Microsoft Power Tools for Data Analysis #18
DAX Iterators, DAX Table Functions, Granularity, Cardinality, and Materialized Tables
Notes from Video:

Table of Contents

1) DAX Iterator, Tables & Grain from Last Video 2
2) DAX Iterator, Tables & Grain from This Video 4
3) DAX Iterator Functions 6
4) Context Transition Reminder: Convert Row Context into Filter Context 7
5) Materialize Tables 7
6) Be Careful of Context Transition & Iterating over a Fact Table that "Materialize" Unnecessary Tables 7
7) Opening DAX Studio 8
8) Dax Studio for Timing Formulas 9
9) More notes on DAX Studio for timing formulas 10
10) Be Careful of Context Transition & "Double Count" Problem 11
11) Grain or Granularity 12
12) Importance of DAX Table Functions to create Correct Grain in Iterators : 13
13) DAX Table Functions 13
14) ALL Function 14
15) VALUES Function 14
16) CROSSJOIN Function 16
17) ALLEXCEPT Function 17
18) ALLNOBLANKROW Function 17
19) DISTINCT Function 17
20) FILTER Function 17
21) CALCULATETABLE Function 17
22) ADDCOLUMNS Function 17
23) Table Functions to Return Unique List, Including a Blank 17
24) Table Functions to Return Unique List, Excluding a Blank 17
25) CONCATENATEX and VALUES to list values in the Current Filter Context : 18
26) Cardinality of Tables in Iterator Functions : 20
iii) Examples of Measures & Timing for Total Revenue Calculation from Video 20
iv) Examples of Measures & Timing for Ave Transitional Rev Calculation that was NOT seen in Video 22
27) Important considerations with Iterator Functions and Table Functions in DAX Measures 24
18) 21 Million Row Data Set Test and Timing of formulas (similar times in Excel and Power BI Desktop) 25
19) FILTER Function & Cardinality in Measures (More Next Video) 28

1) DAX Iterator, Tables & Grain from Last Video :

Class so far:

DAX X Iterator Functions:

1) With Row Context
2) Formula Iterates a Table,
3) Creates Array of Answers,
4) Then Aggregates

```
Total Revenue =
SUMX (
    fTransactions,
    RELATED ( dProduct[RetailPrice] ) * fTransactions[UnitsSold]
)
```

```
Average Transactional Revenue :=
AVERAGEX (
    fTransactions,
    RELATED ( dProduct[RetailPrice] ) * fTransactions[Units]
)
```

Class so far:

DAX Table Functions:

1) DAX Formulas that Deliver a Table of Values

```
dDate = CALENDAR(
            DATE(YEAR(MIN(fTransactions[Date])),1,1),
            DATE(YEAR(MAX(fTransactions[Date])),12,31)
)
```

```
% of Grand Total:=
        DIVIDE(
                [Total Revenue],
                CALCULATE(
                        [Total Revenue],
                        ALL(fTransactions)
                )
        )
```

Class so far:

Grain

1) Grain = Granularity = Size or Level of Data or Summarization
2) We can Iterate Over Different Grains for Different Calculations

Transaction Grain Calculation:

```
Average Transactional Revenue :=
AVERAGEX (
    fTransactions,
    RELATED ( dProduct[RetailPrice] ) * fTransactions[Units]
)
```

Day Grain Calculation:

```
Average Daily Revenue :=
AVERAGEX ( dDate, [Total Revenue] )
```

2) DAX Iterator, Tables & Grain from This Video :

New in this Video:

DAX Iterator Functions:

1) Some of the DAX Iterator Functions:

SUMX delivers an aggregate sum value.
AVERAGEX delivers an aggregate average value.
MAXX delivers an aggregate max value.
CONCATENATEX delivers a text string with joined items.
FILTER delivers a table of values.
ADDCOLUMNS delivers a table of values.

2) Be Careful of Iterating over a Fact Table & "Materializing" Unnecessary Tables

```
Average Transactional Revenue =
AVERAGEX ( fTransactions, [Total Revenue Iterate Over Fact] )
```

3) Be Careful of Context Transition & "Double Count" Problem

$5,661.57 Ave Transactional Revenue 2nd F	$7,306.62 Ave Transactional Revenue 2nd M

4) Try not to Iterate over Fact Table. Can we reduce "Cardinality"?

```
Average Transactional Revenue :=
AVERAGEX (
    fTransactions,
    RELATED ( dProduct[RetailPrice] ) * fTransactions[UnitsSold]
)
```

```
Average Transactional Revenue Iterate Product =
SUMX (
    dProduct,
    dProduct[RetailPrice] * CALCULATE ( SUM ( fTransactions[UnitsSold] ) )
)
    / COUNTROWS ( fTransactions )
```

New in this Video:

DAX Table Functions:

Some of the DAX Table Functions:

- **ALL (Table or column or columns) – Removes all filters…**
- **VALUES(Column or Table) = Returns a unique list of records…**
- **CROSSJOIN(Table,Table) = Cartesian product…**
- ALLNOBLANKROW(Table or column or columns) = Removes all filters without extra blank…
- DISTINCT(Column or Table) = Returns a unique list of records without extra blank…
- ALLEXCEPT(Table,Column) = Removes all filters except…
- FILTER(Table, Filter) = Filters a table by Iterating…
- CALCULATETABLE(Table,Filters1, Filter2…) = Filters a table based on Data Model…

New in this Video:

Grain:

Use Table Functions to Determine Grain of Iterator

```
Average Monthly Revenue =
AVERAGEX ( VALUES ( dDate[Year Month] ), [Total Revenue] )
```

```
Average Monthly Revenue CJ =
AVERAGEX (
    CROSSJOIN ( VALUES ( dDate[Year] ), VALUES ( dDate[Month] ) ),
    [Total Revenue]
)
```

