Contents

[Data Types 3](#_Toc81133656)

[Vertipaq Compression and encoding 3](#_Toc81133657)

[Value encoding 3](#_Toc81133658)

[Hash Encoding (Dictionary Encoding) 3](#_Toc81133659)

[Run Length Encoding RLE 3](#_Toc81133660)

[RLE + Hash encoding 4](#_Toc81133661)

[Vertipaq Relationships 4](#_Toc81133662)

[Summary 4](#_Toc81133663)

[Evaluation order 4](#_Toc81133664)

[DAX Variables 4](#_Toc81133665)

[Variables can 4](#_Toc81133666)

[Variables cannot 5](#_Toc81133667)

[Variable Evaluation Order 5](#_Toc81133668)

[Coalesce function 5](#_Toc81133669)

[Modifiers 5](#_Toc81133670)

[CALCULATE 6](#_Toc81133671)

[REMOVE FILTERS 6](#_Toc81133672)

[KEEP FILTERS 6](#_Toc81133673)

[Cumulative Total 7](#_Toc81133674)

[Overall Total 7](#_Toc81133675)

[Percent of Total 7](#_Toc81133676)

[Common Table and Filter Functions 8](#_Toc81133677)

[• Filter Data 8](#_Toc81133678)

[• Add Data 8](#_Toc81133679)

[• Create Data 8](#_Toc81133680)

[• () Table Constructor 8](#_Toc81133681)

[Calculated Tables 8](#_Toc81133682)

[FILTER 8](#_Toc81133683)

[ALL 9](#_Toc81133684)

[Review FILTER (Example) 9](#_Toc81133685)

[DISTINCT 9](#_Toc81133686)

[VALUES 9](#_Toc81133687)

[VALUES VS DISTINCT: THE BLANK ROW 9](#_Toc81133688)

[SELECTEDVALUE() 10](#_Toc81133689)

[ALLEXCEPT() 10](#_Toc81133690)

# Data Types

|  |  |  |  |
| --- | --- | --- | --- |
| **DAX Data Type** | **Power BI Data Type** | **Storage Type** | **Example** |
| Integer | Whole Number | 64-bit value | Max: 9,223,372,036,854,775,807 |
| Decimal | Decimal Number | Double-precision floating-point value | 64-bit precision |
| Currency | Fixed Decimal Number | Fixed Decimal Number (Stored as Integer) | 317.9988 (4 digits after period) |
| DateTime | DateTime, Date, Time | Floating-point number | 1/1/2020 12:00 pm = 43830.50 |
| Boolean | True/False | True/False | True/False |
| String | Unicode String | 16-bit characters | “Maven Analytics” = “MAVEN ANALYTICS” |

Data Types represent how values are stored in DAX storage engine

Formatting represents how values appear to end users (%, date, $, etc.)

# Vertipaq Compression and encoding

* Goal of compression and encoding is to reduce the amount of memory needed to evaluate a DAX query.
* Columnar storage
* Based on sample of data of a column, one or more of the following encoding methods will be used to compress the data
  + Value encoding
  + Uses different mathematical processes to reduce the number of bits needed to store integer values
  + Hash encoding (Dictionary encoding)
  + Identifies the distinct string values and creates a new table with indexes
  + Run length encoding (RLE)
  + Reduces the size of the dataset by identifying repeated values found in adjacent rows
  + Actual storage algorithms are proprietary, so not all details are available

## Value encoding

Value encoding uses a mathematical process to determine relationships between the values in a column, and convert them into smaller values for storage

* Value encoding only works for integer values (including currency), and cannot be applied to strings or floating-point values

## Hash Encoding (Dictionary Encoding)

Hash encoding builds a “dictionary” of distinct items in a column, assigns a unique integer value (index) to each item, and stores the data using the index values rather than the full text strings.

