Power BI for Business Intelligence

DAX Cheat Sheet

> Math & statistical functions

- **SUM(<column>)** Adds all the numbers in a column.
- **SUMX(<table>, <expression>)** Returns the sum of an expression evaluated for each row in a table.
- **AVERAGE(<column>)** Returns the average (arithmetic mean) of all the numbers in a column.
- **AVERAGEX(<table>, <expression>)** Calculates the average (arithmetic mean) of a set of expressions evaluated over a table.
- **MEDIAN(<column>)** Returns the median of a column.
- **MEDIANX(<table>, <expression>)** Calculates the median of a set of expressions evaluated over a table.
- **GEOMEAN(<column>)** Calculates the geometric mean of a column.
- **GEOMEANX(<table>, <expression>)** Calculates the geometric mean of a set of expressions evaluated over a table.
- **COUNT(<column>)** Returns the number of cells in a column that contain non-blank values.
- **COUNTX(<table>, <expression>)** Counts the number of rows from an expression that evaluates to a non-blank value.
- **DIVIDE(<numerator>, <denominator> [,<alternateresult>])** Performs division and returns alternate result or BLANK() on division by 0.
- **MIN(<column>)** Returns a minimum value of a column.
- **MAX(<column>)** Returns a maximum value of a column.
- **COUNTROWS([<table>])** Counts the number of rows in a table.
- **DISTINCTCOUNT(<column>)** Counts the number of distinct values in a column.
- **RANKX(<table>, <expression>[, <value>[, <order>[, <ties>]]])** Returns the ranking of a number in a list of numbers for each row in the table argument.

> Filter functions

- **FILTER(<table>, <filter>)** Returns a table that is a subset of another table or expression.
- **CALCULATE(<expression>[, <filter1> [, <filter2> [, …]]])** Evaluates an expression in a filter context.
- **HASONEVALUE(<columnName>)** Returns TRUE when the context for columnName has been filtered down to one distinct value only. Otherwise it is FALSE.
- **ALLNOBLANKROW(<table> | <column>[, <column>[, <column>[,…]]])** Returns a table that is a subset of another table or expression.
- **ALL([<table> | <column>[, <column>[, <column>[,…]]]])** Returns all the rows in a table, or all the values in a column, ignoring any filters that might have been applied.
- **ALLEXCEPT(<table>, <column>[, <column>[,..]])** Returns all the rows in a table except for those rows that are affected by the specified column filters.
- **REMOVEFILTERS([<table> | <column>][, <column>[, <column>[,…]]])** Clear all filters from designated tables or columns.

> Logical functions

- **IF(<logical_test>, <value_if_true>[, <value_if_false>])** Checks a condition, and returns a certain value depending on whether it is true or false.
- **AND(<logical 1>, <logical 2>)** Checks whether both arguments are TRUE, and returns TRUE if both arguments are TRUE. Otherwise, it returns FALSE.
- **OR(<logical 1>, <logical 2>)** Checks whether one of the arguments is TRUE to return TRUE. The function returns FALSE if both arguments are FALSE.
- **NOT(<logical>)** Changes TRUE to FALSE and vice versa.
- **SWITCH(<expression>, <value>, <result>[, <value>, <result>]…[, <else>])** Evaluates an expression against a list of values and returns one of possible results
- **IFERROR(<value>, <value_if_error>)** Returns value_if_error if the first expression is an error and the value of the expression itself otherwise.

> Date & time functions

- **CALENDAR(<start_date>, <end_date>)** Returns a table with a single column named "Date" that contains a contiguous set of dates.
- **DATE(<year>, <month>, <day>)** Returns the specified date in datetime format.
- **DATEDIFF(<date_1>, <date_2>, <interval>)** Returns the number of units between two dates as defined in **<interval>**.
- **DATEVALUE(<date_text>)** Converts a date in text to a date in datetime format.
- **DAY(<date>)** Returns a number from 1 to 31 representing the day of the month.
- **WEEKNUM(<date>)** Returns weeknumber in the year.
- **MONTH(<date>)** Returns a number from 1 to 12 representing a month.
- **QUARTER(<date>)** Returns a number from 1 to 4 representing a quarter.

