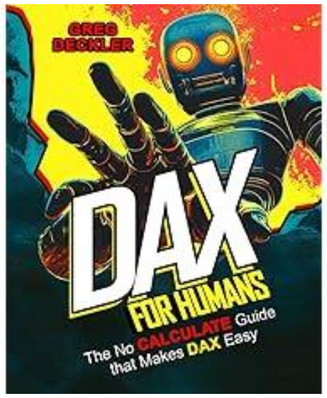
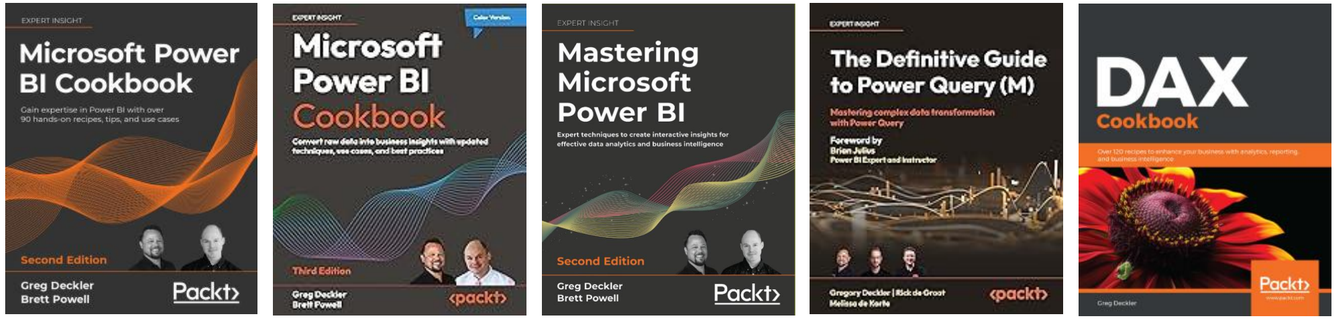
DAX Counterculture

with Greg Deckler

ABOUT ME



Vice President at a global consulting firm and a professional technology consultant for over 30 years. Seven-time Microsoft MVP for Data Platform and an active member of the Power BI Community site with over 7,500 solutions authored and over 200 Quick Measure Gallery submissions. Founded the Columbus Azure ML and Power BI User Group (CAMLPUG) and have presented at different conferences and events including Dog Food, SharePoint Saturday, CloudDevelop and M3. Author of Enterprise DNA Power Tools (Quick Measures Pro, Power Sort Pro, Conductor Pro, Metadata Mechanic Pro, DAX Editor Pro).



DAX For Humans: https://www.youtube.com/@daxforhumans

Microsoft Hates Greg: https://www.youtube.com/@microsofthatesgreg

Enterprise DNA Tools:
https://github.com/gdeckler/MicrosoftHatesGregsQuickMeasures/blob/main/EnterpriseDNAInstaller%20-%201.3.6.msi

OVERVIEW

Every technology, over time, develops a certain amount of “conventional wisdom” or “best practices”. However, this culture or group think does not always apply in all cases or is even necessarily correct.

This session will cover the following topics:

- CALCULATE
- Time Intelligence functions
- Looping
- DAX Index
- EARLIER
- Columns vs. Measures
- VALUES
- Optimizing your DAX
- Optimizing your Semantic Model
- User Defined Functions

CALCULATE

Conventional Wisdom

- CALCULATE is a fantastic, powerful, highly performant function

Unconventional Wisdom

- Don’t use CALCULATE, especially when starting out
- CALCULATE is a *complex* function with many quirks and nuances
- CALCULATE is tied to the data model
- CALCULATE is hard to debug
- Instead, use the pattern of table variables and X aggregation functions

CALCULATE

A DAX PATTERN TO SOLVE MOST PROBLEMS

```
Sum Total Cost No Pickle =
    VAR __ExcludeItem = "Pickle"
    VAR __Table =
        FILTER(
            'Table',
            'Table'[Item] <> __ExcludeItem
        )
    VAR __Result = SUMX( __Table, [Total Cost] )
RETURN
    __Result //TOCSV( __Table )
```