* With hash encoding, storage requirements are defined by the number of unique items in the column (cardinality), NOT by the length of the string values themselves
* Could also be applied to 64-bit integers and floating-point numbers

## Run Length Encoding RLE

Reduces the size of the column by replacing duplicate rows with a table containing each distinct value and the count of instances

* Note: RLE only works when the same value is repeated in consecutive rows, but the Vertipaq engine automatically sorts data on import and refresh to find the optimal compression

## RLE + Hash encoding

Columns can have Run Length and either Hash (dictionary) or Value encoding

* Compression type is determined by cardinality, number of repeat values, row count, and data type

# Vertipaq Relationships

Vertipaq has a special way of mapping relationships between columns in your data model, which allows it to evaluate complex, multi-column queries

NOTE: This is not the same as creating a table relationships in your data model; this is essentially a blueprint that Vertipaq uses to map pairs of primary and foreign keys across related tables.

E.g. => Find Quantity Sold of a specific product

Step 1 – Search in product dictionary to find number from product lookup

Step 2 – Use relationship to find all rows in sales where product line matches above number

Step 3 – Returns a datacache containing filtered sales table and sends to formula engine

Step 4 – Formula engine evaluates the [Quantity Sold] measure against datacache.

# Summary

* DAX uses 2 engines
  + Formula Engine
    - Formula engine interprets and evaluates DAX.
  + Storage Engine
    - Storage engine compresses and encodes
* Storage Engines – both create data cache and send them to formula engine for DAX query evaluation
  + Vertipaq – used for in-memory storage
  + DirectQuery – used for direct connection to external sources
* Vertipaq stores data using a columnar database setup
  + Data is stored in individual columns that can be accessed quickly, but queries that call multiple tables may require more complex logic to produce a datacache.
* Raw data is compressed and encoded to optimize processing.
  + Data can be compressed using ***Value***, ***Hash*** (Dictionary) or ***Run Length*** encoding (RLE) based on cardinality, repeat values, row count, and data type

# Evaluation order

Individual functions typically evaluate from left to right starting with the first parameter (followed by the second, third, etc.)

Nested functions evaluate from the inside out, starting with the innermost function and working outward from there

# DAX Variables

Variables (VAR) are DAX expressions which can be reused multiple times within a query, and are commonly used for 2 key purposes

Readability – makes complex code more human readable

Performance – Only evaluated once no matter how often they are used within a query.

## Variables can

* Simplify and streamline DAX code
* Improve efficiency by eliminating redundant expressions
* Evaluate in the order they’re written (variables can only reference previously declared variables within the query)
* Store either table or scalar values

## Variables cannot

* Start with a number
* Include spaces or special characters (except underscores)
* Share the name of another table in the model
* Be accessed outside of the query in which they are declared
* Contain only certain keywords reserved for DAX (SUM, Date, CALCULATE , etc.)

# Variable Evaluation Order

Variables are “locked in” as soon as the DAX engine reads them; this means you cannot modify how a variable is defined later in your query (i.e. through a CALCULATE function)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AGGREGATION** | **ROUNDING** | **INFORMATION** | **CONVERSION** | **LOGICAL** |
| Dynamically aggregrate values within a column | Round values to different levels of precision | Used to analyze the data type or output of expression | Force a specific data type conversion | Returning information about values in a conditional expression |
| SUM | FLOOR | ISBLANK | CURRENCY | IF |
| AVERAGE | TRUNC | ISERROR | INT | AND |
| MAX | ROUNDDOWN | ISLOGICAL | FORMAT | OR |
| MIN | MROUND | ISNONTEXT | DATE | NOT |
| COUNT | ROUNDUP | ISNUMBER | TIME | TRUE/FALSE |
| COUNTA | CEILING | ISTEXT | DATEVALUE | SWITCH |
| DISTINCTCOUNT | ISO\_CEILING |  | VALUE | COALESCE |
| PRODUCT | ROUNDUP |  |  |  |

# Coalesce function

Replaces “IF + ISBLANK()” pattern, making your code more readable, and can be optimized by the engine.