3) DAX Iterator Functions :

   i) Iterator Functions create Row Context and make a Row-by-Row calculation on a table (with a certain granularity) to generate an array of answers which can be aggregated, joined or delivered as a table. The number of elements in the array is equal to the number of rows in the table (cardinality) that is being iterated.
   ii) Examples of Iterator Functions that generate Row Context:
      1. SUMX delivers an aggregate sum value.
      2. AVERAGEX delivers an aggregate average value.
      3. MAXX delivers an aggregate max value.
      4. CONCATENATEX delivers a text string with joined items.
      5. FILTER delivers a table of values.
      6. ADDCOLUMNS delivers a table of values.
   iii) X Iterator functions:
      1. Examples; SUMX, AVERAGEX, MAXX, CONCATENATEX, and more…
      2. Have two arguments:
         i. The 1st argument is a Table with a certain granularity (size) and cardinality (number of elements).
         ii. The 2nd argument is an expression (formula or column) that uses Row Context to make a Row-by-Row calculations for each row in the 1st argument table. This Row-by-Row calculation results in an array of answers, which is aggregated according to the Function name. For example, SUMX sums the elements in the array and CONCATENATEX would join the elements in the array.
      3. What X Iterator Functions can do:
         i. Simulate a Calculated Column in a single Measure and then make an aggregate calculation.
            1. Example we saw in Video #15,
               =SUMX(fTransactions,ROUND(RELATED(dProduct[RetailPrice])*fTransactions[UnitsSold]*(1-fTransactions[Discount]),2))
               i. The fTransactions table has a transactional granularity.
               ii. The ROUND formula iterates the entire fTransactions table to calculate the Transactional Line Item Revenue for each row.
               iii. The resultant array of answers is then summed to deliver a single aggregated answer.
         ii. Calculate a Row-by-Row amount at a given granularity based on the table that is in the first argument of the Iterator Function.
            1. Example we saw in video #15 & #16:
               =AVERAGEX(dDate,[Total Revenue])
               i. The table dDate has day granularity.
               ii. The Total Revenue Measure iterates the entire dDate table to calculate the Daily Revenue Total for each row.
               iii. The resultant array of answers is then averaged to deliver a single aggregated answer.

iii. If the second argument contains a Measure or a formula inside the CALCULATE Function Context Transition will be performed for each row in the table.

1. Example in Video #15 & #16:
   =AVERAGEX(dDate,[Total Revenue])
   i. First, the Total Revenue Measure receives the Filter Context from the external report AND then through Context Transition, the day level filter from the dDate table flows into the Total Revenue Measure . It is Context Transition (Row Context converted to Filter Context) that allows the Day Level filter to flow from each row in the dDate table into the Measure.

iv) Other Iterators that are not X Iterator Functions:
1. Examples: FILTER and ADDCOLUMNS.
2. The DAX FILTER function iterates a table to determine which rows to include, and then delivers a filtered table as the final result.
3. The DAX ADDCOLUMNS functions iterates an existing table and makes a calculation for each row and then adds the resultant array as a new column, and then delivers a table with a new column as the final result.

4) Context Transition Reminder: Convert Row Context into Filter Context :

i) Context Transition: When a Measure with a hidden CALCULATE Function (or a formula with CALCULATE wrapped around it) converts Row Context into an Equivalent Filter Context.

5) Materialize Tables :

i) Materialize = a table that must be create from the columnar database storage engine in order for the formula to calculate an answer.

6) Be Careful of Context Transition & Iterating over a Fact Table that "Materialize" Unnecessary Tables :

i) In the Video we had the choice between these two formulas:

1.
```
Ave Transactional Rev Iterate Fact 2nd F :=
AVERAGEX (
    fTransactions,
    fTransactions[UnitsSold] * RELATED ( dProduct[RetailPrice] )
)
```

2.
```
Ave Transactional Rev Iterate Fact 2nd M :=
AVERAGEX ( fTransactions, [Total Revenue Iterate Over Fact] )
```

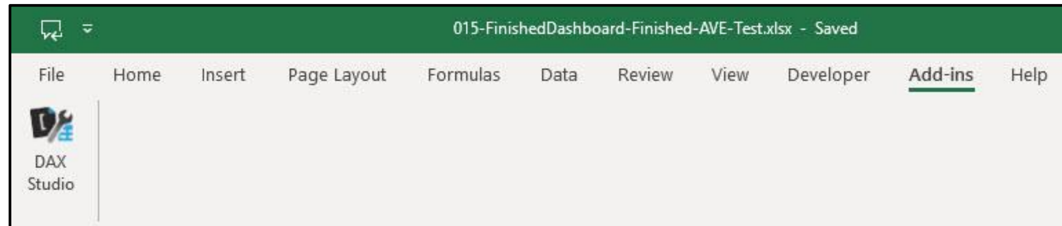
where the Measure looks like this:

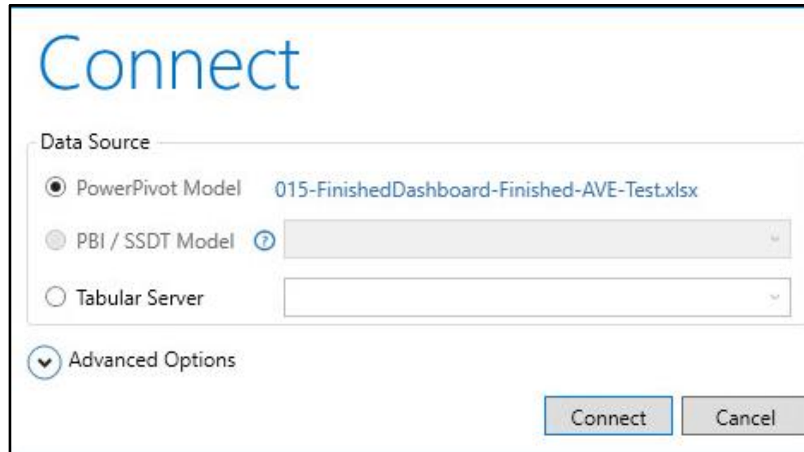
```
Total Revenue Iterate Over Fact :=
SUMX (
    fTransactions,
    fTransactions[UnitsSold] * RELATED ( dProduct[RetailPrice] )
)
```

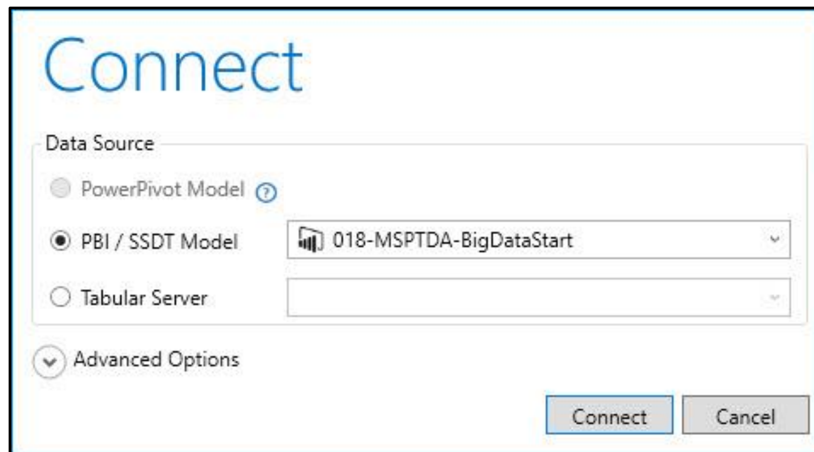
ii) The problems with Measure #2, “Ave Transactional Rev Iterate Fact 2nd M” are listed here:
1. Context Transition is not necessary in the formula to calculate Average Transactional Revenue.
2. Context Transition materializes a Fact Table for each row in the Iterator.
3. If the Fact Table has duplicate records, we get a “Double Count” Error.

iii) Measure #2, “Ave Transactional Rev Iterate Fact 2nd M” takes a LONG time to calculate and should be avoided. It is the fact that the Fact Table has to materialize for each row as it iterates over the Fact Table, fTransactions, in AVERAGEX that causes the slow down. In the next section of this pdf file we learn how to use DAX Studio to time these formulas.