> Time intelligence functions

- **DATEADD(<dates>, <number_of_intervals>, <interval>)** Moves a date by a specific interval.
- **DATESBETWEEN(<dates>, <date_1>, <date_2>)** Returns the dates between specified dates.
- **TOTALYTD(<expression>, <dates>[, <filter>][, <year_end_date>])** Evaluates the year-to-date value of the expression in the current context.
- **SAMEPERIODLASTYEAR(<dates>)** Returns a table that contains a column of dates shifted one year back in time.
- **STARTOFMONTH(<dates>) // ENDOFMONTH(<dates>)** Returns the start // end of the month.
- **STARTOFQUARTER(<dates>) // ENDOFQUARTER(<dates>)** Returns the start // end of the quarter.
- **STARTOFYEAR(<dates>) // ENDOFYEAR(<dates>)** Returns the start // end of the quarter.

> Relationship functions

- **CROSSFILTER(<left_column>, <right_column>, <crossfiltertype>)** Specifies the cross-filtering direction to be used in a calculation.
- **RELATED(<column>)** Returns a related value from another table.

> Table manipulation functions

- **SUMMARIZE(<table>, <groupBy_columnName>[, <groupBy_columnName>]…[, <name>, <expression>]…)** Returns a summary table for the requested totals over a set of groups.
- **DISTINCT(<table>)** Returns a table by removing duplicate rows from another table or expression.
- **ADDCOLUMNS(<table>, <name>, <expression>[, <name>, <expression>]…)** Adds calculated columns to the given table or table expression.
- **SELECTCOLUMNS(<table>, <name>, <expression>[, <name>, <expression>]…)** Selects calculated columns from the given table or table expression.
- **GROUPBY(<table> [, <groupBy_columnName>[, [<column_name>] [<expression>]]…)** Create a summary of the input table grouped by specific columns.
- **INTERSECT(<left_table>, <right_table>)** Returns the rows of the left-side table that appear in the right-side table.
- **NATURALINNERJOIN(<left_table>, <right_table>)** Joins two tables using an inner join.
- **NATURALLEFTOUTERJOIN(<left_table>, <right_table>)** Joins two tables using a left outer join.
- **UNION(<table>, <table>[, <table> [,…]])** Returns the union of tables with matching columns.

> Text functions

- **EXACT(<text_1>, <text_2>)** Checks if two strings are identical (**EXACT()** is case sensitive).
- **FIND(<text_tofind>, <in_text>)** Returns the starting position a text within another text (**FIND()** is case sensitive).
- **FORMAT(<value>, <format>)** Converts a value to a text in the specified number format.
- **LEFT(<text>, <num_chars>)** Returns the number of characters from the start of a string.
- **RIGHT(<text>, <num_chars>)** Returns the number of characters from the end of a string.
- **LEN(<text>)** Returns the number of characters in a string of text.
- **LOWER(<text>)** Converts all letters in a string to lowercase.
- **UPPER(<text>)** Converts all letters in a string to uppercase.
- **TRIM(<text>)** Remove all spaces from a text string.
- **CONCATENATE(<text_1>, <text_2>)** Joins two strings together into one string.
- **SUBSTITUTE(<text>, <old_text>, <new_text>, <instance_num>)** Replaces existing text with new text in a string.
- **REPLACE(<old_text>, <start_posotion>, <num_chars>, <new_text>)** Replaces part of a string with a new string.

> Information functions

- **COLUMNSTATISTICS()** Returns statistics regarding every column in every table. This function has no arguments.
- **NAMEOF(<value>)** Returns the column or measure name of a value.
- **ISBLANK(<value>) // ISERROR(<value>)** Returns whether the value is blank // an error.
- **ISLOGICAL(<value>)** Checks whether a value is logical or not.
- **ISNUMBER(<value>)** Checks whether a value is a number or not.
- **ISFILTERED(<table> | <column>)** Returns true when there are direct filters on a column.
- **ISCROSSFILTERED(<table> | <column>)** Returns true when there are crossfilters on a column.
- **USERPRINCIPALNAME()** Returns the user principal name or email address. This function has no arguments.

> DAX statements

- **VAR(<name> = <expression>)** Stores the result of an expression as a named variable. To return the variable, use RETURN after the variable is defined.
- **COLUMN(<table>[<column>] = <expression>)** Stores the result of an expression as a column in a table.
- **ORDER BY(<table>[<column>])** Defines the sort order of a column. Every column can be sorted in ascending (**ASC**) or descending (**DESC**) way.