Returns the first argument or expression that does not evaluate to BLANK. If all arguments/expressions evaluate to BLANK, BLANK is returned.

# Modifiers

|  |  |  |
| --- | --- | --- |
| **Modify Filters** | **Use Relationships** | **Change Filter Propagation** |
| ALL | USERELATIONSHIP | CROSSFILTER |
| ALLSELECTED |  |  |
| ALLNOBLANKROW |  |  |
| ALLEXCEPT |  |  |
| KEEPFILTERS |  |  |
| REMOVEFILTERS |  |  |

# CALCULATE

interprets its filter arguments as table irrespective of Boolean or a function

=CALCULATE(Expression,[Filter1],[Filter2],…)

CALCULATE filter expressions accept both Boolean and table functions (individually or at the same time!), but all filter arguments are automatically converted into a table

E.g. = CALCULATE ([Customer Sales],’Store Lookup’[store\_id]=1)

E.g. = CALCULATE ([Customer Sales],ALL(‘Store Lookup’))

Filter portion is internally read as a table – DAX interprets this as a table.

Any time you write a function that contains a logical statement (IN, <, >, =, etc.) you’re creating a table (internally processed with FILTER & ALL

# REMOVE FILTERS

* Clears filters from specified tables or columns
* Alias for ALL, but can only be used as a CALCULATE modifier (not as a table modifier)

=REMOVEFILTERS(TableNameorColumnName, [ColumnName],[…])

TableNameorColumnName

Table or column you want to clear filter from

E.g. = CALCULATE([Customer Sales],REMOVEFILTERS(‘Store Lookup’))

[ColumnName]

Repeatable column names that allow you to specifically remove filters from individual columns within the base table

E.g. = CALCULATE ([Customer Sales],REMOVEFILTERS(‘Store Lookup’[Store\_id]))

# KEEP FILTERS

Does not remove an existing column or table filter for an individual CALCULATE expression

Allows you to control which specific filters are applied to a calculation

= KEEPFILTERS(Expression)

* KEEPFILTER does not actually modify existing filters, but adds new filter context (think **inner join**)

# Cumulative Total

Cumulative Total =

CALCULATE(

SUM(

‘Sales by Store’[quantity\_sold]

),

FILTER(

ALL(

‘Calendar’[Transaction Date]

),

‘Calendar’[Transaction Date] <= MAX(‘Calendar’[Transaction\_Date]

)

)

# Overall Total

Overall Total =

CALCULATE(

SUMX(

‘Sales by Store’,

‘Sales by Store’[unit\_price] \* ‘Sales by Store’[quantity\_sold]

),

ALL(

‘Sales by Store’

)

)

# Percent of Total

Percent of Total Sales =

VAR CurrentSales =

SUMX(

‘Sales by Store’,

‘Sales by Store’[unit\_price] \* ‘Sales by Store’[quantity\_sold]

)

VAR AllStoreSales =

CALCULATE(

SUMX(

‘Sales by Store’,

‘Sales by Store’[unit\_price] \* ‘Sales by Store’[quantity\_sold]

),

ALL(

‘Sales by Store’

)

VAR Ratio =

DIVIDE(

CurrentSales,

AllStoreSales

)

RETURN

Ratio

# Common Table and Filter Functions

## Filter Data

* + ALL
  + FILTER
  + DISTINCT
  + VALUES
  + ALLEXCEPT
  + ALLSELECTED

## Add Data

* + SELECTCOLUMNS
  + ADDCOLUMNS
  + SUMMARIZE

## Create Data

* + ROW
  + DATATABLE
  + GENERATESERIES

## () Table Constructor

# Calculated Tables

DAX functions with **table** arguments can typically accept either **physical** tables or **calculated**, **virtual** tables (with functions like FILTER, VALUES, etc.)