7) Opening DAX Studio :

1. Search Google and download DAX Studio.
2. In Excel, you can use Add-in button as seen here:
    i. 

3. In Excel, to Connect to the Excel File, use this dialog box:
    i. 

4. In Power BI Desktop, to connect to a Power BI Desktop file, use the Start Menu or a DAX Studio Shortcut to open DAX Studio, then use the "PBI / SSDT Model" dialog button as seen here: **SSDT = SQL Server Data Tools
    i. 

8) Dax Studio for Timing Formulas :

i) Tables, Columns, Calculated Columns and Measures from Data Model
ii) Editor pane of DAX Studio (White area) contains the code of your Query. We have to use EVALUATE Command to materialize table. We use the ROW DAX Function with a column name in the first argument and the Measure in the second argument so that we can materialize a one-column and one-row table with our Measure Result.
iii) “Server Timing” button in the Traces group will allow us to time the Measure in milliseconds.
iv) Click Run and Clear Cache button, to run query and time Measure.
v) FE = Formula Engine. FE can only use one processor to calculate the formula.
vi) SE = Storage Engine. SE can use multiple processors to calculate the formula. In general, the more that the SE does the work, the faster the formula will calculate.
vii) Total Time (in milliseconds)= SE Time + FE Time.
viii) SE CPU = total time SE uses with multiple processors.
ix) Rows tells how many rows where in the table that had to be materialized internally during the Measure Calculation process.
x) xmSQL code is the language that is used by the VertiPaq Engine (Columnar Database and Formula Calculation Engine)

9) More notes on DAX Studio for timing formulas :

i) Times are in Millisecond → 1000 ms = 1 second.
ii) Any time below 16 milliseconds is "noise". "16 milliseconds is the granularity of the clock"
iii) 20, 40 milliseconds not much time…
iv) SE CPU Time
    1. Time in milliseconds that describes how much CPU has been consumed
    2. Usually time spent by CPU consuming the SE result
    3. This can be bigger than Total because it is using multiple cores (threads)
    4. **SE CPU Time = SE Time * Number of Threads (they call then "Cores")**
        i. "Parallelism"
        ii. Each Core or Thread works on one segment (1 Million in Excel and Power BI Desktop and 8 million in SSAS)
v) Rows
    1. Number Rows Materialized from SE Query
    2. Row column in DAX Studio may not yield an accurate number : it is just an estimate. You might have to look in Query Plan to see actual number of rows that are materialized.
vi) KB
    1. Size of materializing

10) Be Careful of Context Transition & "Double Count" Problem .

i) The “Double Count” problem occurs when:

1. The Fact Table has no Primary Key and there are duplicate records.
2. You are using a formula with the CALCULATE Function wrapped around it or a Measure with a hidden CALCULATE Function wrapped around it, and the formula is calculating across an iterator function or down a Calculated Column, and as a result, Context Transition converts the Row Context into an Equivalent Filter Context.

ii) Why Double Count Problems Happens:

1. When the above conditions are met and the formula calculates on a row that is a duplicate record, rather than having the Fact Table filter down to a single row as part of the filtering process in “Filter Context”, the Fact Table filters down to a table that contains all the duplicate records. This happens because the hidden CALCULATE Function wrapped around the Measure Converts the Row Context into an Equivalent Filter Context, which filters the table down to the two duplicate records. With those two rows in the Fact Table, the Measure calculates a Total Revenue Answer that is double what the correct answer is.
2. In our Video Example (as seen below), Row #1 and #2 in the Fact Table are duplicates and so when the Measure in the “Line Item Rev M” Calculated Column hits Row #1, the Measure filters the Fact Table down to both duplicate records and rather than calculating a Total Revenue amount of $1,425.45, it calculates an incorrect answer that is double that, $2,850.90.

```
1 Line Item Rev M = [Total Revenue]
```

Date	Product	Sales	SalesRepID	Units	Line Item Rev F	Line Item Rev M
December 13, 2018	Carlota	$1,425.45	6	51	1425.45	2850.9
December 13, 2018	Aspen	$11,160	3	558	11160	11160
December 13, 2018	Carlota	$1,425.45	6	51	1425.45	2850.9
December 14, 2018	Carlota	$3,493.75	5	125	3493.75	3493.75

Duplicate Records

iii) Solutions for this problem are:

1. Solutions #1: Use Formula rather than Measure in the second argument of the AVERAGEX function so Context Transition will not occur. As seen here, use Measure #1, rather than Measure #2:
2. Solution #2: Use Power Query to add Primary Key using the “Add Column, Index Column” feature. This will eliminate all duplicates.

11) Grain or Granularity

- Size or level of detail.
  - i. What is the grain or granularity of the sales table?
    1. Is the grain of the sales table at the invoice line item product level (Transaction Level)?
    2. Is the grain of the sales table at the Total Invoice Level?
  - ii. Is the granularity of the 1) helper column or 2) iteration of the DAX Formula or 3) of the report at the day, month, year level?
- More or Less Granularity?
  - i. The more granular, the smaller the size, like a grain of sand is small.
  - ii. The less granularity, the more aggregated the number is (the bigger the number is), like moving from adding total sales from daily sales to monthly sales to year sales and so on.
- Granularity is important when designing a data model and creating formulas in both Power Query and the Data Model.
- Example of Grain or Granularity:

Grain or Granularity = Size or level of detail.

Grain = Invoice Line or Product or Transaction Level
The **more granular**, the smaller the size.