> DAX Operators

Comparison operators	Meaning
=	Equal to
= =	Strict equal to
>	Greater than
<	Smaller than
> =	Greater than or equal to
= <	Smaller than or equal to
< >	Not equal to

Text operator	Meaning	Example
&	Concatenates text values	Concatenates text values | [City]&", "&[State]

Logical operator	Meaning	Example
&&	AND condition	([City] = "Bru") && ([Return] = "Yes"))
||	OR condition	([City] = "Bru") || ([Return] = "Yes"))
IN {}	OR condition for each row	Product[Color] IN {"Red", "Blue", "Gold"}

DAX syntax

A DAX formula always starts with an equal operator (=). After the equals sign, you can provide any expression that evaluates to a scalar or can be converted to a scalar. Let us understand the syntax with an example by breaking down the following DAX measure formula:

```
1   Total Sales = SUM (Sales[Sales Amount])
```

Total Sales: The name of the calculated column.

Equal operator: Indicates the beginning of the DAX formula.

SUM: An aggregation function of DAX

Parenthesis ( ): Grouping of arguments

Sales: Table to be referenced

Square brackets [Sales Amount]: The square brackets contain the referenced column, which is also the argument. The value of this argument must be passed to the function.

Data types in DAX

DAX can perform computations on different data types, which include the following:

Text (Binary): "Hello, world".

Decimal (Float): 1.23.

Whole number (Integer): 123.

Boolean: TRUE or FALSE.

Date (Date/Time): DATE(2023,5,11).

Currency: A fixed decimal number.

DAX Operators

DAX formulas rely on operators to perform arithmetic calculations, compare values, work with strings, or test conditions.

Here is an overview of some commonly used operators in DAX:

Operator Type	Symbol	Application	Example
Parenthesis	`()`	Grouping of arguments and precedence order	`(5+7) * 5`
Arithmetic	`+`	Addition	`5 + 3 = 8`
	`-`	Subtraction	`9 - 5 = 4`
	`*`	Multiplication	`3 * 9 = 27`
	`/`	Division	`18/3 = 6`
	`^`	Exponentiation	`16^4 = 65536`
Logical	`&&`	AND condition between two Boolean expressions	`[Region] = "USA" && [Quantity] > 5`
	`||`	OR condition between two Boolean expressions	`[Region] = "USA" || [Quantity] > 5`
Comparison	`=`	Equal to	`[Region] = "USA"`
	`<>`	Not equal to	`Region] <> "USA"`
	`>`	Greater than	`[Quantity] > 5`
	`>=`	Greater than or equal to	`[Quantity] >= 10`
	`<`	Less than	`[Quantity] < 5`

Operator Type	Symbol	Application	Example
	<=	Less than or equal to	`[Quantity] <= 5`
Text Concatenation	&	Concatenation of strings	`[Region] & ", " & [City]`

DAX Functions

A function is a named formula within an expression. Most functions have required and optional arguments, also called parameters, as input. When the function is executed, a value is returned.

DAX includes functions to perform calculations using dates and times, create conditional values, work with strings, perform lookups based on relationships, and iterate over a table to perform recursive calculations. Some of the most used classes of DAX functions are given below.

Text Functions

You can use these functions to return part of a string, search for text within a string or concatenate string values to create a new column.

With the **CONCATENATE** function, you can join two text strings into one text string. For example, you can combine an employee's first name and last name into a new column by defining a DAX formula as follows:

```
1    Full Name = CONCATENATE (Employees[FirstName], CONCATENATE ( " ", Employees[LastName] ) )
```

The **LEFT** function returns the leftmost characters from a text value. For example, you could create a column that shortens the names of the months for better visualization purposes. The **LEFT** function allows you to create a column with only the first three letters of each month as follows:

```
1    Short Name = LEFT (Date[Month], 3)
```

Date/Time Functions

These functions in DAX are like date and time functions in Microsoft Excel. However, DAX functions are based on the datetime data types used by Microsoft SQL Server.

NOW(): The NOW function displays the current date and time on a worksheet or calculates a value based on the current date and time. It updates the value each time you open the worksheet.

YEAR(<date>): Returns the year of a date as a four-digit integer from the date column. You can add a column for a year from your date table.