TOTAL SALES =

Physical Table

SUMX(

‘Sales by Store’,

‘Sales by Store’[quantity\_sold]

)

TOTAL SALES =

Generated by the DAX engine for the duration of the calculation

SUMX(

FILTER(

‘Sales by Store’,

‘Sales by Store’[quantity\_sold]>3

)

‘Sales by Store’[quantity\_sold]

)

# FILTER

=FILTER(Expression, [Filter1],[Filter2]…)

* Returns a filtered table based on one or more filters
* Filter is both a table and an iterator function
  + When using as an iterator, keep in mind the # rows that you want when using filter
  + When calling very large number of rows in a table, it would impede measure performance
  + Used commonly to reduce the number of rows to scan

# ALL

=ALL(Table *or* ColumnName, [ColumnName1]…)

* Returns all of the rows in a table or all of the values in a column and it ignores or clears any initial filters
  + ALL is both a **table** **filter** and a **CALCULATE** modifier
  + Removes initial **filter** **context**

Does not accept **table** **expressions** (only physical table references)

# Review FILTER (Example)

Step 1 – Measure is written

Step 2 – FILTER creates a virtual, calculated table

Sales where Order Qty more than 3 =

CALCULATE(

[Customer Sales],

FILTER(

‘Sales by Store’,

‘Sales by Store’[quantity\_sold]>3

Step 3 – DAX evaluates [Customer Sales] against the filtered table (qty sold > 3)

)

)

# DISTINCT

* Returns a **single column** table of unique values when a **column name** is given.
* If a table is supplied, DISTINCT returns all unique combinations of values

=DISTINCT(ColumnName or TableExpression)

Usually used in combination with another counting measure

ColumnName –**Column** you want to extract unique values from

Table – **Table** you want to extract combination of unique values from

# VALUES

VALUES() returns a ***single column*** table of unique values when a ***column name*** is given.

If a ***table name*** is supplied, VALUES returns the entire table (including duplicates) plus a blank row.

= VALUES(Tablename OR columnName)

Tablename

The table you want to pull all columns and rows from (not unique)

ColumnName

The column you want to extract unique values from

## VALUES VS DISTINCT: THE BLANK ROW

E.g. Sales table has product IDs that are not present in the Product lookup table.

* Instead of throwing an error when a value is missing from a lookup table, the DAX engine adds a **blank row** to the **1** side of the relationship and **not** the **many** side (which will appear in visuals when missing values are present)
* Different table functions handle the presence of this blank row differently; for example, **VALUES** will always show the blank row but **DISTINCT** will not.
  + If you think you might have missing values in your lookup tables (or aren’t sure), use **VALUES** to check.

# SELECTEDVALUE()

*Returns a value when there’s only one value in a specified column, otherwise returns an (optional) alternate result*

= SELECTEDVALUE(ColumnName, [AlternateResult])

ColumnName – Column you want to return a unique value from.

E.g. –

* ‘Customer lookup’[customer\_First\_Name]
* ‘Product Lookup’[product]

[AlternateResult] – Optional – The value returned when there is either no value or more than one value in the specified column (if omitted, blank is returned)

E.g. –

* ‘-‘
* ‘N/A’

SELECTEDV ALUE “SYNTAX SUGAR” – DAX engine internally processes SELECTEDVALUE() as a combination of IF, HASONEVALUE, and VALUES

# ALLEXCEPT()

Removes all report context filters in the table ***except*** the filters applied to the specified columns in the query

= ALLEXCEPT(TableName,[CoulumnName],[ColumnName],[…])

TableName

Name of an existing table (table expressions is not allowed here)

E.g. –

* ‘Sales by Store’
* ‘Returns’
* FILTER(‘Sales by Store’, ‘Sales by Store’[Store\_id] IN {2,5,7,8})

[ColumnName],[ColumnName],[…]

Additional column references within the ***same referenced table*** or a table that is on the ***one-side of the relationship*** (adding additional columns is optional)

E.g. –

* ‘Sales by Store’[Product\_group]
* ‘Sales by Store’[product\_category]
* ‘Calendar’[Transaction\_date]