The DAX pattern can be described thusly:

- Create one or more **VAR**'s depending upon the requirements of the calculation. In this case, we wish to exclude **Pickle** rows, so we create a variable that consists of the text "Pickle".
- Create a table **VAR** that filters and potentially groups the rows as required by the calculation. In this case, we wish to exclude rows where the item is "Pickle", so we construct a simple DAX expression using the **FILTER** function.
- Use an X aggregator to perform a calculation over the rows of the table variable. In this case we wish to sum the **Total Cost** column so we use the **SUMX** function with the table variable **__Table** as the first parameter and the **Total Cost** column as the second parameter.

CALCULATE

dax.guide

CALCULATE evaluation follows these steps:

1. CALCULATE evaluates all the explicit filter arguments in the original evaluation context, each one independently from the others. This includes both the original row contexts (if any) and the original filter context. Once this evaluation is finished, CALCULATE starts building the new filter context.
2. CALCULATE makes a copy of the original filter context to prepare the new filter context. It discards the original row contexts, because the new evaluation context will not contain any row context.
3. CALCULATE performs the context transition. It uses the current value of columns in the original row contexts to provide a filter with a unique value for all the columns currently being iterated in the original row contexts. This filter may or may not contain one individual row. There is no guarantee that the new filter context contains a single row at this point. If there are no row contexts active, this step is skipped. Once all implicit filters created by the context transition are applied to the new filter context, CALCULATE moves on to the next step.
4. CALCULATE evaluates the CALCULATE modifiers used in filter arguments: USERELATIONSHIP, CROSSFILTER, ALL, ALLEXCEPT, ALLSELECTED, and ALLNOBLANKROW. This step happens after step 3. This is very important, because it means that one can remove the effects of the context transition by using ALL as a filter argument. The CALCULATE modifiers are applied after the context transition, so they can alter the effects of the context transition.
5. CALCULATE applies the explicit filter arguments evaluated at 1. to the new filter context generated after step 4. These filter arguments are applied to the new filter context once the context transition has happened so they can overwrite it, after filter removal — their filter is not removed by any ALL* modifier — and after the relationship architecture has been updated. However, the evaluation of filter arguments happens in the original filter context, and it is not affected by any other modifier or filter within the same CALCULATE function. If a filter argument is modified by KEEPFILTERS, the filter is added to the filter context without overwriting existing filters over the same column(s).

docs.microsoft.com

Remarks

- When filter expressions are provided, the CALCULATE function modifies the filter context to evaluate the expression. For each filter expression, there are two possible standard outcomes when the filter expression is not wrapped in the KEEPFILTERS function:

- If the columns (or tables) aren't in the filter context, then new filters will be added to the filter context to evaluate the expression.
- If the columns (or tables) are already in the filter context, the existing filters will be overwritten by the new filters to evaluate the CALCULATE expression.
- The CALCULATE function used without filters achieves a specific requirement. It transitions row context to filter context. It's required when an expression (not a model measure) that summarizes model data needs to be evaluated in row context. This scenario can happen in a calculated column formula or when an expression in an iterator function is evaluated. Note that when a model measure is used in row context, context transition is automatic.

- This function is not supported for use in DirectQuery mode when used in calculated columns or row-level security (RLS) rules.

CALCULATE

CALCULATE WEIGHTED AVERAGE

Value weighted by MonthSort per Month =

```
VAR __CATEGORY_VALUES = VALUES('Table'[Month])
RETURN
        DIVIDE(
                SUMX(
                        KEEPFILTERS(__CATEGORY_VALUES),
                        CALCULATE(SUM('Table'[Value]) * SUM('Table'[MonthSort]))
                ),
                SUMX(
                        KEEPFILTERS(__CATEGORY_VALUES),
                        CALCULATE(SUM('Table'[MonthSort]))
                )
        )
```