Date	Invoice #	Units	SalesRepKey	ProductKey	Sales
10/20/17	27002	44	1	4	$620.95
10/20/17	27002	37	1	3	$484.31
10/20/17	27003	38	2	5	$376.69
10/20/17	27003	82	2	1	$1,141.96
10/20/17	27004	56	3	2	$725.45
10/20/17	27004	25	3	2	$222.67
10/20/17	27005	130	4	3	$1,038.10
10/21/17	27006	10	3	1	$154.41
10/24/17	27007	82	5	5	$1,236.00
10/24/17	27008	53	3	4	$684.48
10/24/17	27009	12	5	6	$127.10
10/24/17	27010	22	4	5	$269.52
10/25/17	27011	82	3	7	$739.69
10/25/17	27012	172	5	7	$1,201.01
10/25/17	27012	78	5	6	$546.06
10/25/17	27012	12	5	1	$162.45

Grain = Invoice Level
The **less granularity**, the more aggregated the number is (bigger number)

Date	Invoice #	SalesRepKey	Sales
10/20/17	27002	1	$1,105.26
10/20/17	27003	2	$1,518.65
10/20/17	27004	3	$948.12
10/20/17	27005	4	$1,038.10
10/21/17	27006	3	$154.41
10/24/17	27007	5	$1,236.00
10/24/17	27008	3	$684.48
10/24/17	27009	5	$127.10
10/24/17	27010	4	$269.52
10/25/17	27011	3	$739.69
10/25/17	27012	5	$1,909.52

For Criteria in Report ==>> Granularity is decreasing
Numbers are more aggregated

Day Grain — **Month** Grain — **Year** Grain

Date	Day	Month	MonthNumber	Year
10/1/17	1	Oct	10	2017
10/2/17	2	Oct	10	2017
10/3/17	3	Oct	10	2017
10/4/17	4	Oct	10	2017
10/5/17	5	Oct	10	2017
10/6/17	6	Oct	10	2017
10/7/17	7	Oct	10	2017
10/8/17	8	Oct	10	2017
10/9/17	9	Oct	10	2017
10/10/17	10	Oct	10	2017
10/11/17	11	Oct	10	2017
10/12/17	12	Oct	10	2017
10/13/17	13	Oct	10	2017
10/14/17	14	Oct	10	2017
10/15/17	15	Oct	10	2017
10/16/17	16	Oct	10	2017
10/17/17	17	Oct	10	2017

12) Importance of DAX Table Functions to create Correct Grain in Iterators :

i) DAX Table functions like VALUES and ALL and CROSSJOIN are important DAX Functions that can help us to create the correct Grain or Cardinality inside Iterator Functions.

ii) Examples of VALUES Functions that creates a Month Grain in the AVERAGEX Iterator Function:

```
Ave Monthly Rev :=
AVERAGEX ( VALUES ( dDate[Year Month] ), [Total Revenue Iterate Over Fact] )
```

iii) Examples of VALUES and CROSSJOIN Functions that create a Month Grain in the AVERAGEX Iterator Function:

```
Ave Monthly Rev CJ :=
AVERAGEX (
    CROSSJOIN ( VALUES ( dDate[Year] ), VALUES ( dDate[Month] ) ),
    [Total Revenue Iterate Over Fact]
)
```

13) DAX Table Functions :

i) DAX Table Functions are DAX Formulas that deliver a table of values.

ii) Table Functions can be used:

1. Internally in a Measure, but the final answer from the calculation is a scalar value (single value).
2. Internally in a Calculated Column or iterator function, but the final answer from the calculation is a scalar value (single value).
3. In a Query that returns a whole table:
   i. In Power BI Desktop to deliver a table to the Data Model.
   ii. In DAX Studio using the EVALUATE command.
   iii. In an Excel Sheet using the Existing Connections feature and the EVALUATE command.
4. To create the correct Grain or Cardinality in Iterator Functions.

iii) Some Useful DAX Table Functions **(Summary Notes)**:

1. ALL (Column or columns) – Removes all filters and returns a unique list of records as a table, with an additional single blank row if there are unmatched items in the relationship. ALL Removes filters from the Current Filter Context. ALL(Table) removes all filters and returns full table.
2. VALUES(Column or Table) = Returns a unique list of records in the Current Filter Context, with an additional single blank row if there are unmatched items in the relationship. Can "see" Filter Context". VALUES(Table) returns full table, filtered in the current filter context.
3. CROSSJOIN(Table,Table) = Cartesian product of two or more tables that returns a table.
4. ALLNOBLANKROW(Table or column or columns) = Removes all filters and returns a table that contains a unique list of records without an extra blank row for unmatched items in a relationship.
5. DISTINCT(Column or Table) = Returns a unique list of records in the Current Filter Context, without an extra blank row for unmatched items in a relationship.
6. ALLEXCEPT(Table,Column) = Removes all filters and returns a table that contains a unique list of records from the columns in table, without the excluded column.
7. FILTER(Table, Filter) = Filters a table using a column from that table and delivers a table of values. Iterates Row-by-Row. Can "see" only Table in first argument.
8. CALCULATETABLE(Table,Filters1, Filter2…) = this function can filter a table based on any columns in the Data Model that has a Relationship with the Table being filtered. This function, like CALCULATE, can change the Filter Context. Can "see" whole Data Model.
9. ADDCOLUMNS(Table,"Name New Column", Expression) = Adds new column/s to a table.

14) ALL Function :

i) Function and arguments: ALL (Table or column or columns)

ii) ALL Table Function does two things:

   1. If ALL is used as a filter in CALCULATE, it will remove all filters on the table or column.
   2. If ALL is used as a Table Function in other functions (like FILTER or SUMX), it removes all filters and delivers a unique list of records, including a blank for unmatched items in a relationship. Actual Blanks from a column are included in the unique list.

iii) Notes for different scenarios with ALL:

   1. ALL(table) = removes all filters on table and returns the full table.
   2. ALL(column From Dimension Table) removes all filters and returns a unique list, including one extra blank row if there are any unmatched items in the relationship. The extra blank row is to catch any items like in a SUM Calculation.
   3. ALL(column from Fact Table) removes all filters and returns a unique list of all items, including all unmatched items in a relationship.
   4. ALL(column, column) removes all filters and returns a unique list of records, or a list/table of all combinations of attributes from the columns, or distinct (unique) combination of values.
      i. All columns must be from same table.
   5. ALL will only except a table or column, or columns. It will not accept a different Table Function.
   6. In a PivotTable, CALCULATE([Total Revenue],ALL(Fact Table)) returns the Grand Total because all filters are removed from the Fact Table
   7. VALUES corresponds to ALL because they both return a blank if there are unmatched items in a relationship.
   8. ALL is not an iterator and does not change Filter Context, it only removes filters from table or column, or columns.