ALLEXCEPT is typically used as a CALCULATE modifier and not a stand-alone table function

# ALLSELECTED()

Returns all rows in a table or values in a column ignoring filters specified in the query but keeping any other existing filter context

ALLSELECTED(TableNameORColumnName,[ColumnName],[ColumnName],[…])

TableNameorColumnName

Either name of table or column you want to remove filters from (This input is rquired unless being used as a CALCULATE modifier)

E.g. –

* ALLSELECTED
* ‘Sales by Store’
* ‘Sales by Store’[product\_group]

[ColumnName],[ColumnName],[…]

Additional column references within the ***same*** referenced table (adding additional columns is optional)

E.g. –

* ‘Sales by Store’[product\_group]
* ‘Sales by Store’[product\_category]

ALLSELECTED respects existing filter context **except** row and column filters within a visual. This function can be used to obtain visual totals or subtotals in queries.

# SELECTCOLUMNS()

Returns a table with selected columns from the table plus any new columns specified by the DAX expression

SELECTCOLUMNS(Table, Name, Expression, […])

Table – Any DAX expression that returns a table

* E.g. ‘Sales by Store’, FILTER(‘Employees’,’Employees’[ID] IN {2,5,7,9,10}

Name – Name of the new column to be added, must be wrapped in double quotes

* E.g. ‘Employee Name & ID’, ‘Employee Full Name’

Expression – An expression that returns a scalar value (i.e. column reference, integer, or string). Repeatable.

* E.g. ‘Employees’[ID], [First Name] & ‘ ‘ & [Last Name]

SELECTCOLUMNS is an iterator function that’s useful when you need to reduce the number of columns in a table for calculation

SELECTCOLUMNS starts with a blank table whereas ADDCOLUMNS starts with the entire original table and tacks on columns

# ADDCOLUMNS()

Returns a table with selected columns from the table plus any new columns specified by the DAX expression

ADDCOLUMNS(Table, Name, Expression, […])

Table – Any DAX expression that returns a table

* E.g. ‘Sales by Store’, FILTER(‘Employees’,’Employees’[ID] IN {2,5,7,9,10}

Name – Name of the new column to be added, must be wrapped in double quotes

* E.g. ‘Employee Name & ID’, ‘Employee Full Name’

Expression – An expression that returns a scalar value (i.e. column reference, integer, or string). Repeatable.

* E.g. ‘Employees’[ID], [First Name] & ‘ ‘ & [Last Name]

SELECTCOLUMNS starts with a blank table whereas ADDCOLUMNS starts with the entire original table and tacks on columns

# SUMMARIZE()

Creates a summary of the input table grouped by the specified columns

SUMMARIZE(Table, GroupBy\_ColumnName, [Name], [Expression])

Table – any DAX expression that returns a table of data

* E.g. ‘Sales by Store’, FILTER(‘Sales by Store’,’Sales by Store’[product\_id]=16)

GroupBy\_ColumnName – Name of an existing column to be used to create summary groups based on the values found in the column. Cannot be an expression.

* E.g. ‘Sales by Store’[store\_id], ‘Customer Lookup’[customer\_id]

Name,Expression – Depracated and not recommended to be used

SUMMARIZE isn’t an aggregation function. Instead, it returns all unique combinations of values based on the columns selected.

# GENERATING ROWS AND TABLES OF DATA

## ROW()

Returns a single row table with new columns specified by the DAX expression(s)

ROW(Name, Expression, [Name, Expression, […] ] )

Name – Name of the new column, enclosed in double quotes. Repeatable.

