Name."
2. Now add a blank string to the end of the *Product Categories* column expression and see what happens.

Data lineage breaks as soon as the expression contains more than a column reference. Any function or operator added to the *Product Categories* column will break the data lineage.

The functions UNION, INTERSECT, and EXCEPT we used earlier preserve data lineage if the first table argument contains column references to the data model. That is why including the RedBlackDAX table variable in the first argument resulted in the wrong numbers.

Data lineage isn't preserved inside of an ADDCOLUMNS function either. Let's define two of the table variables we did in the last section. Here is the code:

```
EVALUATE
    VAR RedBlack =
        CALCULATETABLE(
            VALUES( 'Product'[Color] ),
            'Product'[Color] IN { "Red", "Black" }
        )
    VAR RedBlackDAX = { "Red", "Black" }
```

3. Create a table using ADDCOLUMNS to show the sales amount for red and black products using the RedBlack table variable. Then again using the RedBlackDAX table variable.

Using DAX Studio is a great way to visualize the result of DAX queries, but it isn't necessary in some cases. We can simply create a calculated table in our Power BI report to see the results.

4. Create a calculated table in Power BI that uses variables to define a table of product categories and then add a column for sales. Use this code for your table variable:

   ```
   {"Bikes", "Clothing", "Components", "Accessories" }
   ```

5. Now use TREATAS around your array of product categories and see how the table changes. Then compare it to simply using the VALUES function on *Product Category*.

Think about the filter context, then compare your DAX with the solution in the Check Your Code section.

Data lineage is the connection of column and table references to the data model. TREATAS can restore data lineage that has been broken.

Check Your Code

Step 1 - 3

See file *AdDAX - 02.10 - Solution - Data Lineage.txt* for the DAX code

Step 4 - 5

```
Break Data Lineage =
VAR ProductCategories = {"Bikes", "Clothing", "Components", "Accessories" }
VAR TreatAsProductCategories =
       TREATAS(
              {"Bikes", "Clothing", "Components", "Accessories" },
              'Product Category'[Product Category]
       )
VAR RealProductCategories = VALUES( 'Product Category'[Product Category] )
VAR Result =
    ADDCOLUMNS(
        RealProductCategories,
        "Sales", ROUND( [Sales Amount], 2 )
    )
RETURN
    Result
```

Expanded Tables

A data model may look like disparate tables connected by relationships and on the outside that is what it is, but the UI doesn't tell the whole story. Under the hood a data model contains expanded tables. The expanded table of the fact table in a star schema is the whole data model. We will see how expanded tables factor into DAX measures along with why you may want to or may not want to use them.

In this exercose we will be counting the number of product subcategories and see how the measures change when different slicers are used.

1. Create a measure called *Count of Subcategory* that counts the number of subcategories.

When the slicers of product category and product subcategory are selected, the number changes correctly. But when the product name is selected, the number doesn't change. We can see why by looking at the data model. The *Product* table can't filter the *Product Subcategory* table unless we use a bidirectional filter which is not recommended.

2. Create a measure called *Expanded Table* to activate expanded tables by using CALCULATE with the previous measure and the product table in the filter argument.

The product name slicer now correctly filters the *Expanded Table* measure.

3. Expanded tables are not activated with only a column reference. To see this, create another measure that uses VALUES with product name inside of a CALCULATE.

Think about the filter context, then compare your DAX with the solution in the Check Your Code section.

Each table in a data model has an expanded table. The outermost dimension tables are their own expanded table whereas the fact table often include most of if not the whole data model.

Check Your Code

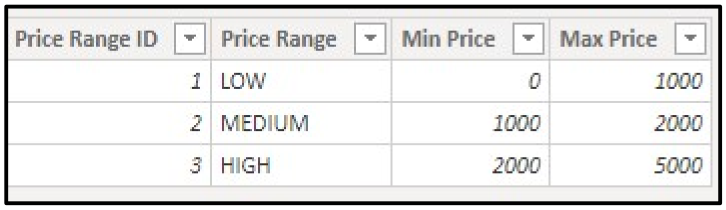
```
Step 1

Count of Subcategory = COUNTROWS('Product SubCategory')

Step 2

Expanded Table =
CALCULATE(
    [Count of Subcategory],
    'Product'
)

Step 3

Column Example =
CALCULATE(
    [Count of Subcategory],
    VALUES( 'Product'[Name] )
)
```

Segmentation

Static Segmentation

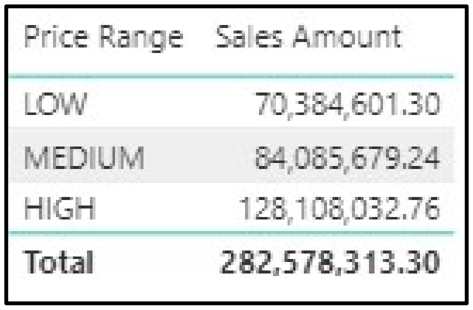
Static segmentation happens when we categorize a table on a static field like product price that does not change across dimensions like time or customer. Let us work on the next exercise to better understand this concept.

