

Domain 3: Model data with Power BI

Configure a semantic model

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/configure-semantic-model-power-bi/1-introduction>

Introduction

Completed

This module explains how to design a semantic model, which is the task you undertake as a data modeler once you apply your Power Query queries. At this point in your development, you have a model comprising one or possibly more tables—but there's more work to be done. Your goal is to develop a model that supports the reporting requirements, and that's user-friendly and intuitive. Essentially, you're developing a *semantic layer* over the data.

Your first design efforts should focus on configuring the relationships between model tables. You can then move on to enhance the model design by setting table and column properties, and by creating other model objects, like hierarchies, measures, and parameters.

Note

This module focuses on Import model designs. Therefore, it doesn't cover working with different model frameworks, like DirectQuery and composite models. Also, it doesn't cover user-defined aggregations, row-level security (RLS), or development tasks that involve writing Data Analysis Expressions (DAX) formulas.

2. Configure relationships

<https://learn.microsoft.com/en-us/training/modules/configure-semantic-model-power-bi/2-relationships>

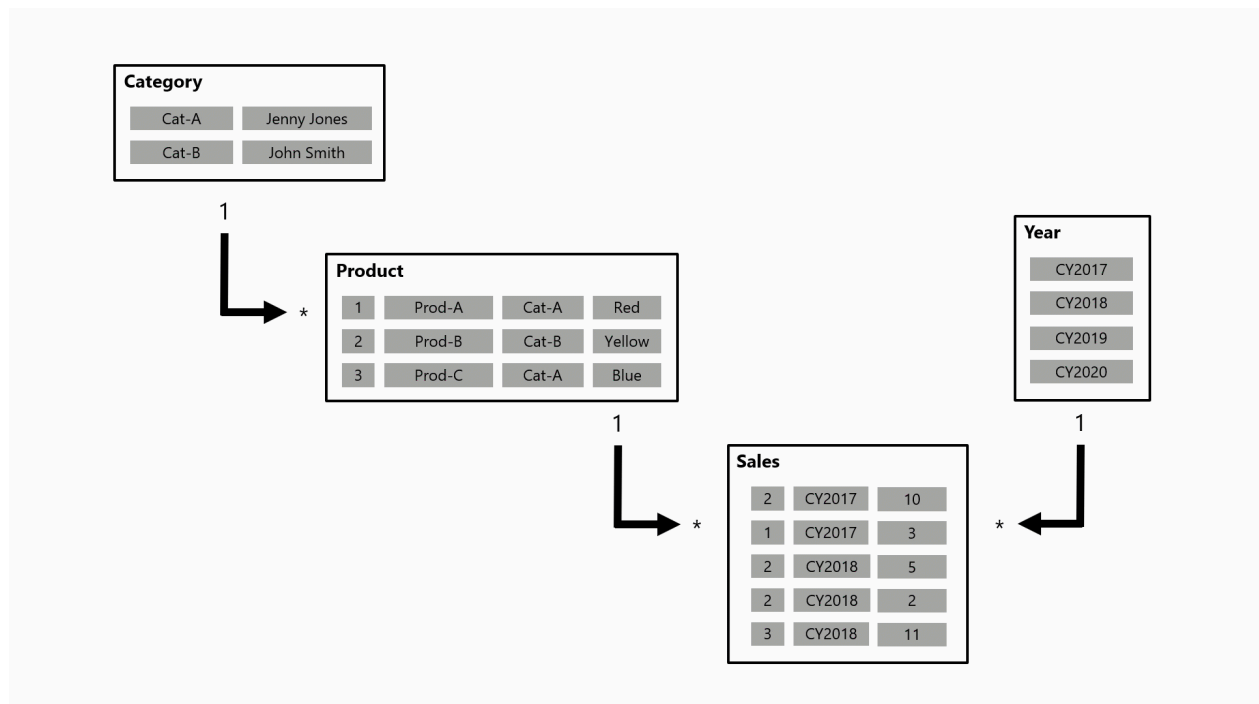
Configure relationships

Completed

When your model comprises more than one table, you need to ensure that appropriate relationships exist between the tables. Relationships propagate filters applied on one model table to a different model table. They continue to propagate so long as there's a relationship path to follow, which can involve propagation to multiple tables.

For example, when a visual filters the **Year** column of the **Date** table, a relationship to the **Sales** table automatically filters that table so that rows representing sales for that year are summarized. For report authors, this is normal and expected behavior.

Here's an animated example that shows how relationships propagate filters to other tables.



Configure data load options

Before you apply your Power Query queries to load data to your model, you should first inspect the Power BI Desktop *Data load* options and adjust them when necessary.

Specifically, you might want to enable or disable relationship settings. When enabled, these settings can import relationships detected in source data, update or delete relationships when refreshing data, and autodetect new relationships.

Relationships

- ✓ Import relationships from data sources on first load ⓘ
- ✓ Update or delete relationships when refreshing data ⓘ
- ✓ Autodetect new relationships after data is loaded ⓘ

[Learn more](#)

When relationships aren't automatically created, you can create them in the **Manage relationships** window, or by switching to Model view.

Sometimes, a table doesn't need a relationship to another model table, which is known as a *disconnected table*. A disconnected table is useful when you want to support a what-if scenario or field parameters. Both scenarios are described later in this module.

Note

For a full explanation of model relationships and links to related guidance articles, see [Model relationships in Power BI Desktop](#).

Columns

Each relationship has a single "from" column and a single "to" column. It's important that the data types for these columns are the same (or equivalent) and that they contain matching values.

You should give consideration the relationships your model needs when defining the table structures in Power Query. To support the one-side of a relationship (described next), you must ensure that column contains unique values. Further, if your data source has a multi-column key, you need to transform the data to produce single-column keys for the related tables. If the related column data types don't match, you can adjust the data types with Power Query.

Understand cardinality

Every relationship has a cardinality type, which is either:

- One-to-many (**1:***)
- Many-to-one (***:1**)
- One-to-one (**1:1**)
- Many-to-many (***:***)

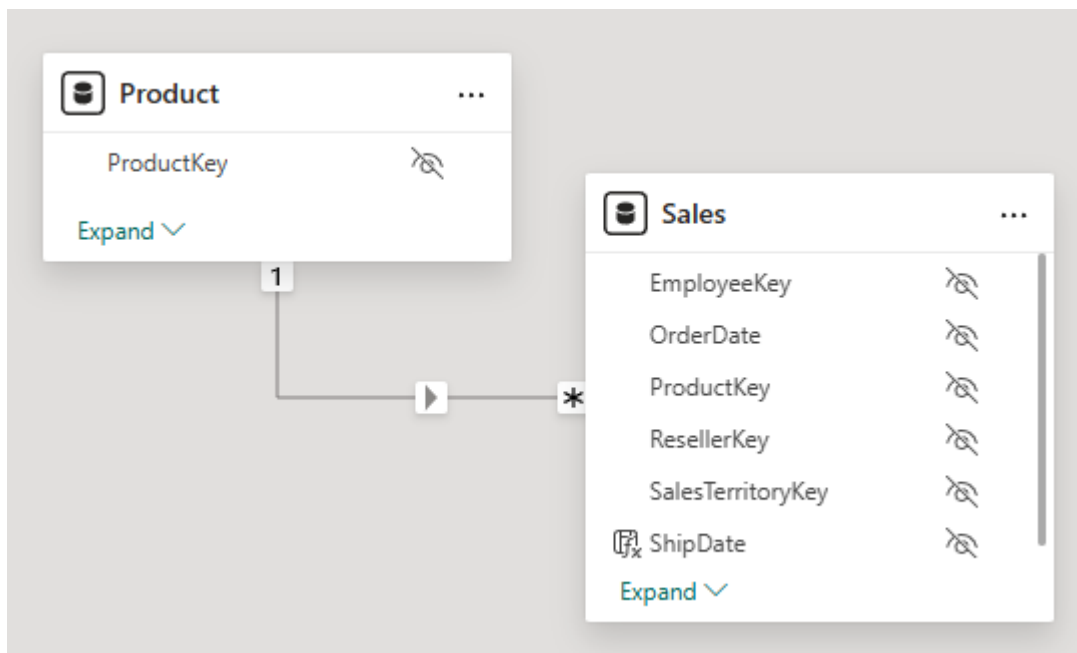
When Power BI Desktop automatically creates a relationship, it determines the cardinality based on the values already loaded into the columns. Sometimes, it might not set it correctly (because the table is yet to be loaded with rows of data), so you need to update the setting.