NO CALCULATE WEIGHTED AVERAGE

Better Weighted Average per Category =

```
  VAR __Table =
        SUMMARIZE(
                'Table’,
                [Month],
                "Value", SUM('Table'[Value]) * SUM('Table'[MonthSort]))
  VAR __Table1 =
        SUMMARIZE(
                'Table’,
                [Month],
                "Value", SUM('Table'[MonthSort]))
  VAR __Result  = DIVIDE(SUMX(__Table, [Value]), SUMX(__Table1, [Value]))
RETURN
  __Result
```

TIME INTELLIGENCE (TI) FUNCTIONS

Conventional Wisdom

- DAX's TI functions make time intelligence calculations easy

Unconventional Wisdom

- Never, ever use DAX's TI functions
- DAX's TI functions are "black boxes" that make way too many assumptions
  - DAX's TI functions assume a standard, Gregorian, calendar
  - Assume you have a Date dimension
- Some are stupidly complex (PARRELPERIOD, DATESINPERIOD, DATEADD)
- Most are glorified FILTER, MAX or MIN statements
- Useless for week calculations
- Aren't even named correctly
- Use offsets instead

...	-5	-4	-3	-2	-1	0	1	2	3	4	5	...

- The new calendars are awful to configure.

DAX Looping

Conventional Wisdom

- There is no looping in DAX

Unconventional Wisdom

- Sure, there is

```
For Loop =
// Provide some starting values
VAR __n = 5
VAR __sum = 0
// Generate a "loop table", this will emulate a for loop for i=1 to some number
VAR __loopTable = GENERATESERIES(1,__n)
// Add in our calculated sum, emulating calculations done as iterations over the loop
VAR __loopTable1 = ADDCOLUMNS(__loopTable,"__sum",__sum +
    SUMX(FILTER(__loopTable,[Value]<=EARLIER([Value])),[Value]))
// Determine our MAX interation, the maximum value for "i"
VAR __max = MAXX(__loopTable1,[Value])
RETURN
// Return the value associated with the maximum value of "i" which is the last iteration
    in our "loop"
MAXX(FILTER(__loopTable1,[Value]=__max),[__sum])

While Loop =
// Provide some starting value via user input
VAR __i = 10
// Generate a "loop table", this will emulate a while loop
VAR __loopTable = GENERATESERIES(1,__i)
// Add in our calculated value, emulating calculations done as iterations over the loop
VAR __loopTable1 = ADDCOLUMNS(__loopTable,"__i",[Value] - 1)
RETURN
COUNTROWS(FILTER(__loopTable1,[__i]>1))+1
```

DAX Index

Conventional Wisdom

- There is no way to create an index in DAX and DAX cannot preserve sort order

Unconventional Wisdom

- Sure, there is and yes it can

```
DAX Indexed Table =
    VAR __SourceTable = 'Table'
    VAR __Count = COUNTROWS(__SourceTable)
    VAR __SortText = CONCATENATEX('Table',[Column1],"|")
    VAR __Table =
        ADDCOLUMNS(
            GENERATESERIES(1,__Count,1),
            "Column1",PATHITEM(__SortText,[Value],TEXT)
        )
RETURN
    __Table
```

EARLIER

https://dax.guide/earlier/

- DISTINCTCOUNT
- DISTINCTCOUNTNOBLA...
- DIVIDE
- DOLLARDE
- DOLLARFR
- DURATION
- EARLIER
- EARLIEST

EARLIER DAX Function NOT RECOMMENDED

Syntax | Return values | Remarks | Examples | Articles | Related

Returns the value in the column prior to the specified number of table scans (default is 1).