MONTH(<date>): Returns the month of a date as a number (1 - 12).

Logical Functions

These functions evaluate logical conditions and return true or false values.

IF(<logical_test>, <value_if_true>, <value_if_false>): Returns one value if a condition is true and another value if it is false.

AND(<logical1>, <logical2>): Returns TRUE if all its arguments are TRUE; returns FALSE if one or more argument is FALSE.

OR(<logical1>, <logical2>): Returns TRUE if any argument is TRUE; returns FALSE if all arguments are FALSE.

NOT(<logical>): Reverses the logic of its argument.

Aggregation Functions

These functions perform aggregations. Commonly these functions create sums and averages and find minimum and maximum values. You can also filter a column in DAX based on related tables, before creating aggregations.
Common aggregation functions are:
COUNT(<column>): Returns the count, or total, of all rows in a column.

SUM(<column>): Returns the sum of all values in a column.

AVERAGE(<column>): Returns the average of all values in a column.

MIN(<column>): Returns the smallest value in a column.

MAX(<column>): Returns the largest value in a column.

Time Intelligence Functions
These functions create calculations using built-in knowledge of calendars and dates. You can build meaningful comparisons across comparable periods for sales, inventory, and so on using time and date ranges combined with aggregations or calculations.
SAMEPERIODLASTYEAR(<dates>): Returns a parallel period calculated against the dates provided.

DATESYTD(<dates>): Returns dates from the beginning of the year until the last date in the dates column provided.
DATESMTD(<dates>): Returns dates from the beginning of the month until the last date in the provided dates column.
DATESQTD(<dates>): Returns dates from the beginning of the quarter until the last date in the provided dates column.
EDATE(<start_date>, <months>): Returns the date that is a specified number of months before or after the start date.

Statistical Functions
These functions calculate values related to statistical distributions and probability, such as standard deviation and number of permutations. Common statistical functions are:
STDEV.P(<ColumnName>): Returns the standard deviation of the entire population.

MEDIAN(<column>): Returns the median of numbers in a column.

RANKX(<table>, <expression>[, <value>[, <order>[, <ties>]]]): Returns the ranking of a number in a list of numbers for each row in the table argument.

Relational Functions
These functions are for managing and utilizing relationships between tables. For example, you can specify a particular relationship to be used in a calculation.
RELATED(<column>): Returns a related value from another table.

RELATEDTABLE: Evaluates a table expression in a context modified by the given filters.

Information Functions
These functions look at a table or column provided as an argument to another function and determine if the value matches the expected type.
ISBLANK(<expression>): Checks if a value is blank and returns TRUE or FALSE.

CONTAINS(<table>, <column>, <value>): Checks if the values in a column already exist in another column and returns a value of TRUE or FALSE.

Points to remember:
DAX is not case-sensitive but distinguishes between blanks and zeros.

Use comments to explain your code. You can use // for a single-line comment and /* ... */ for a multi-line comment.
Remember, many DAX functions require an existing relationship between tables, so ensure your data model is set up correctly.
Study row and filter context, as they are fundamental to understanding and using DAX effectively.

In this reading, you'll explore six important statistical functions that you'll make frequent use of in DAX expressions. These are **AVERAGE**, **MEDIAN**, **COUNT**, **DISTINCTCOUNT**, **MIN**, and **MAX**. You'll also learn how they are used through practical examples.

The AVERAGE function

The AVERAGE function, also known as the mean, sums up all the numbers in a dataset and divides the result by the total count of numbers. This function is frequently used to identify a central tendency in a dataset. It is beneficial when you need to find the middle ground or commonality within data.

Example: You want to analyze the average quantities of a product sold over a specified period to determine the typical sales volume for Adventure Works.

```
1    Avg. Of Quantities Sold = AVERAGE (Sales [Quantity])
```

In this example, **Sales** is the table name, and **Quantity** is the column name that contains the numbers for which you want the average.

Related Functions:

There are several functions related to AVERAGE.

AVERAGEX

This function calculates the average (arithmetic mean) of a set of expressions evaluated over a table.

```
1    = AVERAGEX (Sales, Sales [Quantity] * Sales [Price])
```

In the above example, the **AVERAGEX** function is applied to the **Sales** table. The expression within the function multiplies the quantity sold by the price for each row in the **Sales** table, giving the sales amount for each product. Then the function calculates the average of these sales amounts.