The *One-to-many* and *Many-to-one* relationships are essentially the same, just in different directions. They're the most commonly set cardinality types, typically supporting relationships between dimension tables (the "one" side) and fact tables (the "many" side) in a [star schema](#) design.

For example, the `ProductKey` column in the `Product` table has a one-to-many relationship with the `ProductKey` column in the `Sales` table.

Note

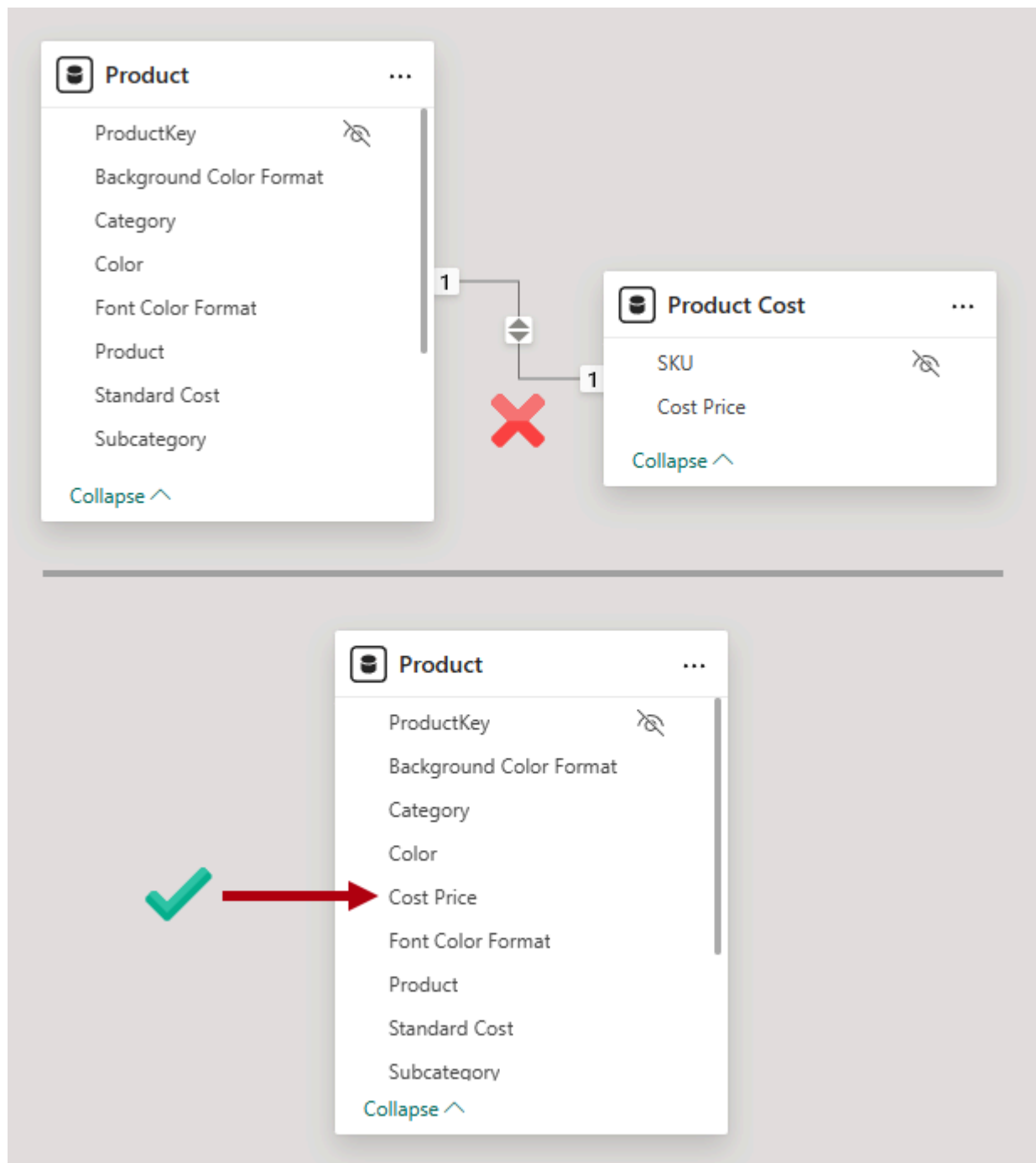
The screenshots of model diagrams in this unit only show columns that are used in relationships.



One-to-one cardinality allows you to relate two tables that each have a unique column. This type of relationship isn't common because it's considered a better practice to use Power Query to merge queries to produce a single model table. That way, you reduce the number of model tables and produce a more intuitive experience for report authors, who can find related fields in a single table.

Consider a scenario where there's a `Product Cost` table that contains the `Cost Price` column, which is sourced from a supplementary data store. Its `SKU` column contains the product stock-keeping unit (SKU). When a relationship is created to the `Product` table on the `ProductKey` column, a one-to-one relationship is established because both columns contain unique values.

In a second scenario, the modeler takes a different approach. They merge the Power Query queries together and add the `Cost Price` column to the `Product` query. They then disable the load of the `Cost Price` query. It results in only one product table.



Many-to-many cardinality allows you to configure complex relationships between model tables. It's useful when there isn't a column of unique values. For example, the **Target** table stores facts at product category level, yet the **Product** table stores products at product SKU level. While the **ProductKey** column stores the SKU, the **Category** column stores the category name, of which there are many duplicates. When you relate the **Category** column of the **Product** table to the **Category** column of the **Target** table, the cardinality must be set as *Many-to-many*. In this instance, it allows relating a dimension table to a fact table at a higher level or granularity.

Understand cross filter direction

A model relationship is defined with a cross filter direction. The direction determines how filters propagate. Where there's a "one" side, filters always propagate to the other side. Where there's a "many" side, it's possible to allow propagation to the other side too.