EARLIER

Conventional Wisdom

- EARLIER is a pariah and blight on the DAX language, a by-gone relic of old that should never, ever be used in any circumstance what-so-ever

Unconventional Wisdom

- That's just mean

```
VAR __table1 =
        ADDCOLUMNS(__table,"__next",
          MINX(
            FILTER(__table,
                [MachineName]=EARLIER([MachineName])  &&
                [RepairType]<>"PM"
            ),
            [RepairStarted]
          )
        )
```

```
VAR __table1 =
        ADDCOLUMNS(__table,"__next",
          MINX(
                VAR __MachineName = MAX( [MachineName] )
                RETURN
                  FILTER(__table,
                      [MachineName]= __MachineName )  &&
                      [RepairType]<>"PM"
                  ),
            [RepairStarted]
          )
        )
```

Columns vs. Measures

Conventional Wisdom

- Never use calculated columns and instead use measures
  - Calculated columns impact data refresh times
  - Calculated columns add to the overall size of the data model

Unconventional Wisdom

- In self-service BI scenarios calculated columns are superior to measures because Microsoft has never bothered to fix their broken table and matrix visualizations and thus the measure totals problem
- The impacts are small
- Even grouping and binning use calculated columns

VALUES

Conventional Wisdom

- VALUES is a useful function

Unconventional Wisdom

- Anything VALUES can do, DISTINCT does better

Optimizing Your DAX

Conventional Wisdom

- Optimizing your DAX is really important and complicated with 800+ page books written on the subject.

Unconventional Wisdom

- DAX queries (Formula Engine and Storage Engine) are only about 5% of overall processing time for the average report and you can learn 95% of what you need to know in 1 chapter or about 25 pages.
  - Refine and consolidate logic structures
  - Use SWITCH(TRUE()) instead of nested IF statements
  - Don’t use unnecessary functions (VALUE, VALUES)
  - Filter early and filter often
  - Filter, filter, filter
  - Nested filters perform better than SWITCH(TRUE()) statements
  - All of your code in a single measure performs better than having your DAX spread across multiple measures
- Optimize your report instead. Optimizing the visual layer can improve load times by up to 10x.

THE FASTEST VISUAL IN POWER BI

ChartType	Total (ms)	Multiplier
KeyInfluencers	0.00	0.00
QA	0.00	0.00
Card	20.00	1.00
Decomposition	26.67	1.33
KPI	26.67	1.33
Slicer	26.67	1.33
Treemap	26.67	1.33
Funnel	30.00	1.50
Gauge	30.00	1.50
Multicard	30.00	1.50
Scatterplot	33.33	1.67
Table	33.33	1.67
BarChart	36.67	1.83
100ColumnChart	40.00	2.00
ColumnChart	40.00	2.00
Matrix	40.00	2.00
StackedArea	40.00	2.00
StackedColumn	40.00	2.00
StackedLineAndColumn	43.33	2.17
100BarChart	43.33	2.17
ArcGIS	43.33	2.17
AreaChart	43.33	2.17
LineAndColumn	43.33	2.17
Pie	43.33	2.17
RibbonChart	43.33	2.17
WaterfallChart	43.33	2.17
100StackedArea	46.67	2.33
Donut	46.67	2.33
LineChart	46.67	2.33
StackedBar	46.67	2.33
Violin	53.33	2.67
BubbleMap	60.00	3.00
MAQSoftwareBoxAndWhisker	60.00	3.00
NewCard	63.33	3.17
FilledMap	123.33	6.17
Narrative	160.00	8.00

GREG DECKLER

Optimizing Your Report

- Everything static should be a single image as the background of your report
- Sparklines and Table size slow you down
- Deneb is twice as fast as R visuals and 7 times faster than Python visuals
- Multi-row card visual is a third faster than the New Card visual. The break even between the original Card visual and New Card visual is 15 measures.
- Azure 3D Column and Heat Maps are twice as fast as Azure Bubble Map and four times faster than Azure Filled Map.
- ArcGIS bubble map is faster than Azure bubble map
- All third-party maps tested were faster than Azure Bubble and Filled Maps (PBI Consultants Maps, Squillion, TMap, Icon Map)
- Original bubble and filled map visuals are 2 to 4 times as fast as Azure Bubble and Filled Map visuals

Semantic Model

Conventional Wisdom

- Star Schemas are blazingly fast and everything else is garbage.