AVERAGEA

This function returns the values' average (arithmetic mean) in a column. It handles text and non-numeric values. Whenever there are no rows to aggregate, the function returns a blank.

```
1    = AVERAGEA([Amount])
```

The above example returns the average of non-blank cells in the referenced column.

The MEDIAN function

The MEDIAN function calculates the middle value in a set of numbers. It sorts the numbers in ascending order and then selects the middle number. The median is the average of the two middle numbers for datasets with an even number of observations.

Unlike the average, the median is less affected by outliers and extreme values. This means that it's useful for datasets with skewed distributions. Only numeric data types are supported in this function. Dates, logical values, and text columns are not supported.

Example: You are analyzing customer data at Adventure Works. You can easily analyze customers' locations from the dimension attributes. However, you also need to identify the median age of customers. You can perform this calculation by using the DAX median function as follows:

```
1    Median Customer Age = MEDIAN(Customers [Age])
```

Customers is the table name, and the **Age** column contains the numbers you require for the median **customer age** column.

The COUNT function

The COUNT function counts the number of rows in a column or a table. It is often used to measure the size of a dataset. You can use it to count all or only rows meeting specific criteria. The only argument in the function is the column. When the function finds no rows to count, it returns a blank.

Example: You need to identify the total number of customers in a particular country or city. In this case, you can use the **COUNT DAX** function.

```
1    Number of Customers = COUNT([CustomerID])
```

CustomerID is the column name that contains the values to be counted.

Related functions:

The **COUNT** function has several related functions.

COUNTA

The **COUNTA** functions count the number of cells in a column that are not empty. It counts rows containing numeric and non-blank values, including text, dates, and logical values.

```
1    = COUNTA ('Reseller' [Phone])
```

The above function returns all rows in the reseller table with any value in the column that stores phone numbers.

COUNTAX

The **COUNTAX** function counts non-blank results when evaluating the result of an expression over a table. It works just like the **COUNTA** function but is used to iterate through the rows in a table and count rows where the specified expressions result in a non-blank result.

```
1    COUNTAX(<table>, <expression>)= COUNTAX (FILTER (Reseller, [Status] = "Active"), [Phone])
```

In the above example, the function counts the number of non-blank rows in the **phone** column while using the table that results from filtering the reseller table on Status = active.

COUNTBLANK

This function counts the number of blank cells in a column. If no rows are found that meet the condition, blanks are returned.

```
1    = COUNTBLANK (Reseller [BankName])
```

The above function returns the blank values for the **BankName** column in the **Reseller** table.

COUNTROWS

The **COUNTROWS** function counts the number of rows in the specified table or a table defined by an expression.

```
1    = COUNTROWS(['Orders'])
```

The above example shows how to count the number of rows in the **Orders** table, while the following example counts rows from the **resellersales** table.

```
1    = COUNTROWS(RELATEDTABLE(ResellerSales))
```

The DISTINCTCOUNT function

The **DISTINCTCOUNT** function counts the number of distinct values in a dataset. This function is helpful when you need to understand the count of unique values or categories.

The only argument allowed for this function is a column. You can use columns containing any type of data. When the function finds no rows to count, it returns a **BLANK**. Otherwise, it returns the count of distinct values.

Example: You have a sales dataset that records **customerID**, **products**, and **sales amount**. The **COUNT** function gives you the total counts, but **DISTINCTCOUNT** provides information about the customers' distinct counts and associated purchases.

```
1    Number of distinct Customers = DISTINCTCOUNT([CustomerID])
```

In this code, **CustomerID** is the column name that contains values to be counted.

The MIN function

The MIN function is used to identify the smallest values in a column or between two scalar expressions. These values provide an overview of the range of your data.

Example: Adventure Works records data on resellers. The company's resellers operate on specific margin values. You must calculate and generate insights on the reseller margin values as a calculated column. You can use the **MIN** function to generate data on the minimum reseller margin within your data.

```
1    Minimum Reseller Margin =MIN([Reseller Margin])
```

In this example, **Reseller Margin** is the column name that contains values to be evaluated.