* E.g. “Profit”, “Cost”

Expression – Any DAX expression that returns a single scalar value

* E.g. [Customer Sales], SUM(‘Food Inventory’[quantity\_start\_of\_day]

## DATATABLE()

Returns a table containing new, static data

DATATABLE(Name, Type, [Name, Type], […] ], Data)

Name – Column name in quotes

* E.g. “Test Number” , “Value”

Type – Column data type

* E.g. STRING, DOUBLE, INT, CURRENCY

Data – Data points to add to table, called and separated with curly brackets

* E.g. {“first”, 1}, {“second”,2}

DATATABLE can only accept fixed data inputs (no table or column references, expressions or calculations)

## GENERATESERIES()

Returns a single column table containing sequential values, based on a given increment

GENERATESERIES(StartValue, EndValue, [IncrementValue])

Start & End value for the series (can be positive or negative.

* E.g. 0,1,10,25,50,234, etc., -1,-10.75, -45.5, etc.

IncrementValue – Value used to increment between Start and End value (must be non-zero and positive) – default value is 1

* E.g. -25,1,1.7745, 5, etc.

GENERATESERIES is a great way to build al custom range of data to be used as a parameter input

## Table Constructor {}

Can be used to build tables containing one or more columns or rows, based on fixed values or DAX expressions which return scalar values

**NOTE – Column headers can’t be defined using table constructor syntax**

Uses curly brackets to return a table containing one or more columns and records

={scalarExpr1, scalarExpr2,…}

OR

={(scalarExpr1, scalarExpr2, …), (scalarExpr1, scalarExpr2, …)}

# CALCULATED TABLE JOINS

Common use cases:

* Blending or combining data across multiple tables
* Creating advanced calculations like new vs returning users or repeat purchase behavior
* Querying tables to troubleshoot errors or better understand relationships in a data model

## CROSSJOIN

Returns a table contains the cartesian (product of 2 sets, forming a new set containing all ordered pairs) product of the specified tables

CROSSJOIN (Table, Table, […] ] )

Table – List of table expressions to include in the crossjoin

* E.g. ‘Product Lookup’, VALUES(‘Sales by Store’[store\_id]

|  |  |
| --- | --- |
| Product Category | Product Group |
| Coffee | Beverages |
| Bakery | Food |

|  |
| --- |
| Store ID |
| 3 |
| 5 |

CROSSJOIN

|  |  |  |
| --- | --- | --- |
| Product Category | Product Group | Store ID |
| Coffee | Beverages | 3 |
| Bakery | Food | 3 |
| Coffee | Beverages | 5 |
| Bakery | Food | 5 |

Resulting table contains 4 rows (2 \* 2) and 3 columns (2+1)

All column names must be different in all of the tables.

## UNION

Combines or “Stacks” rows from two or more tables sharing the same column structure

UNION(Table, Table, […] ] )

Table – Accepts any DAX expression for two (or more) tables with **identical column structure**

* E.g. ‘Sales Target 2019’, ‘Sales Target 2020’j, ‘Sales Target 2019’, DATATABLE()

All tables must contain the same number of columns

Columns are combined based on their positioning in their respective tables

Column names are determined by the first table expression

## EXCEPT

Returns all rows from the left table which do not appear in the right table

EXCEPT(LeftTable, RightTable)

LeftTable and RightTable – tables that will be used for joining

First table must be a table within the data model

* E.g. (Churned Customers ): =EXCEPT(‘Customers 2019’,’Customers 2020’)

Both tables must contain the same number of columns

Columns are combined based on positioning in their respective tables

Column names are determined by the left table

The resulting table does NOT retain relationships to other tables (can’t be used as an expanded table)

## INTERSECT

Returns all the rows from the left table which also appear in the right table

INTERSECT(LeftTable,RightTable)

LeftTable and RightTable – Tables that will be used for joining

First table must be a table within the data model

* E.g. (previous month active customers): = INTERSECT( VALUES(‘Sales’[Customer ID]), CALCULATETABLE( VALUES (‘Sales’[Customer ID]), DATEADD (‘Calendar’[Date],-1,MONTH))

Order matters! (T1, T2) may have a different result set than (T2, T1)

Columns are compared based on positioning in their respective tables

Duplicate rows are retained

Column names are determined by the left table

The resulting table does NOT retain relationships to other tables (can’t be used as an expanded table)