Unconventional Wisdom

- Flat, single tables for the majority of situations up to at least 100 million are actually faster in general.
- Flat vs. Star 100M Rows https://youtu.be/ZBEcWkp8Kh0

Why? How Tables are Stored

- **Partitions → Segments**
  - Each table is split into one or more *partitions* (usually one by default).
  - Each partition is split into **segments** of ~1M rows. Segment stats (min/max, distinct count, sort info) let VertiPaq skip whole chunks (“segment elimination”) during queries.
- **Columnar storage**
  - Columns are stored separately. A scan of a few columns never touches the others.
- **Encodings & compression**
  - **Dictionary encoding (most common):**
    - Per column, VertiPaq builds a **dictionary** of unique values.
    - The column’s data becomes a stream of **value IDs** (integers) that reference that dictionary.
    - IDs are **bit-packed** (fewest bits possible).
    - Runs of repeated IDs can be **run-length encoded (RLE)**. A A A B B C => A,3,B,2,C,1
  - **Value encoding (for numeric columns with many unique values):**
    - Stores scaled integers/floats directly, then bit-packs them (may skip a separate dictionary).
  - **Additional tricks:** sorting by a column can increase RLE effectiveness; low-cardinality columns compress extremely well.
- **No global dictionaries**
  - Each column has its **own** dictionary. The integer ID for “USA” in Table A’s Country column has nothing to do with the ID for “USA” in Table B’s Country column.

Why? How Relationships are Stored

Metadata + indexes, not pre-joined data

- A relationship record (one-to-many, direction, active/inactive, single/both direction, etc.) is stored as model metadata.
- VertiPaq also creates internal indexes on the columns that participate in relationships (and on some other frequently filtered/grouped columns). Think of these as value→row locators optimized for fast lookups during filter propagation.

The hidden “blank row”

- In many-to-one relationships, the lookup table gets a hidden blank row to absorb fact-side keys that don’t match (the “unknown member”). This preserves DAX semantics when referential integrity isn’t perfect.

Why? Filter Propagation

Suppose you filter a lookup table (Customers[Country] = "USA") and there's a relationship to a fact table (Sales[CustomerKey] → Customers[CustomerKey]):

1. **Filter on lookup**
   VertiPaq resolves "USA" via Customers[Country]'s dictionary and applies a bitmap over matching Customers rows.
2. **Translate to keys**
   From the filtered Customers rows, it gets the relevant CustomerKey values using Customers[CustomerKey]'s dictionary/encodings.
3. **Find fact rows**
   Using the fact column's internal index on Sales[CustomerKey], it maps those key values to the exact row IDs in Sales—often with segment elimination along the way.
4. **Apply bitmap on the fact**
   The result is a bitmap marking the Sales rows that survive the relationship filter.
   No join table is materialized; it's all dictionary lookups, indexes, and bitmaps on compressed columns.

Because each column has its own dictionary, the engine does not reuse IDs across tables. Instead, it turns filtered values on the lookup side into actual values, then finds the corresponding value IDs within the fact column's *own* dictionary and applies those via its column index. That's the essence of relationship traversal in VertiPaq.

Why Flat Tables are Often Faster

- Flat tables avoid relationship traversals.
- Segment pruning can be more efficient.
- Compression can overperform if the data is sorted or highly repetitive.
- Dictionary lookups are simpler (single dictionary space).
- DAX formulas avoid additional context propagation overhead.

In short: **VertiPaq's query pipeline is incredibly fast at scanning compressed columns, but filter propagation across relationships introduces real work.**

This is why a fully flattened, pre-joined table can outpace a perfectly modeled star schema.

User Defined Functions

Conventional Wisdom

- Game changing for everyone

Unconventional Wisdom

- Interesting but not game changing for most
- Lack enterprise features
- Kind of clunky to work with
- Less reuseable than measures (currently)
- No recursion ☹
- They can return tables, and you can pass column references ☺
- UDF Deep Dive: https://youtu.be/Asg04n3UY80


DAX
FOR HUMANS