The possible cross filter options are dependent on the relationship cardinality type:

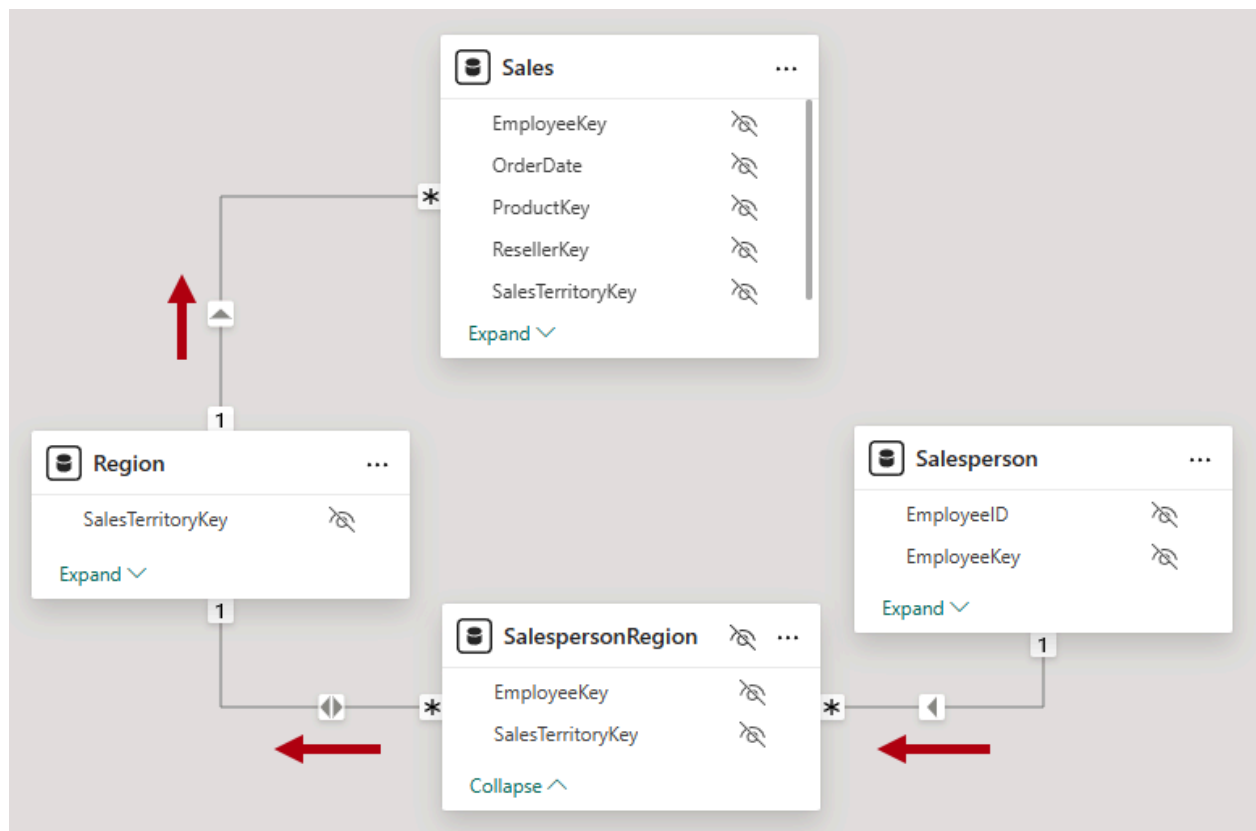
- **One-to-many (or Many-to-one):** Single or both
- **One-to-one:** Both
- **Many-to-many:** Single to either table, or both

Generally, it's a good practice to avoid or minimize cross filters in both directions. That's because it results in better query performance and usually produces an intuitive experience for report consumers.

A valid reason to allow cross filtering of a one-to-many relationship in both directions is to enable many-to-many analysis between two dimension tables. Consider an example where salespeople are assigned to multiple regions, and conversely regions can have multiple salespeople assigned. To implement this design, your model needs a bridging table that associates salespeople and regions.

The following screenshot shows a model diagram that relates the `Salesperson` table to `Sales` table. The `SalespersonRegion` table, which is a bridging table, contains the `EmployeeKey` and `SalesTerritoryKey` columns, of which neither contains unique values. Notice how the filter propagation works:

1. Filters applied to the `Salesperson` table propagate to the `SalespersonRegion` table.
2. Filters applied to the `SalespersonRegion` table propagate to the `Region` table *because cross filtering is supported in both directions*.
3. Filters applied to the `Region` table propagate to the `Sales` table.



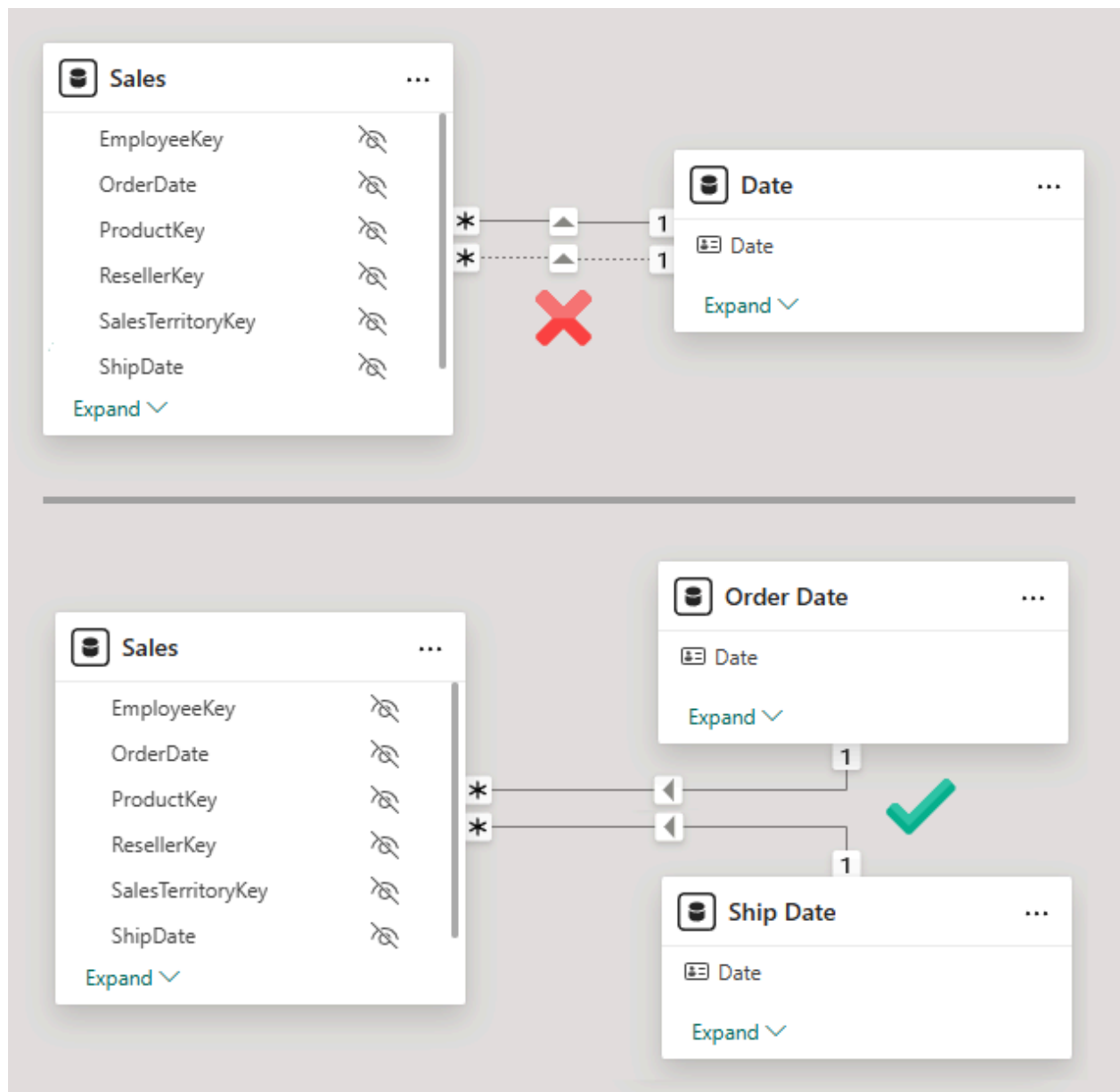
Active vs. inactive relationships

There can only be one active filter propagation path between two model tables. However, it's possible to introduce other relationship paths, though you must set these relationships as *inactive*. Inactive relationships can only be made active during the evaluation of a model calculation by using the `USERELATIONSHIP` DAX function.