1. Create the following table called *Price Range* using the enter data functionality:



Price Range ID	Price Range	Min Price	Max Price
1	LOW	0	1000
2	MEDIUM	1000	2000
3	HIGH	2000	5000

2. Create a calculated column in the Product table called *Price Range ID*. We want this column to categorize Price based on the Price Range from our new table. (Hint: Use a variable to filter the *Price Range* table and don't forget to use DISTINCT)
3. Now create a relationship between the *Price Range* and *Product* tables based on the new calculated column created in step 2.
4. Create a visual that will display *Price Range* and *Sales Amount*.



Price Range	Sales Amount
LOW	70,384,601.30
MEDIUM	84,085,679.24
HIGH	128,108,032.76
Total	**282,578,313.30**

The category cutoffs do not change here either unless defined by a calculated column. In that case, you may run into circular dependencies which is resolved by using DISTINCT instead of VALUES.

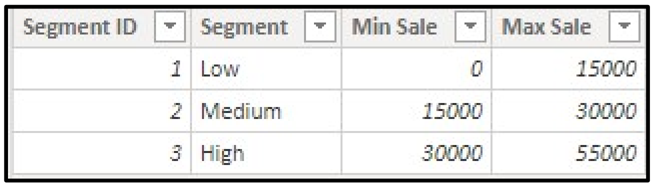
Check Your Code

```
Step 1

   -  Select Enter data in the top ribbon
   -  Enter the data into each column and row

Step 2

Price Range ID =
VAR Filters =
    FILTER (
        'Price Range',
        'Price Range'[Min Price] <= 'Product'[Price]
            && 'Price Range'[Max Price] > 'Product'[Price]
    )
RETURN
    CALCULATE (
       DISTINCT ( 'Price Range'[Price Range ID] ),
       Filters
    )
```

Dynamic Segmentation

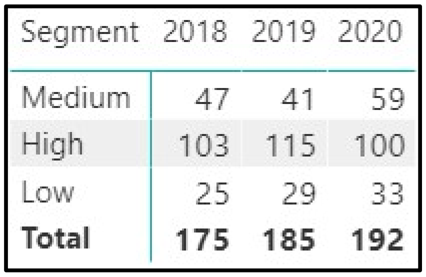
With Dynamic Segmentation we cannot have a relationship between our tables because the segment changes over time or across your other fields. Think of the sales of a customer being high in one year, then low the next year.

1. Create a table called *Customer Segment* by manually entering data. Your table should look like this:



Segment ID	Segment	Min Sale	Max Sale
1	Low	0	15000
2	Medium	15000	30000
3	High	30000	55000

2. Now create a measure called *# of Customers in Segment* that will count the number of customers that had sales in the past years (Hint: Use SUMX and variables to define the min and max values). Your final matrix visual should look like this:



Segment	2018	2019	2020
Medium	47	41	59
High	103	115	100
Low	25	29	33
Total	**175**	**185**	**192**

Dynamic segmentation isn't possible without DAX and this DAX does the trick while keeping your model running as efficiently as possible.

Relationships will not work for dynamic segmentation because the category changes over time, or another dimension. DAX saves the day.

Check your Code

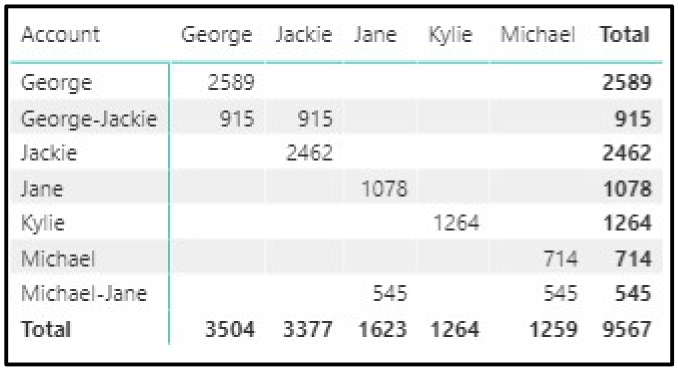
```
Step 1

   -  Select Enter data in the top ribbon
   -  Enter the data into each column and row

Step 2

# of Customers in Segment =
SUMX (
    'Customer Segment',
    VAR segmin = 'Customer Segment'[Min Sale]
    VAR segmax = 'Customer Segment'[Max Sale]
    RETURN
        COUNTROWS (
            FILTER (
                Customer,
                [Sales Amount] > segmin && [Sales Amount] <= segmax
            )
        )
)
```