Related functions:

There are several other functions related to the **MIN** function:

MINA

This function returns the smallest value in a column. It does not ignore logical values and text. When **MINA** operates with Boolean data types, it considers **TRUE** as **1** and **FALSE** as **0**.

```
1    = MINA (Sales [Freight])
```

The above expression returns the minimum freight charge from the **Sales** table.

MINX

This function returns the smallest value that results from evaluating an expression for each table row. The function takes a table or an expression that returns a table as its first argument. The second argument contains the expression evaluated for each row of the table.

```
1    = MINX (FILTER (Sales, [SalesTerritoryKey] = 5), [Freight])
```

The above example filters the **Sales** table and returns only rows for a specific sales territory. The formula then finds the minimum value in the **Freight** column.

The MAX function

The **MAX** function is used to identify the largest value in a column or the larger value between two scalar expressions. The **MIN** and **MAX** functions can provide an overview of the range of your data.

Example: The Adventure Works **Sales** dataset contains a **Sales Amount** column. This column represents sales for different transactions. You can use the **MAX** function to find the maximum sales amounts.

```
1    Max Sales Amount = MAX (Sales [Sales Amount])
```

Sales is the name of the table, and **Sales Amount** is the column name that contains the values to be evaluated.

Related functions:

There are several other functions related to the **MAX** function:

MAXA

This function returns the largest value in a column and does not ignore logical values and text. Therefore, this function can also be used in the date column.

```
1    = MAXA([Margin]
```

The following example returns the greatest value from a calculated column named **Margin**.

MAXX

This function evaluates an expression for each table row and returns the largest value. The table argument to the **MAXX** function can be a table name or an expression that evaluates a table. The second argument indicates the expression to be evaluated for each table row.

```
1    = MAXX (Sales, Sales [TaxAmt]+ Sales [Freight])
```

The following formula uses an expression as the second argument to calculate the total taxes and shipping amount for each order in the **Sales** table.

Syntax

Let's briefly recap the **CALCULATE** syntax with the following example:

```
CALCULATE(<expression>[, <filter1> [, <filter2>])
```

The syntax of this DAX formula begins with the **CALCULATE** keyword, followed by the argument in parentheses. The argument contains an expression and one or more filters. These filters are one of the key elements of the **CALCULATE** function. As you learned earlier in this course, filters can be used to retrieve greater levels of granularity from your data.

Let's review the different types of filters and how they work inside the **CALCULATE** function.

Boolean filter expressions

A Boolean filter expression evaluates as either **TRUE** or **FALSE**. In other words, the statement returns a positive or negative value.

There are several rules that Boolean filter expressions must abide by:

They can reference columns from a single table.

They cannot reference measures.

They cannot use a nested **CALCULATE** function.

They cannot use functions that scan or return a table, including aggregation functions.

For example, Adventure Works wants to calculate the total sales of high-end products, defined as products whose unit price is equal to or greater than **$500**. This calculation can be performed using a Boolean filter with **CALCULATE** in DAX as follows:

```
Sales of high-end products =
CALCULATE (
    SUM ( Sales[Total Sales] ),
    FILTER ( Products, Products[Unit Price] >= 500 )
)
```

The measure makes use of a **FILTER** function to apply its second argument. This argument checks the **Unit Price** column of the **Products** table and evaluates all records as either True or False. Any records with a value greater than, or equal to, **$500** are evaluated as **TRUE**. So, this formula filters all **TRU**E records and returns a table listing all products whose unit price is greater than or equal to **$500**.

Table filter expressions

Table filter expressions implement a table as a filter. Such an expression could reference a table in the data model, but more likely, it is a function that returns a table object.

You can also use the **FILTER** function to apply complex filter conditions, including those that a Boolean filter expression cannot define.

For example, Adventure Works wants to generate its total sales in **Germany** for 2019. To calculate this formula, Adventure Works must use two table filters. The first filter is from the **Region** table, and the second is from the **Date** table as follows:

```
2019 Sales in Germany =
CALCULATE (
    SUM ( [Total Sales] ),
    FILTER ( Region[Country] = "Germany", Date[Year] = "2019" )
)
```

In this **CALCULATE** statement, the **FILTER** function identifies all instances of **Germany** in the **Country** column of the **Region** table. It also identifies all instances of **2019** in the **Year** column of the **Date** table. It then returns these results as a calculated table that computes the total sales for Germany in 2019.