It's common enough that multiple relationships to a dimension table exist, which is known as a *role-playing dimension*. For example, consider that the `Sales` table has two date columns: `OrderDate` and `ShipDate`. However, if you relate both columns to the `Date` table, only one relationship can be active.

To allow filtering by either order date, ship date, or both at the same time, you need two date tables. By creating `Order Date` and `Ship Date` tables, each can have active relationships to the `Sales` table.

The following screenshot compares two model design scenarios. The first scenario shows a single date table with an active and inactive relationship. The second scenario shows two dates tables, each with active relationships. (Active relationships are represented by solid lines; inactive relationships are represented by dotted lines.)



Work with the model diagram

In the model diagram, relationships are represented by the lines that connect tables.

An easy way to create a relationship is to drag and drop columns between tables in the model diagram. An easy way to edit a relationship is to simply double-click it.

You can interpret the properties of a relationship by viewing it in the model diagram:

- The cardinality of a relationship is described by the "one" (1) or "many" (*) icons located at the ends of the relationship line.
- The cross filter direction of a relationship is described by the arrows located in the middle of the relationship line.
- An active relationship is a solid line; an inactive relationship is a dotted line.

To determine which columns are related, you can hover the cursor over the relationship to highlight the related columns.

3. Configure tables

<https://learn.microsoft.com/en-us/training/modules/configure-semantic-model-power-bi/3-tables>

Configure tables

Completed

You're ready to enhance its design when your model contains the required tables and columns and the relationships are configured. You enhance the model by setting properties of model objects, like tables and columns. You can also create new model objects, like hierarchies, measures, and parameters.

Configure table properties

Power BI Desktop provides you with choice when creating model objects and setting their properties. You can use the ribbon, the **Data** pane, or the model diagram and its associated **Properties** pane. Typically, it's easier to work in Model view and use the **Properties** pane because it supports multi-select and bulk property updates.

Every model table has a **Name** property, which allows you to rename it. Its name is always inherited from the Power Query query name. So, if you rename a table in Power BI Desktop, the Power Query name is updated. Model table names must be unique in the model, and you should strive to set user-friendly names.

A model table also has a **Description** property, which is optional. It allows you to set a detailed definition of the table that appears when report authors hover their cursor over the table in the **Data** pane.

The **Synonyms** property allows you to set one or more alternative names for the table, so that Q&A or Copilot can better understand and resolve requests made to automatically generate visuals or DAX formulas.

A model table can be hidden, in which case it's not listed in the **Data** pane (unless the report author expressly wants hidden objects to be shown). You should hide model objects that shouldn't be used directly by report authors. For example, you should hide a bridging table used to support a many-to-many relationship between dimension tables, like the `SalespersonRegion` table described in the previous unit.

Mark date tables

By default, Power BI includes an **Auto date/time** feature to support time intelligence. When enabled, it creates hidden date tables for each column that has a **date** or **date/time** data type.

Time intelligence

☐ Auto date/time ⓘ [Learn more](#)

While this option might be useful for novice data modelers or experimentation, it's better to disable it. You can use an existing source data table or create a calculated table with DAX. Each date table in your model should contain a column of dates and be *marked* as a date table. That way, DAX time intelligence functions return appropriate results.

Marking a date table involves selecting the column that contains date values.

Mark as a date table

To enable the creation of date-related visuals, tables and quick measures using this table's

Keep in mind any built-in date tables that are already associated with this table will be ren referring to them may break. [Learn more](#)

Mark as a date table

☒ On

Choose a date column

Date

✔ Validated successfully

Power BI Desktop performs validations to ensure that the date column you select contain unique values, no BLANKs, and comprises contiguous values from beginning to end.

4. Configure columns

<https://learn.microsoft.com/en-us/training/modules/configure-semantic-model-power-bi/4-columns>

Configure columns

Completed

There are many column properties that you can set. Like with tables, you can set the name, description, synonyms, and "is hidden" properties. It's common to hide columns that are used by relationships, especially when they're based on aren't meaningful key values.

Column names must be unique within the model table, and if the column is visible, you should set a user-friendly name. If you change the column name in Power BI Desktop, a new step is appended to the Power Query query to modify the column name there.

You can assign columns to a display folder, which helps organize the fields for a table. Consider using display folders when your table comprises many visible fields.

Configure column formatting

Every column has a **Data type** property, which is inherited from the Power Query query. It's concerned with how values are *stored*. If you change the data type in Power BI Desktop, a new step is appended to the query to modify the data type there.

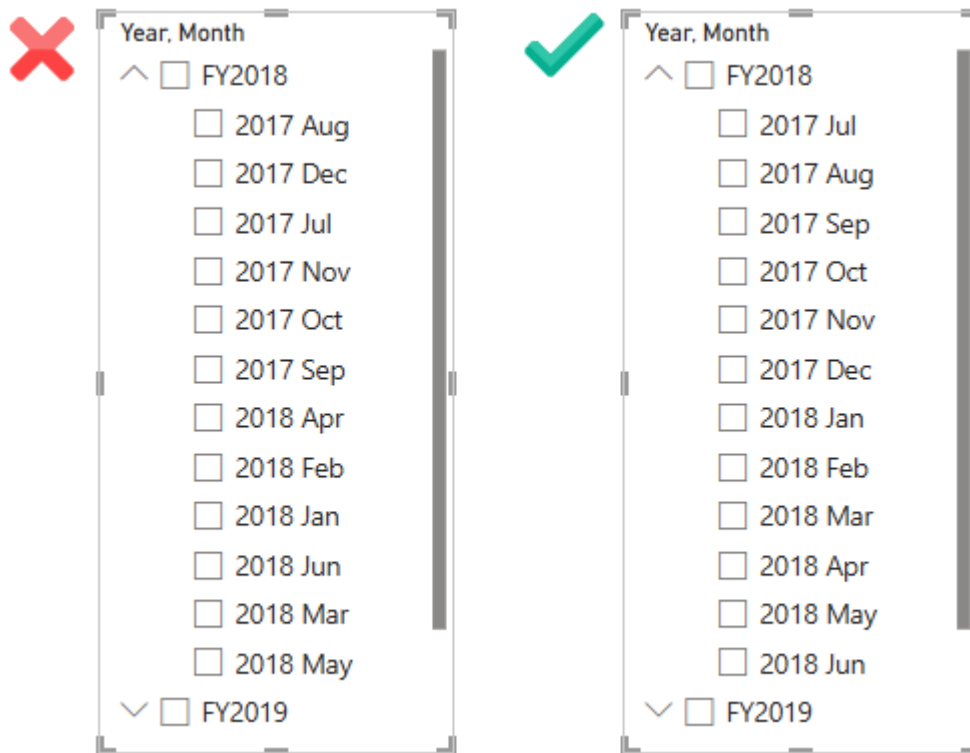
Related, though different, is the **Format** property. This property is concerned with how the values are *presented* in visuals. For example, a column with a **fixed decimal number** data type might be formatted as a currency.