Advanced Relationships

Many-to-Many Relationships

Best practice is not to use many-to-many or bidirectional relationships in your model, but when you do need to use them there are ways to work around them to make sure your measures are accurate. You do not need to create a many-to-many relationship in your model to get the correct measures, it can be done using DAX and bridge tables!

1. Create a measure called *Transaction Amount #1* in the *Transactions* table that will calculate transactions for each customer. (Hint: You can do this one of several ways. Try using CROSSFILTER first, then SUMMARIZE)
2. Add a matrix visual with the *Account* on the rows, *Customer* on the columns and your measure in the values.



Account	George	Jackie	Jane	Kylie	Michael	**Total**
George	2589					**2589**
George-Jackie	915	915				**915**
Jackie		2462				**2462**
Jane			1078			**1078**
Kylie				1264		**1264**
Michael					714	**714**
Michael-Jane			545		545	**545**
Total	**3504**	**3377**	**1623**	**1264**	**1259**	**9567**

Many-to-many relationships can be useful when properly controlled. DAX allows you to control which measures use a many-to-many relationship and which do not

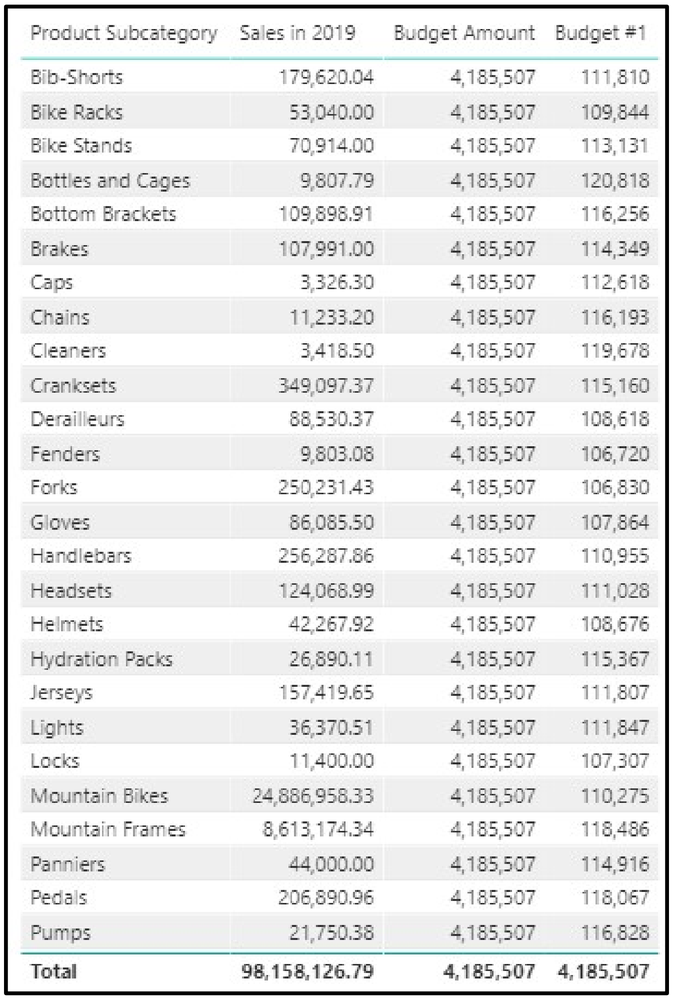
Check your Code

```
Step 1

#1 =
CALCULATE(
    [Total Transactions Amount],
    CROSSFILTER(AccountsCustomers[Account ID], Account[Account ID],
Both)
)

Step 2

#2 =
CALCULATE(
    [Total Transactions Amount],
    AccountsCustomers
    )

Step 3

#3 =
CALCULATE(
    [Total Transactions Amount],
    SUMMARIZE(AccountsCustomers, Account[Account])
)
```

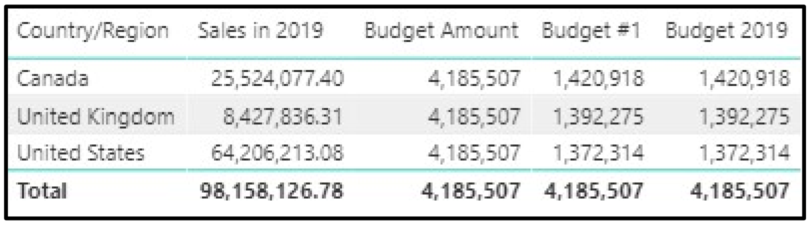
Relationships at Different Granularities

Granularity is the level of detail your data has. Calculating measures across different granularities often will give you incorrect results.