15) VALUES Function :

i) Function and arguments: VALUES(Column or Table)

ii) VALUES function that works on a single column returns a unique list of records in the Current Filter Context, with an additional single blank row if there are unmatched items in the relationship. Actual Blanks from a column are included in the unique list.

iii) VALUES function that works on a full table returns the full table, filtered in the current filter context.

   1. VALUES(Table) delivers full table in Current Filter Context.

iv) If you use VALUES in a Measure, it will see the Current Filter Context.

v) If VALUES returns a single item, it is converted to a scalar value if needed by the formula.

vi) VALUES can be used to deliver a parameter from an Excel Table to a DAX Formula (seen later in the class).

vii) Differences between ALL and VALUES:

   1. ALL
      i. Can take a table, a single column or two or more columns as an argument.
      ii. ALL Removes filters in the Current Filter Context and shows the unique records of all items for a column or columns and the full table for a table.
   2. VALUES:
      i. Can take a column or a table as its argument.
      ii. Sees Current Filter Context and delivers a unique list of the items given the Current Filter Context for a column, and the full table given the Current Filter Context.

3. Picture about VALUES and ALL Differences & Similarities from Video:

VALUES
• VALUES(Column) = Unique List
• VALUES(Table) = Full Table
• "Sees" Filter Context
• Return 1 Blank Cell when there are Unmatched Items in Relationship
• If there is no Filter Context = They Both Return Same Items
• VALUES(Column or Table)

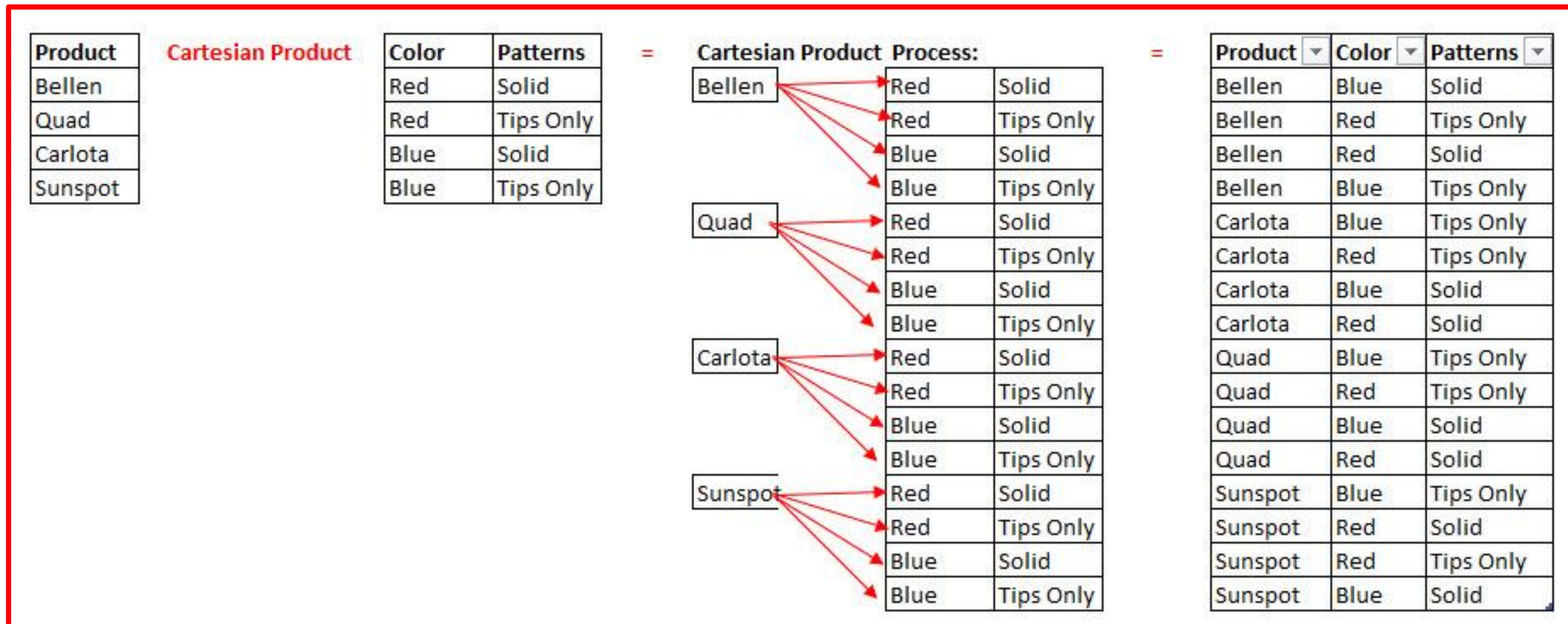
ALL
• ALL(Column or Columns) = Unique List
• ALL(Table) = Full Table
• Removes Filters
• Return 1 Blank Cell when there are Unmatched Items in Relationship
• If there is no Filter Context = They Both Return Same Items
• ALL(Column or Columns from Same Table or Table)

16) CROSSJOIN Function ;

i) Function and arguments: CROSSJOIN(Table,Table)

ii) CROSSJOIN performs a Cartesian Product of two or more tables that returns a table where the # rows = product of the # of rows from all tables and the # columns = the sum of the # of columns in all tables.

iii) A Crossjoin in set theory performs a Cartesian Product on two or more tables as seen below:



Product
Bellen
Quad
Carlota
Sunspot

Color	Patterns
Red	Solid
Red	Tips Only
Blue	Solid
Blue	Tips Only

Color	Patterns
Red	Solid
Red	Tips Only
Blue	Solid
Blue	Tips Only
Red	Solid
Red	Tips Only
Blue	Solid
Blue	Tips Only
Red	Solid
Red	Tips Only
Blue	Solid
Blue	Tips Only
Red	Solid
Red	Tips Only
Blue	Solid
Blue	Tips Only

Product	Color	Patterns
Bellen	Blue	Solid
Bellen	Red	Tips Only
Bellen	Red	Solid
Bellen	Blue	Tips Only
Carlota	Blue	Tips Only
Carlota	Red	Tips Only
Carlota	Blue	Solid
Carlota	Red	Solid
Quad	Blue	Tips Only
Quad	Red	Tips Only
Quad	Blue	Solid
Quad	Red	Solid
Sunspot	Blue	Tips Only
Sunspot	Red	Solid
Sunspot	Red	Tips Only
Sunspot	Blue	Solid

17) ALLEXCEPT Function :

  i) Function and arguments: ALLEXCEPT(Table,Column)
  ii) ALLEXCEPT(Table,Column) = Removes all filters and returns a table that contains a unique list of records from the columns in table, without the excluded column.
  iii) ALLEXCEPT excludes specified columns and will include any future columns when updated.