Set sort order

Sometimes, the natural order of the column values doesn't meet your requirements. Text values naturally sort alphabetically, while numbers and dates naturally sort from smallest (or earlier) to largest (or later). When you need the values to sort differently, you can refer to a column in the same table that contains values suited to sorting the column.

For example, in the `Date` table, the `Month` column contains the English name of the months, like *2017 Aug* (for August 2017). The natural sort order results in months listed alphabetically, not chronologically (in which case August is ordered before all other months). In this case, you can add the `MonthKey` column to the table, which stores a numeric value comprising the year and month number.

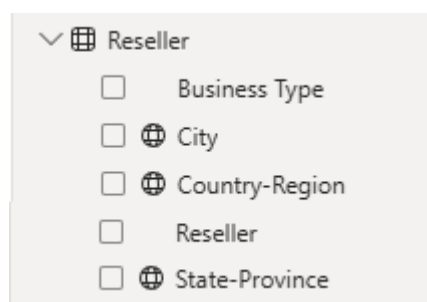
You can then set the **Sort by Column** property of the `Month` column to use the `MonthKey` column. You should then hide the `MonthKey` column because it shouldn't be used directly by report authors.



Categorize data

You can set the **Data category** property to describe the content of a column.

Setting this property is especially useful when a column stores a spatial value, either in text or as a decimal latitude or longitude value. Columns that are categorized with a spatial value appear in the **Data** pane with a spatial icon, and Power BI can more accurately geocode and visualize the data when used in map visuals.



You can also categorize columns that contain a URL.

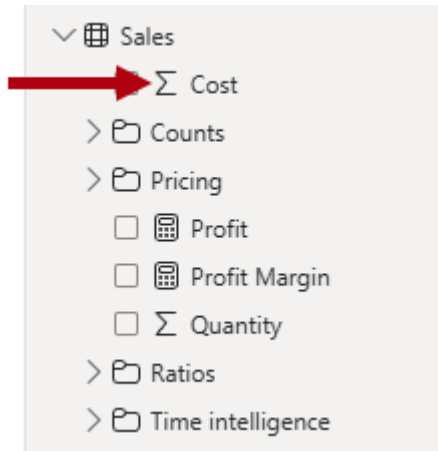
When the URL contains a general web link, set the **Data category** property to *Web URL*. When used in a table or matrix visual, the report author can configure the format options to condense the URL into a compact and easily recognizable link icon.

When the URL contains a link to an image, set the property to *Image URL*. When used in a table, matrix, slicer, or multi-row card visual, the report author can configure the format options to display the images.

Set summarization

You can set the **Summarize by** property for numeric columns to determine how they summarize (or not) *by default*. Options include Sum, Min, Max, Average, Count, and Discount Count (or None).

Summarizable columns appear in the **Data** pane with a sigma symbol (Σ), and report authors can determine how they're summarized when used by a visual.



Tip

When you want to control how report authors (or Q&A) summarize a numeric column, create a measure (described in a later unit) and hide the column.

5. Configure hierarchies

<https://learn.microsoft.com/en-us/training/modules/configure-semantic-model-power-bi/5-hierarchies>

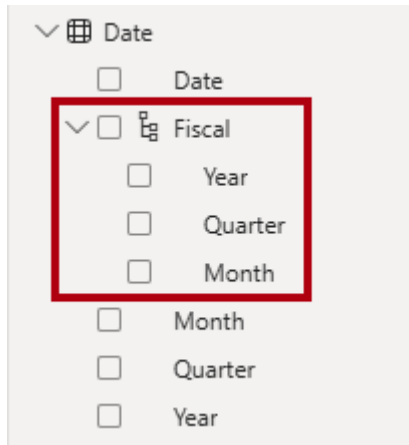
Configure hierarchies

Completed

Hierarchies are optional. By creating them, you provide clues to report authors about relationships *between* columns in a table. If you don't create the hierarchy, report authors can still achieve the same result by adding multiple columns to a visual, but it involves more knowledge and effort.

You can add one or more hierarchies to any model table. The hierarchy levels must be based on columns from that table, and they provide a navigation path across those columns.

For example, you can add a hierarchy named **Fiscal** to the **Date** table with levels based on the **Year**, **Quarter**, and **Month** columns.



The following screenshot shows a matrix visual that has the **Fiscal** hierarchy on its rows.

Year	Sales
FY2018	\$16,429,043
FY2018 Q1	\$3,195,733
2017 Jul	\$489,328
2017 Aug	\$1,540,072
2017 Sep	\$1,166,332
FY2018 Q2	\$4,874,023
FY2018 Q3	\$4,069,304
FY2018 Q4	\$4,289,983
FY2019	\$27,979,780
FY2020	\$33,139,748
Total	\$77,548,570

Like model columns, you can set description, synonym, display folder, and "is hidden" properties for hierarchies.

6. Configure measures

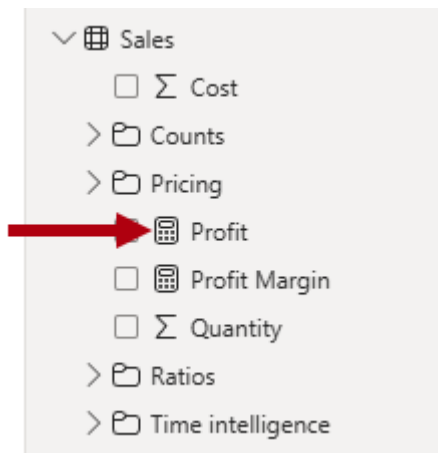
<https://learn.microsoft.com/en-us/training/modules/configure-semantic-model-power-bi/6-measures>

Configure measures

Completed

While report authors can summarize any column in Power BI visuals, they can't achieve complex summarization requirements, like a year-to-date (YTD) calculation. In this case, you need to create a measure.

A measure is a named DAX formula that's added to a model table. It achieves summarization, and it appears in the **Data** pane with a calculator icon. Measure names must be unique within the model. Measures can be simple, such as `SUM` or `AVERAGE`. Measures can also be complex and calculate the *Sales Amount* minus *Product Cost* to find the **Profit**.



These calculations extend the data in your semantic model by providing the essential calculations required for effective data visualization, enabling you to uncover deeper insights and trends within your data.