Filter modification functions

The filter modifier functions are used for more than adding filters. They provide additional control when modifying a filter context. For example, when you add a filter to the **CALCULATE** statement, **CALCULATE** overrides any existing filters previously created in the column. You must use filter modifier functions to ensure that **CALCULATE** adds the new filter to any previous filter.

Here's an overview of some of the most common filter modification functions used within the **CALCULATE** function of DAX.

REMOVEFILTERS

The **REMOVEFILTERS** function is used to remove all filters or remove filters from one or more columns of a table or all columns of a single table.

For example, a **Product** and **Region** filter has been applied to the tables in theAdventure Worksdatabase.You want to analyze the company's total sales without any filters applied to the **Sales** table. You can write a DAX expression as follows:

```
Total Sales =
CALCULATE
(
    [Total Sales],
    REMOVEFILTERS ( Product ),
    REMOVEFILTERS ( Region)
)
```

REMOVEFILTERS eliminates all filters from the tables specified within the parentheses. In this case, it removes filters from the **Product** and **Region** dimensions.

KEEPFILTERS

KEEPFILTERS adds a filter without removing existing filters on the same columns. In other words, you can keep the existing filters and still evaluate the expression.

For example, Adventure Works must calculate its total sales while keeping the existing color filter on the **Product** table. The required DAX expression is as follows:

```
Blue Products Sale =
CALCULATE
(
    [Total Sales],
    KEEPFILTERS ( Products[Color] = "Blue" )
)
```

If an existing active filter exists on **Product[Color]**, then **KEEPFILTERS** ensures the filter is not overridden. Instead, it is merged with the new filter.

ALL

The **ALL** family of functions (**ALLEXCEPT, ALLSELECTED, ALLNOBLANKROW**) removes filters from one or more columns. They can also be used to remove all columns of a single table.

This function acts in an equivalent way to **REMOVEFILTERS**.

For example, Adventure Works can use the **ALL** function to calculate total sales for every region as follows:

```
Total Sales =
CALCULATE
(
    [Total Sales],
    ALL ( Product ),
    ALL ( Region)
```

```
    ALL ( Region)
)
```

In this formula, the **ALL** function returns the required results as a table or a column, which can be used in other DAX functions. **REMOVEFILTERS** does not return a table or column. Instead, it just removes filters from the specified tables or columns.

CROSSFILTER

The **CROSSFILTER** function modifies the filter direction (from **Both** to **Single**, or from **Single** to **Both**). It can also be used to disable a relationship between tables.

Adventure Works wants to analyze its total number of products by a specific year. However, the default cross-filter direction of the data model does not allow the company to compute the value. So, Adventure Works can create a measure using DAX without changing the default cross-filter direction in the data model as follows:

```
Product by Year =
CALCULATE
(
    DISTINCTCOUNT ( Products[ProductKey] ),
    CROSSFILTER
        ( Sales[ProductKey],
        Products[ProductKey],
        BOTH )
)
```

The formula analyzes the measure based on the **Year** column from the **Date** table and returns accurate results according to the analytical needs of the business.

USERELATIONSHIP

The **USERELATIONSHIP** function engages an inactive relationship between related columns. This means that the active relationship will automatically become inactive.

By default, Power BI utilizes the active relationship for all analysis and visualization. However, there may be times when your analysis requires using the inactive relationship. In this instance, you can use **USERELATIONSHIP**.

For example, Adventure Works wants to calculate its total sales using the **Shipping Date** column from the **Sales** table and the **Date** column from the **Date** table. This relationship is currently inactive. So, Adventure Works can use the **USERELATIONSHIP** function within the **CALCULATE** function as follows:

```
Sales by Shipping date =
CALCULATE (
    SUM ( Sales[SalesAmount] ),
    USERELATIONSHIP ( Sales[ShippingDate], Date[Date] )
)
```

In this formula, the **USERELATIONSHIP** function only switches the default relationship between related tables for the current measures. Remember that this approach typically handles inactive relationships within role-playing dimensions.

Next, let’s explore some examples of these functions and discover how they can be used to generate time intelligence insights.

PREVIOUSYEAR

These functions return all dates from the previous year. They can be instrumental when making Y-o-Y (Year Over Year) comparisons. Other functions in this group include **PREVIOUSMONTH**, **PREVIOUSQUARTER**, and **PREVIOUSDAY**. These functions are used for the historical evaluation of data.

```
1    PREVIOUSYEAR(<dates>[,<year_end_date>])
```

<dates> is a column containing dates.