18) ALLNOBLANKROW Function :

  i) Function and arguments: ALLNOBLANKROW(Table or column or columns)
  ii) ALLNOBLANKROW can take a column, or columns or a table and removes all filters and returns a table that contains a unique list of records without an extra blank row for unmatched items in a relationship. Actual Blanks from a column are included in the unique list.
  iii) DISTINCT corresponds to ALLNOBLANKROW because they both ignore blanks due to unmatched items.
  iv) ALLNOBLANKROW can be used:
    1. For counting the rows in a Dimension Table and ignore any non-matched items in the Fact Table.
    2. When you need a table function that excludes unmatched items in a relationship and will be used in the first argument of an iterator function.
  v) ALL and ALLNOBLANKROW count is equal if there is a blank in the Primary Key of the Dimension table.

19) DISTINCT Function :

  i) Function and arguments: DISTINCT(Column or Table)
  ii) DISTINCT returns a unique list of records in the Current Filter Context, without an extra blank row for unmatched items in a relationship. Actual Blanks are included in the unique list.
  iii) If you use DISTINCT in a Measure, it will give a unique list given the Current Filter Context.

20) FILTER Function :

  i) Function and arguments: FILTER(Table, Filter), like: FILTER(dProduct,dProduct[Product]="Quad")
  ii) The DAX FILTER function iterates a table to determine which rows to include, and then delivers a filtered table as the final result.
  iii) Can only filter columns that exist in the table that sits in the first argument of FILTER.
  iv) This is an iterator function that uses Row Context and iterates each row in the table that sits in the first argument of FILTER.
  v) More later in class about FILTER…

21) CALCULATETABLE Function :

  i) Function and arguments: CALCULATETABLE(Table,Filters1, Filter2…), like: CALCULATETABLE(fTransactions,dProduct[Product]="Quad")
  ii) CALCULATETABLE can filter a table based on any columns in the Data Model that has a Relationship with the Table being filtered. This function, like CALCULATE, can change the Filter Context.
  iii) Is not an iterator function, but instead it applies filters through the Relationships (Expanded Table as we learn next video).
  iv) Usually will calculate more quickly than FILTER when there are duplicate values in the column being used to filter.
  v) More later in class about CALCULATETABLE…

22) ADDCOLUMNS Function :

  i) Function and arguments: ADDCOLUMNS(Table,"Name New Column", Expression)
  ii) Adds new column/s to a table.

23) Table Functions to Return Unique List, Including a Blank:

  i) ALL
  ii) VALUES

24) Table Functions to Return Unique List, Excluding a Blank:

  i) ALLNONBLANKROW
  ii) DISTINCT

25) CONCATENATEX and VALUES to list values in the Current Filter Context :

i) As seen below, CONCATENATEX uses the DAX Table functions VALUES and ALL and then joins all the elements into a single text string.

1. For DAX Table function ALL, all products are listed in every cell in the Pivot Table because the ALL Functions removes all filters in the Filter Context.
2. For DAX Table function VALUES, each cell in the PivotTable shows a list of the products sold for each SalesRep. This happens because the VALUES DAX Function only delivers the products that the SaleRep sold, or only the products that are listed in the Current Filter Context.

ii) Formulas are Listed Here:

```
List Products VALUES :=
CONCATENATEX (
    VALUES ( fTransactions[Product] ),
    fTransactions[Product],
    " - "
)
```

```
List Products ALL :=
    CONCATENATEX (
        ALL ( fTransactions[Product] ),
        fTransactions[Product],
        " - "
    )
```

iii) The PivotTable Report is shown here:

SalesRep	Total Revenue	List Products VALUES	List Products ALL
Chantel	$17,494.20	Carlota - Bellen - Sunspot	Carlota - Quad - Bellen - Sunspot - Aspen
Chin	$17,037.35	Carlota - Quad - Sunspot	Carlota - Quad - Bellen - Sunspot - Aspen
Fran	$6,542.62	Carlota - Quad	Carlota - Quad - Bellen - Sunspot - Aspen
Gigi	$13,451.90	Carlota - Aspen	Carlota - Quad - Bellen - Sunspot - Aspen
Sioux	$12,297.59	Quad - Bellen - Sunspot	Carlota - Quad - Bellen - Sunspot - Aspen
Tyrone	$20,279.20	Bellen - Aspen	Carlota - Quad - Bellen - Sunspot - Aspen
Grand Total	**$87,102.86**	**Carlota - Quad - Bellen - Sunspot - Aspen**	**Carlota - Quad - Bellen - Sunspot - Aspen**

26) Cardinality of Tables in Iterator Functions :

i) Cardinality = number of items in a set (table or array) or number of iterations.

ii) 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.

iii) Examples of Measures & Timing for Total Revenue Calculation from Video :

```
Total Revenue Fact 21M =
    SUMX(
        fTransactions,
        RELATED(dProduct[Price])*fTransactions[Units]
    )
```

DaxStudio - 2.7.4

File | Home | Help

Run | Cancel | Clear Cache | Output — Query; Cut | Copy | Paste | Undo | Redo — Edit; Format Query; To Upper | To Lower | Swap Delimiters | Comment | Uncomment | Merge XML — Format; Find | Replace — Find; All Queries | Query Plan | Server Timings — Traces; Scan | Cache | Internal | Right Layout | Bottom Layout — Server Timings; Connect | Refresh Metadata — Connection

Query1.dax*

Metadata: 018-MSPTDA-BigDataStart; Model
- DateTableTemplate_0e2cd09d-2S
- dDate
- dProduct
- fTransactions
  - Date
  - Measure
  - ProductID
  - Total Revenue Fact 21M
  - Total Revenue Product 630
  - Total Revenue VALUES Price 7
  - Units

```
1 EVALUATE
2     ROW("Total Revenue Fact 21M",
3     SUMX(
4         fTransactions,
5         RELATED(dProduct[Price])*fTransactions[Units]
6     ))
```

254 %

Total	SE CPU
34 ms	78 ms x2.4
FE	SE
1 ms 2.9%	33 ms 97.1%
SE Queries	SE Cache
1	0 0.0%