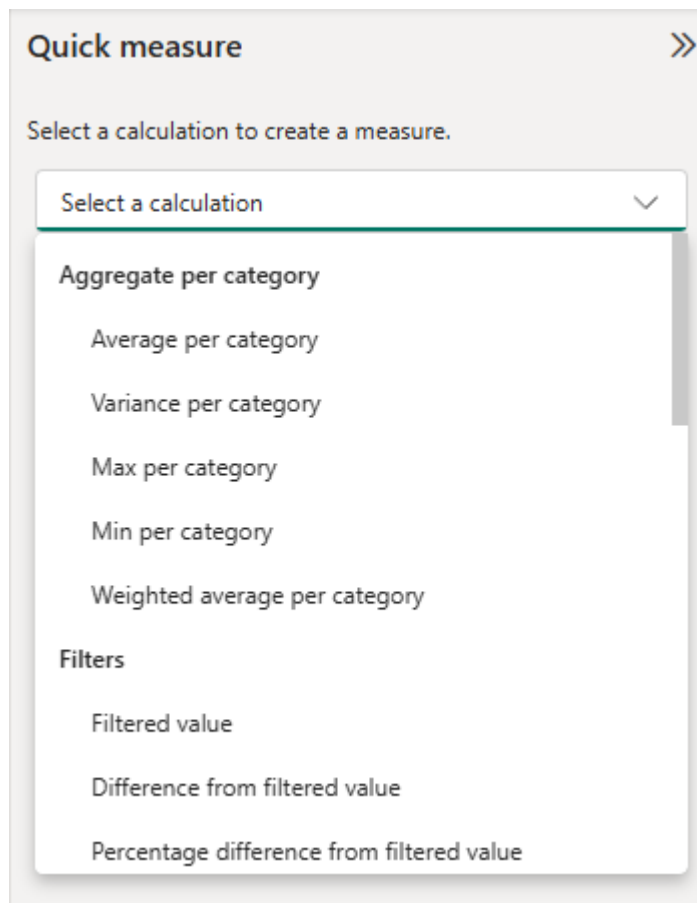
Note

If the model is queried by using Multi-Dimensional Expressions (MDX), which is the case when you use [Analyze in Excel](#), then you must also create measures.

If you don't have DAX skills, you can use Copilot for Power BI to generate a measure based on a prompt. You can also use the *Quick measures* feature.

Use Quick measures

Quick measures allow you to create a measure by selecting a calculation template and dragging fields to configure it. Power BI Desktop then creates a measure based on your configuration.



Like model columns, you can set description, synonym, display folder, and "is hidden" properties for measures. You can also set formatting properties. The **Home table** property allows assigning the measure to any model table.

Tip

You can also use Copilot to autogenerate a description value for the measure, which it does by inspecting the DAX formula. For more information, see the [Copilot for Power BI documentation](#).

7. Configure parameters

<https://learn.microsoft.com/en-us/training/modules/configure-semantic-model-power-bi/7-parameters>

Configure parameters

Completed

A parameter in Power BI lets users change report settings, like filters or calculations, without changing the original data. You can add two types of parameters to your model: *Numeric range* and *Fields*.

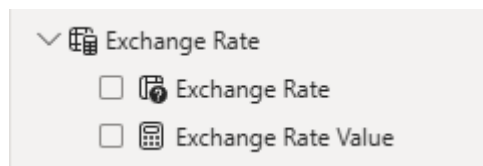
Create a Numeric range parameter

You define a *Numeric range* parameter by setting a numeric data type, minimum and maximum values, an increment value, and a default value. Power BI Desktop then creates a calculated table by using DAX. It also creates a measure that represents the value that's used to filter the table. The table doesn't have (or need) a relationship to any other model table, creating a *disconnected table*.

A numeric range parameter can support a what-if scenario, where the report consumer uses a slicer to filter the table, and measures do something relevant with the selected value. For example, a report could allow the report consumer to set a hypothetical currency exchange rate. A report visual could then display a measure that applies the exchange rate in a meaningful way.

Parameters appear in the **Data** pane with a question mark symbol (?).

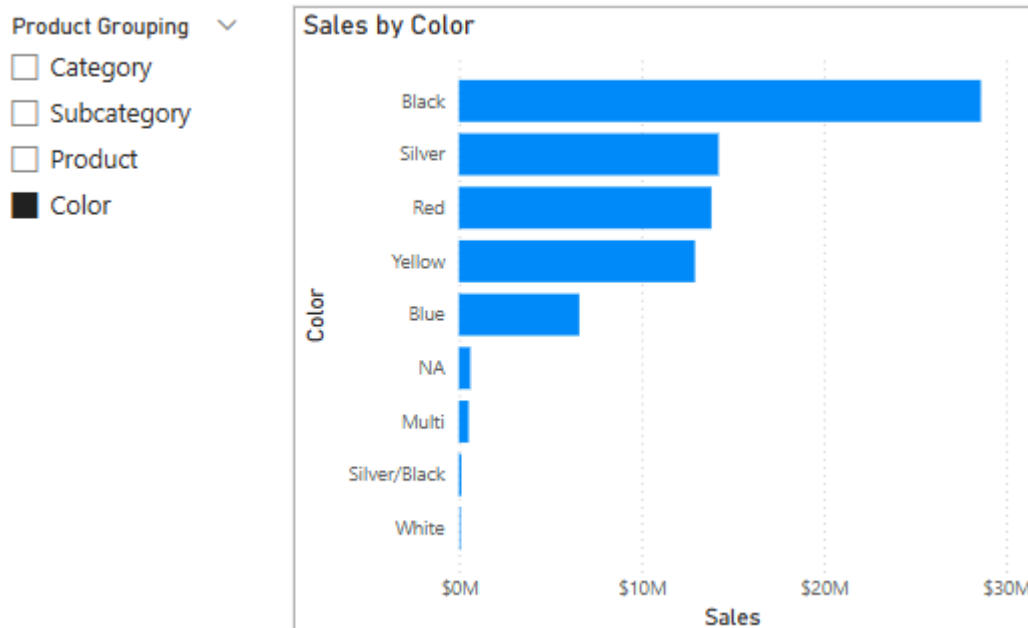
In the following screenshot, the **Exchange Rate** table is the calculated table, and the **Exchange Rate** column contains the numeric range values. The **Exchange Rate Value** is a measure that returns the select value in the range.



Create a Fields parameter

You can define a *Fields* parameter by creating a group of different fields. The parameter field can then be used in visuals and also to set up a slicer. Slicer selection allows report consumers to choose which field to visualize.

Consider an example where a fields parameter named **Product Grouping** comprises four columns from the **Product** table: **Category**, **Subcategory**, **Product**, and **Color**. When the parameter is added to the report page, the report consumer can select which field. The corresponding bar chart places the selected field on its Y-axis.



8. Exercise - Configure a semantic model in Power BI Desktop

<https://learn.microsoft.com/en-us/training/modules/configure-semantic-model-power-bi/8-lab>

Exercise - Configure a semantic model in Power BI Desktop

Completed

In this exercise, you learn how to:

- Create model relationships.
- Configure table and column properties.
- Create hierarchies.

This lab takes approximately **45** minutes to complete.

Note

A virtual machine containing the client tools you need is provided, along with the exercise instructions. Use the "Launch lab" button to launch the virtual machine.

A limited number of concurrent sessions are available. If the hosted environment is unavailable, please try again later.

Alternatively, you can [open the instructions in a separate window](#).