1. Create a table visual that will include *Product Subcategory, Sales Amount 2019*, and *Budget Amount*.
2. Create a measure called *Budget #1* that calculates the correct budget amount across *Product Subcategory* and *Address [Country/Region]*. Create another table that uses *Address[Country/Region]* instead of *Product Subcategory* in the rows. (Hint: Use variables and VALUES)



Product Subcategory	Sales in 2019	Budget Amount	Budget #1
Bib-Shorts	179,620.04	4,185,507	111,810
Bike Racks	53,040.00	4,185,507	109,844
Bike Stands	70,914.00	4,185,507	113,131
Bottles and Cages	9,807.79	4,185,507	120,818
Bottom Brackets	109,898.91	4,185,507	116,256
Brakes	107,991.00	4,185,507	114,349
Caps	3,326.30	4,185,507	112,618
Chains	11,233.20	4,185,507	116,193
Cleaners	3,418.50	4,185,507	119,678
Cranksets	349,097.37	4,185,507	115,160
Derailleurs	88,530.37	4,185,507	108,618
Fenders	9,803.08	4,185,507	106,720
Forks	250,231.43	4,185,507	106,830
Gloves	86,085.50	4,185,507	107,864
Handlebars	256,287.86	4,185,507	110,955
Headsets	124,068.99	4,185,507	111,028
Helmets	42,267.92	4,185,507	108,676
Hydration Packs	26,890.11	4,185,507	115,367
Jerseys	157,419.65	4,185,507	111,807
Lights	36,370.51	4,185,507	111,847
Locks	11,400.00	4,185,507	107,307
Mountain Bikes	24,886,958.33	4,185,507	110,275
Mountain Frames	8,613,174.34	4,185,507	118,486
Panniers	44,000.00	4,185,507	114,916
Pedals	206,890.96	4,185,507	118,067
Pumps	21,750.38	4,185,507	116,828
Total	**98,158,126.79**	**4,185,507**	**4,185,507**



Country/Region	Sales in 2019	Budget Amount	Budget #1	Budget 2019
Canada	25,524,077.40	4,185,507	1,420,918	1,420,918
United Kingdom	8,427,836.31	4,185,507	1,392,275	1,392,275
United States	64,206,213.08	4,185,507	1,372,314	1,372,314
Total	**98,158,126.78**	**4,185,507**	**4,185,507**	**4,185,507**

DAX allows us to create robust measure that work across different visuals and in different contexts like this *Budget #1* measure.

3. Create another measure called *Budget 2019* that replicates the previous measure but uses TREATAS and no variables.

Little steps like this keep your DAX clean and efficient. Remember, write DAX that works, then optimize it.

Often a measure can be written several different ways and return the same result. When this is the case, use the performance analyzer to find the most efficient DAX.

Check your Code

```
Step 2

Budget #1 =
VAR SubCategory =
    VALUES ( 'Product SubCategory'[Product Subcategory] )
VAR Country =
    VALUES ( 'Address'[Country/Region] )
RETURN
    CALCULATE (
        SUM ( Budget[Budget] ),
        Budget[Name] IN SubCategory,
        Budget[CountryRegion] IN Country
    )

Step 3

Budget 2019 =
CALCULATE (
    SUM ( Budget[Budget] ),
    TREATAS ( VALUES ( 'Product SubCategory'[Product Subcategory]
), Budget[Name] ),
    TREATAS ( VALUES ( 'Address'[Country/Region] ),
Budget[CountryRegion] )
)
```

Using Calculated Tables

Calculated tables are a simpler way to get the same results as the last section. They do however add to your data model which could impact performance. Test the performance of using calculated tables vs only using measures to see which gives you faster refresh times.