Line	Subclass	Duration	CPU	Rows	KB	Query
2	Scan	33	78	1	1	WITH $Expr0

```
SET DC_KIND="AUTO";
WITH
    $Expr0 := ( PFCAST ( 'dProduct'[Price] AS  REAL ) * CAST ( PFCAST
( 'fTransactions'[Units] AS  INT ) AS  REAL ) )
SELECT
SUM ( @$Expr0 )
FROM 'fTransactions'
    LEFT OUTER JOIN 'dProduct' ON 'fTransactions'[ProductID]
='dProduct'[ProductID];

'Estimated size ( volume, marshalling bytes ) : 1, 16'
```

```
Total Revenue Product 630 =
    SUMX(
        dProduct,
        dProduct[Price]*CALCULATE(SUM(fTransactions[Units]))
    )
```

```
EVALUATE
    ROW("Total Revenue Product 630",
    SUMX(
        dProduct,
        dProduct[Price]*CALCULATE(SUM(fTransactions[Units]))
    ))
```

Total	SE CPU	FE	SE	SE Queries	SE Cache
16 ms	63 ms (x4.2)	1 ms (6.3%)	15 ms (93.8%)	2	0 (0.0%)

Line	Subclass	Duration	CPU	Rows	KB	Query
2	Scan	14	63	633	10	SELECT 'dPro
4	Scan	1	0	1	1	WITH $Expr

```
SET DC_KIND="AUTO";
SELECT
'dProduct'[ProductID],
SUM ( 'fTransactions'[Units] )
FROM 'fTransactions'
    LEFT OUTER JOIN 'dProduct' ON 'fTransactions'[ProductID]
='dProduct'[ProductID];

'Estimated size ( volume, marshalling bytes ) : 633, 10128'
```

```
Total Revenue VALUES Price 7 =
    SUMX(
        VALUES(dProduct[Price]),
        dProduct[Price]*CALCULATE(SUM(fTransactions[Units]))
    )
```

```
EVALUATE
    ROW("Total Revenue VALUES Price 7",
    SUMX(
        VALUES(dProduct[Price]),
        dProduct[Price]*CALCULATE(SUM(fTransactions[Units]))
    ))
```

Total	SE CPU	FE	SE	SE Queries	SE Cache
12 ms	63 ms (x6.3)	2 ms (16.7%)	10 ms (83.3%)	1	0 (0.0%)

Line	Subclass	Duration	CPU	Rows	KB	Query
2	Scan	10	63	10	1	SELECT 'dPro

```
SET DC_KIND="AUTO";
SELECT
'dProduct'[Price],
SUM ( 'fTransactions'[Units] )
FROM 'fTransactions'
    LEFT OUTER JOIN 'dProduct' ON 'fTransactions'[ProductID]
='dProduct'[ProductID];

'Estimated size ( volume, marshalling bytes ) : 10, 160'
```

iv) Examples of Measures & Timing for Ave Transitional Rev Calculation that was NOT seen in Video :

```
Ave Transactional Rev Iterate Fact M :=
AVERAGEX ( fTransactions, [Total Revenue Iterate Over Fact] )
```

DaxStudio - 2.7.4

Query1.dax*

```
1 EVALUATE
2 ROW("Ave Transactional Rev Iterate Fact M",
3 AVERAGEX(fTransactions,[Total Revenue Iterate Over Fact])
4 )
```

Total	SE CPU
1,852 ms	1,859 ms (x1.6)
FE	**SE**
719 ms (38.8%)	1,133 ms (61.2%)
SE Queries	**SE Cache**
2	0 (0.0%)

Line	Subclass	Duration	CPU	Rows	KB	Query
2	Scan	882	1,328	2,204,103	34,440	WITH $Expr0 := (PFCAST ('dI
4	Scan	251	531	1	1	WITH $Expr0 := [CallbackD

```
SET DC_KIND="AUTO";
WITH
    $Expr0 := ( PFCAST ( 'dProduct'[RetailPrice] AS  REAL ) * CAST ( PFCAST ( 'fTransactions'[UnitsSold] AS  INT ) AS  REAL ) )
SELECT
'fTransactions'[Date], 'fTransactions'[ProductID], 'fTransactions'[SalesRepID], 'fTransactions'[UnitsSold], 'fTransactions'[Index],
SUM ( @$Expr0 )
FROM 'fTransactions'
    LEFT OUTER JOIN 'dProduct' ON 'fTransactions'[ProductID]='dProduct'[ProductID];


'Estimated size ( volume, marshalling bytes ) : 2204103, 35265648'
```

```
Ave Transactional Rev Iterate Product :=
SUMX (
    dProduct,
    dProduct[RetailPrice] * CALCULATE ( SUM ( fTransactions[UnitsSold] ) )
)
    / COUNTROWS ( fTransactions )
```

DaxStudio - 2.7.4

Query1.dax*

```
1 EVALUATE
2 ROW("Ave Transactional Rev Iterate Product",
3 SUMX(
4     dProduct,
5     dProduct[RetailPrice]*CALCULATE(SUM(fTransactions[UnitsSold]))
6 )
7 /COUNTROWS(fTransactions)
8 )
```

Total	SE CPU
5 ms	0 ms (x0.0)
FE	**SE**
2 ms (40.0%)	3 ms (60.0%)
SE Queries	**SE Cache**
2	0 (0.0%)

Line	Subclass	Duration	CPU	Rows	KB	Query
2	Scan	2	0	19	1	SELECT 'dProduct'[ProductID]
4	Scan	1	0	1	1	WITH $Expr0 := (PFCAST (

```
SET DC_KIND="AUTO";
SELECT
'dProduct'[ProductID],
SUM ( 'fTransactions'[UnitsSold] )
FROM 'fTransactions'
    LEFT OUTER JOIN 'dProduct' ON 'fTransactions'[ProductID]='dProduct'[ProductID];


'Estimated size ( volume, marshalling bytes ) : 19, 304'
```

```
Ave Transactional Rev Iterate VALUES Price :=
SUMX (
    VALUES ( dProduct[RetailPrice] ),
    dProduct[RetailPrice] * CALCULATE ( SUM ( fTransactions[UnitsSold] ) )
)
    / COUNTROWS ( fTransactions )
```

DaxStudio - 2.7.4

File | Home | Help

Run | Cancel | Clear Cache | Output — Query | Cut | Copy | Paste | Undo | Redo — Edit | Format Query | To Upper | To Lower | Swap Delimiters | Comment | Uncomment | Merge XML — Format | Find | Replace — Find | All Queries | Query Plan | Server Timings — Traces | Scan | Cache | Internal | Right Layout | Bottom Layout — Server Timings | Connect | Refresh Metadata — Connection

Query1.dax*

Metadata: Microsoft_SQLServer_Analy | Model | dDate | dProduct | dSalesReps | fTransactions

```
1 EVALUATE
2 ROW("Ave Transactional Rev Iterate VALUES Price",
3 SUMX(
4     VALUES(dProduct[RetailPrice]),
5     dProduct[RetailPrice]*CALCULATE(SUM(fTransactions[UnitsSold]))
6 )
7 /COUNTROWS(fTransactions)
8 )
```