Access your environment

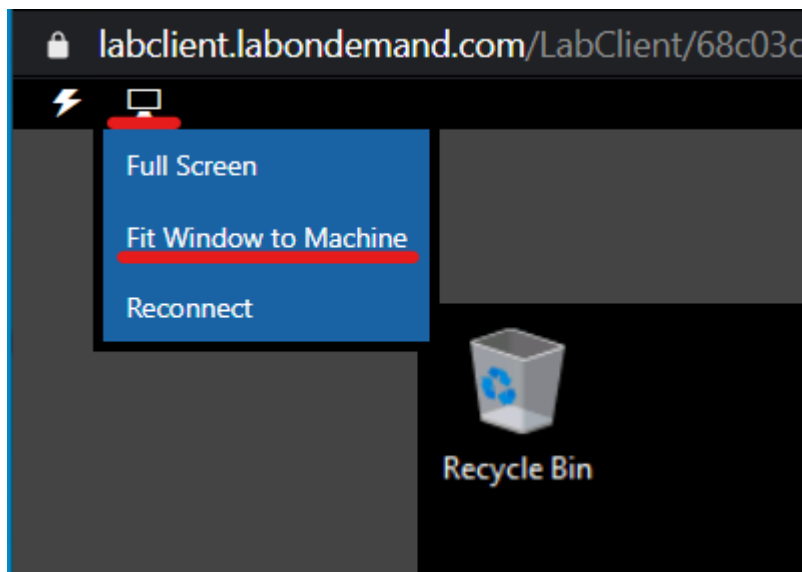
Before you start this lab (unless you are continuing from a previous lab), select **Launch lab** above.

You are automatically logged in to your lab environment as data-ai\student.

You can now begin your work on this lab.

Tip

To dock the lab environment so that it fills the window, select the PC icon at the top and then select **Fit Window to Machine**.



9. Check your knowledge

<https://learn.microsoft.com/en-us/training/modules/configure-semantic-model-power-bi/9-check>

Check your knowledge

Completed

10. Summary

<https://learn.microsoft.com/en-us/training/modules/configure-semantic-model-power-bi/10-summary>

Summary

Completed

In this module, you learned how to develop a semantic layer over your data.

First, you learned about the importance of model relationships and their role to automatically propagate filters to other model tables.

Next, you learned how to create and configure model objects, including tables, columns, hierarchies, measures, and parameters. The end result is a model that delivers your reporting requirements, and that's user-friendly and intuitive.

Write DAX formulas for semantic models

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-write-formulas/1-introduction>

Introduction

Completed

DAX (Data Analysis Expressions) lets you create powerful calculations in Power BI, helping you analyze and visualize your data in new ways.

For example, imagine you need to quickly find year-over-year sales growth or count unique customers in your reports. DAX makes these tasks simple and efficient, even when your data is complex or contains blanks.

In this module, you learn how to:

- Write DAX formulas for calculated tables, columns, and measures.
- Work with different data types and handle blank values.
- Use various DAX functions, including those similar to Excel.
- Apply operators and understand how they work together.
- Improve your formulas with variables for better performance and readability.

After finishing this module, you'll know how to build and optimize DAX calculations in Power BI, making your reports more dynamic and insightful.

2. Understand DAX calculation types

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-write-formulas/1a-calculation-types>

Understand DAX calculation types

Completed

By using Data Analysis Expressions (DAX), you can add three types of calculations to your semantic model:

- Calculated tables
- Calculated columns
- Measures

Note

DAX can also be used to define row-level security (RLS) rules, which are expressions that enforce filters over model tables. However, rules aren't considered to be model calculations so they're out of scope for this module. For more information, see [Row-level security \(RLS\) with Power BI](#).

Calculated tables

You can write a DAX formula to add a calculated table to your model. The formula can duplicate or transform *existing model data*, or create a series of data, to produce a new table. Calculated table data is always imported into your model, so it increases the model storage size and extends data refresh time.

Note

A calculated table can't connect to external data; you need to use Power Query to accomplish that task.

Calculated tables can be useful in various scenarios:

- Date tables
- Role-playing dimensions
- What-if analysis

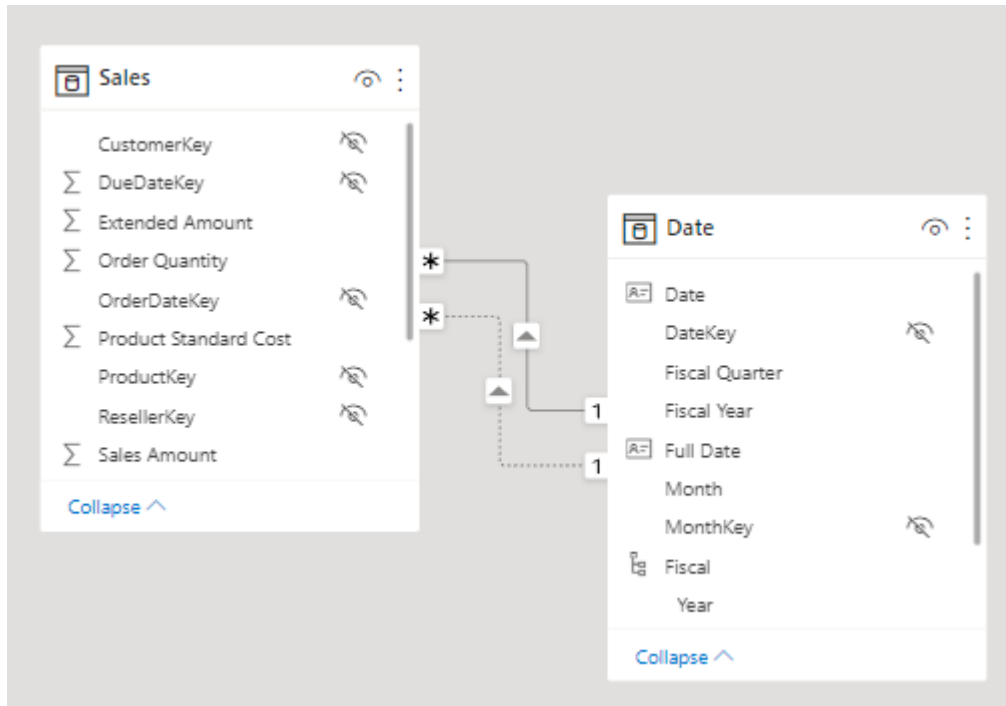
Date tables

Date tables are required to apply special time filters known as *time intelligence*. DAX time intelligence functions only work correctly when a date table is set up. When your source data doesn't include a date table, you can create one as a calculated table by using the [CALENDAR](#) or [CALENDARAUTO](#) functions.