1. Create a calculated table that will have one column containing *Country/Region* information from the *Address* table and from the *Budget* table. (Hint: Use UNION and DISTINCT)
2. Create a calculated table that will have one column containing *Product Subcategory* from the *Product Subcategory* table and from the *Budget* table. (Hint: Use UNION and DISTINCT)

You can then create a relationship between these calculated tables to the tables in the model. For example, the first table would have a relationship to the *Address* and *Budget* table on *Country/Region*. Each of these tables works for showing the budget by one dimension. If you wanted to see budget sliced by Product Category you would need an additional calculated table.

While calculated tables work, using a measure, while more complicated, provides the same results and more while taking up less space.

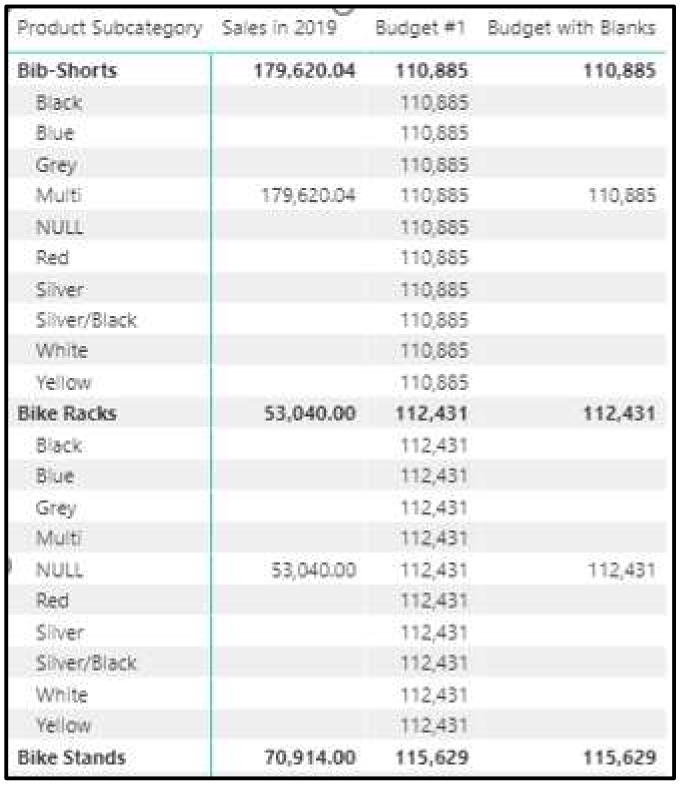
Remember that UNION and other similar functions are expensive and may impact the performance of your Power BI report.

```
Step 1

Country/Region =
DISTINCT(
    UNION(
        DISTINCT('Address'[Country/Region]),
        DISTINCT(Budget[CountryRegion])
    )
)

Step 2

SubCategory =
DISTINCT (
    UNION (
        DISTINCT ( 'Product SubCategory'[Product Subcategory] ),
        DISTINCT ( Budget[Name] )
    )
)
```

Checking Granularity

Here we have a visual that shows us the total budget for each color under the product subcategory, even if that color is not offered for that subcategory. Fix the visual so when you expand down the hierarchy *Budget #1* does not show the same number under the different colors.

1. Create a measure *# Products* that calculates the number of products in the Product table.
2. Create a measure *# All Products* that calculates *# Products* with ALL and VALUES.
3. Create a measure *Budget with Blanks* that will show *Budget #1* only in relevant color and subcategory rows. (Hint: You need to compare the number of products to the number of all products)



Product Subcategory	Sales in 2019	Budget #1	Budget with Blanks
Bib-Shorts	**179,620.04**	**110,885**	**110,885**
Black		110,885	
Blue		110,885	
Grey		110,885	
Multi	179,620.04	110,885	110,885
NULL		110,885	
Red		110,885	
Silver		110,885	
Silver/Black		110,885	
White		110,885	
Yellow		110,885	
Bike Racks	**53,040.00**	**112,431**	**112,431**
Black		112,431	
Blue		112,431	
Grey		112,431	
Multi		112,431	
NULL	53,040.00	112,431	112,431
Red		112,431	
Silver		112,431	
Silver/Black		112,431	
White		112,431	
Yellow		112,431	
Bike Stands	**70,914.00**	**115,629**	**115,629**

DAX gives you control over how your measures calculate and are displayed. That kind of control is not available with quick measures.

Check your Code

```
Step 1

# Products = COUNTROWS('Product')

Step 2

# All Products =
CALCULATE (
    [# Products],
    ALL ( 'Product' ),
    VALUES ( 'Product SubCategory'[Product Subcategory] )
)

Step 3
Budget with Blanks =
IF ( [# All Products] = [# Products], [Budget #1] )
```

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