218 %

Total	SE CPU
4 ms	0 ms (x0.0)

FE	SE
2 ms (50.0%)	2 ms (50.0%)

SE Queries	SE Cache
1	0 (0.0%)

Line	Subclass	Duration	CPU	Rows	KB	Query
2	Scan	2	0	7	1	SELECT 'dProduct'[RetailPrice

```
SET DC_KIND="AUTO";
SELECT
'dProduct'[RetailPrice],
SUM ( 'fTransactions'[UnitsSold] )
FROM 'fTransactions'
    LEFT OUTER JOIN 'dProduct' ON 'fTransactions'[ProductID]='dProduct'[ProductID];

'Estimated size ( volume, marshalling bytes ) : 7, 84'
```

27) Important considerations with Iterator Functions and Table Functions in DAX Measures :

i) Try not to iterate over a Fact Table .
   1. Iterating over Fact Tables can take a long time since there are so many rows.
      i. For example, rather than iterating over this Fact Table with 2 million rows:
         Total Revenue Iterate Over Fact :
         =SUMX(fTransactions,RELATED(dProduct[RetailPrice])*fTransactions[UnitsSold])
      ii. Try iterating over this product table with 16 rows:
         Total Revenue Iterate Over Product :=
         SUMX(dProduct,dProduct[RetailPrice]*CALCULATE(SUM(fTransactions[UnitsSold])))
   2. Iterating over a Fact Table when there is Context Transition may lead to Double Counting if there are:
      i. Duplicate Records
         or
      ii. No Primary Key.

ii) Consider the Cardinality of the table that you are iterating over .
   1. Cardinality = number of items in a set (table or array) or number of iterations.
   2. 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.
   3. For example, if there are duplicate prices for a product, rather than iterating over the full product table:
      Total Revenue Iterate Over Product :=
      SUMX(dProduct,dProduct[RetailPrice]*CALCULATE(SUM(fTransactions[UnitsSold])))
      try iterating over a unique list of product prices:
      Total Revenue Iterate Over VALUES Price :=
      SUMX(VALUES(dProduct[RetailPrice]),dProduct[RetailPrice]*CALCULATE(SUM(fTransactions[UnitsSold])))
   4. For some other tables it may be more efficient to use ALL or VALUES to generate a unique set of records, rather than using the entire table or column.

iii) Be sure to get the correct grain for your table. If you are trying to calculate Average Daily Revenue, use to use the Date Table, whereas, if you are calculating the Average Monthly Revenue, your table will need a month grain.

18) 21 Million Row Data Set Test and Timing of formulas (similar times in Excel and Power BI Desktop) :

```
EVALUATE
    ROW(
        "Total Revenue Fact 21 M",
        SUMX(
            fTransactions,
            RELATED(dProduct[Price])*fTransactions[Units]
        )
    )
```

Total	SE CPU
28 ms	47 ms (x1.7)
FE 1 ms (3.6%)	SE 27 ms (96.4%)
SE Queries 1	SE Cache 0 (0.0%)

Line	Subclass	Duration	CPU	Rows	KB	Query
2	Scan	27	47	1	1	WITH $Expr0 :=

```
SET DC_KIND="AUTO";
WITH
    $Expr0 := ( PFCAST ( 'dProduct'[Price] AS  REAL ) * CAST ( PFCAST ( 'fTransactions'[Units] AS  INT ) AS  REAL ) )
SELECT
SUM ( @$Expr0 )
FROM 'fTransactions'
    LEFT OUTER JOIN 'dProduct' ON 'fTransactions'[ProductID]='dProduct'[ProductID];

'Estimated size ( volume, marshalling bytes ) : 1, 16'
```

DaxStudio - 2.7.4

Query1.dax*

```
EVALUATE
	ROW(
		"Total Revenue Product 630",
		SUMX(
			dProduct,
			dProduct[Price]*CALCULATE(SUM(fTransactions[Units]))
		)
	)
```

Total	SE CPU
14 ms	31 ms x2.6
FE 2 ms 14.3%	SE 12 ms 85.7%
SE Queries 2	SE Cache 0 0.0%

Line	Subclass	Duration	CPU	Rows	KB	Query
2	Scan	12	31	633	10	SELECT 'dProdu
4	Scan	0	0	1	1	WITH $Expr0 :=

```
SET DC_KIND="AUTO";
SELECT
'dProduct'[ProductID],
SUM ( 'fTransactions'[Units] )
FROM 'fTransactions'
	LEFT OUTER JOIN 'dProduct' ON 'fTransactions'[ProductID]='dProduct'[ProductID];

'Estimated size ( volume, marshalling bytes ) : 633, 10128'
```

DaxStudio - 2.7.4

File | Home | Help

Query1.dax*

Metadata: Microsoft_SQLServer_Analy… / Model / dProduct / fTransactions

```
1 EVALUATE
2     ROW(
3         "Total Revenue VALUES Price 7",
4         SUMX(
5             VALUES(dProduct[Price]),
6             dProduct[Price]*CALCULATE(SUM(fTransactions[Units]))
7         )
8     )
```

363 %

Total	SE CPU
11 ms	47 ms x5.2

FE	SE
2 ms 18.2%	9 ms 81.8%

SE Queries	SE Cache
1	0 0.0%

Line	Subclass	Duration	CPU	Rows	KB	Query
2	Scan	9	47	10	1	SELECT 'dProdu

```
SET DC_KIND="AUTO";
SELECT
'dProduct'[Price],
SUM ( 'fTransactions'[Units] )
FROM 'fTransactions'
    LEFT OUTER JOIN 'dProduct' ON 'fTransactions'[ProductID]='dProduct'[ProductID];

'Estimated size ( volume, marshalling bytes ) : 10, 160'
```

19) FILTER Function & Cardinality in Measures (More Next Video) :

ALL Uses Whole Fact Table (Bigger Cardinality):

```
Sales Units >250 ALL Table :=
    CALCULATE(
        [Total Revenue Iterate Over Fact],
        FILTER(ALL(fTransactions),fTransactions[UnitsSold]>250)
    )
```

ALL Uses Unique List of UnitsSold Column (Smaller Cardinality):

```
Sales Units >250 ALL Column :=
    CALCULATE(
        [Total Revenue Iterate Over Fact],
        FILTER(ALL(fTransactions[UnitsSold]),fTransactions[UnitsSold]>250)
    )
```