<year_end_date> is an optional parameter that defines the year-end date.

Example: Adventure Works can use the **PREVIOUSMONTH** function to compare this month's sales with those of the previous month. Such a comparison can reveal short-term trends or the immediate impact of any changes on the business strategy.

NEXTYEAR

The **NEXTYEAR** function is the forward-looking counterpart to **PREVIOUSYEAR**. The other functions of the group include **NEXTMONTH**, **NEXTQUARTER**, and **NEXTDAY**. These functions are used in projections and forecasts.

The function returns a table that contains a column of all dates in the next year, based on the first date in the dates column in the current context.

```
1    NEXTYEAR(<dates>[,<year_end_date>])
```

<dates> is a column containing dates.

<year_end_date> is an optional parameter that defines the year-end date.

Example: If Adventure Works has monthly sales targets for its sales team, it could use **NEXTMONTH** to project whether the team is on track to meet those goals based on the current month's data.

TOTALYTD

The year-to-date calculation is an aggregation of values from the beginning of the year to the specified date. **YTD (year-to-date)** can summarize all sales from January 1st of that year to the specified date.

```
1    TOTALYTD(<expression>, <dates>, [, <filter>][, <year_end_date>])
```

<expression> returns a scalar value.

<dates> is the date column. In this current lesson, you’re using Power BI's default **date** dimension.

<filter> and **<year_end_date>** are optional parameters.

Example: Adventure Works wants to evaluate its real-time sales performance. To compute this measure, you can use the **TOTALYTD** function in DAX. **TOTALYTD** is a simple function to calculate year-to-date values. In this case, you can calculate the **YTDSales** from the **Total sales** column of the **Sales table**.

DATESBETWEEN

This function returns a table that contains all dates between a specified start date and an end date.

```
1   DATESBETWEEN(<dates>, <start_date>, <end_date>)
```

<dates> is the column containing the dates.

<start_date> is the date expression at the start of the calculation.

<end_date> is the date expression that contains the last date for the calculation.

Example: Adventure Works wants to evaluate its summer sales. To achieve this, it must create a measure using the **DATESBETWEEN** function in DAX. It can enter (summer months) June 1, 2018, as the start date and August 31, 2018, as the end date for the function's parameters.

PARALLELPERIOD

The **PARALLELPERIOD** function returns a set of dates separated from those in the specified column by specific intervals (such as days, months, quarters, and years). In a business context, it is often used to compare periods from previous years.

```
1   PARALLELPERIOD(<dates>,<number_of_intervals>,<interval>)
```

<dates> is the column containing the dates.

The **<number_of_intervals>** is the integer value that defines the number of intervals to add or subtract from the date.

<interval> is the unit of time to shift the date. It can be year, quarter, or month.

Example: Adventure Works can use **PARALLELPERIOD** to compare this year's sales with those from two years ago. If sales have risen significantly since then, that may indicate successful strategies or growth in the customer base. Conversely, if sales have stagnated or declined, it might signal a need for changes.

SAMEPERIODLASTYEAR

SAMEPERIODLASTYEAR is a time intelligence function that compares the value for the same period in the previous year. This function is used frequently in retail and e-commerce for year-over-year (YoY) comparisons.

```
1   SAMEPERIODLASTYEAR(<dates>)
```

Example: Suppose Adventure Works wants to compare Q2 sales of this year with the same period last year. The **SAMEPERIODLASTYEAR** function executes this calculation swiftly. This comparison can highlight seasonal trends, marketing campaign effectiveness, and overall business health.

CLOSINGBALANCEYEAR

This function evaluates the **expression** at the last date of the year in the current context.

```
CLOSINGBALANCEYEAR(<expression>,<dates>[,<filter>][,<year_end_date>])
```

<expression> is an expression that returns a scalar value.

<dates> is a column containing dates.

<filter> is an expression that specifies a filter to apply to the current context (this is an optional parameter).

<year_end_date> is an optional parameter that defines the year-end date.

Example: Adventure Works can utilize this function to compute the year-end inventory for each product in categories and subcategories. This helps the company to plan its supply chain.