Role-playing dimensions

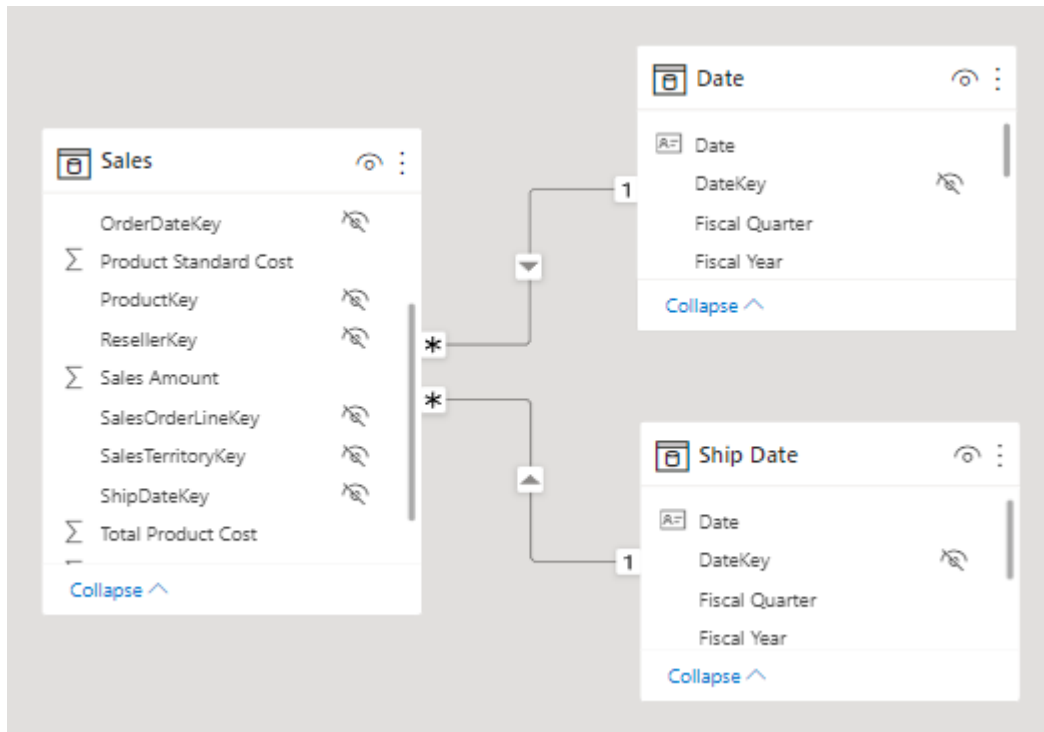
When two model tables have multiple relationships, it could be because your model has a role-playing dimension. For example, if you have a table named `Sales` that includes two date columns, `OrderDateKey` and `ShipDateKey`, both columns are related to the `Date` column in the `Date`

table. In this case, the `Date` table is described as a role-playing dimension because it could play the role of *order date* or *ship date*.



Semantic models only allow one active relationship between tables, which in the model diagram is indicated as a solid line. The active relationship is used by default to propagate filters, which in this case would be from the `Date` table to the `OrderDateKey` column in the `Sales` table. Any remaining relationships between the two tables are inactive. In a model diagram, the relationships are represented as dashed lines. Inactive relationships are only used when they're expressly requested in a calculated formula by using the `USERELATIONSHIP` function.

Perhaps a better model design could have two date tables, each with an active relationship to the `Sales` table. Thus, report users can filter by order date or ship date, or both at the same time. A calculated table can duplicate the `Date` table data to create the `Ship Date` table.



What-if analysis

Power BI Desktop includes a feature called [parameters](#). When you create a numeric range parameter, a calculated table is automatically added to your model.

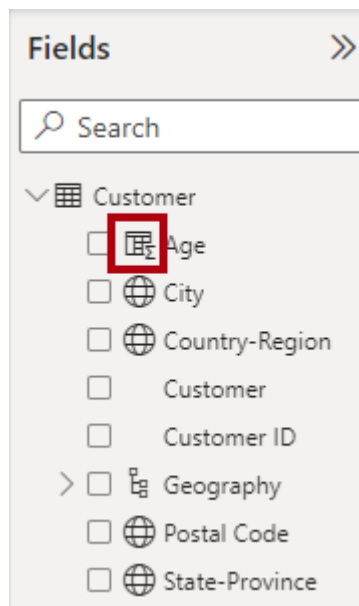
Numeric range parameters allow report users to select or filter by values that are stored in the calculated table. Measure formulas can use selected values in a meaningful way. For example, a numeric range parameter could allow the report user to select a hypothetical currency exchange rate, and a measure could divide revenue values (in a local currency) by the selected rate.

Notably, parameter calculated tables aren't related to other model tables because they're not used to propagate filters. For this reason, they're called *disconnected tables*.

Calculated columns

You can write a DAX formula to add a calculated column to any table in your model. The formula is evaluated for each table row and it returns a single value. When added to an Import storage mode table, the formula is evaluated when the semantic model is refreshed, and it increases the storage size of your model. When added to a DirectQuery storage mode table, the formula is evaluated by the underlying source database when the table is queried.

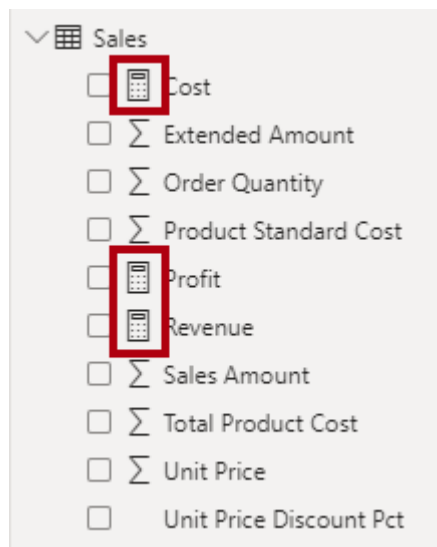
In the **Data** pane, calculated columns are enhanced with a special icon. The following example shows a single calculated column in the **Customer** table called **Age**.



Measures

You can write a DAX formula to add a measure to any table in your model. The formula achieves summarization over model data. Similar to a calculated column, the formula must return a single value. However, unlike calculated columns, which are evaluated at data refresh time, measures are evaluated at query time. Their results are never stored in the model.

In the **Data** pane, measures are shown with the calculator icon. The following example shows three measures in the **Sales** table: **Cost**, **Profit**, and **Revenue**.



Occasionally, measures can be described as *explicit measures*. To be clear, explicit measures are model calculations that are written in DAX and are commonly referred to as simply *measures*. Yet, the concept of *implicit measures* exists, too. Implicit measures are columns that can be summarized by visuals in simplistic ways, like count, sum, minimum, maximum, and so on. You can identify implicit measures in the **Data** pane because they're shown with the sigma symbol (Σ).

Note

Any column can be summarized when added to a visual. Therefore, whether they're shown with the sigma symbol or not, when they're added to a visual, they can be set up as implicit measures.

Additionally, no such concept as a *calculated measure* exists in tabular modeling. The word *calculated* is used to describe calculated tables and calculated columns, which distinguish them from tables and columns that originate from Power Query. Power Query doesn't have the concept of an explicit measure.

3. Write DAX formulas

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-write-formulas/2-formulas>

Write DAX formulas

Completed

Each model calculation type, calculated table, calculated column, or measure is defined by its name, followed by the equals symbol (=), which is then followed by a DAX formula. Use the following template to create a model calculation:

```
<Calculation name> = <DAX formula>
```

For example, the definition of the `Ship Date` calculated table that duplicates the `Date` table data is:

```
Ship Date = 'Date'
```

A DAX formula consists of expressions that return a result. The result is either a table object or a scalar value. Calculated table formulas must return a table object; calculated column and measure formulas must return a scalar value (single value).

Formulas are assembled by using:

- DAX functions
- DAX operators
- References to model objects
- Constant values, like the number 24 or the literal text "FY" (abbreviation for fiscal year)
- DAX variables

- Whitespace

Tip

When entering DAX formulas in Power BI Desktop, you have the benefit of *IntelliSense*. IntelliSense is a code-completion aid that lists functions and model resources. When you select a DAX function, it also provides you with a definition and description. We recommend that you use IntelliSense to help you quickly build accurate formulas.

DAX functions

Similar to Microsoft Excel, DAX is a functional language meaning that formulas rely on functions to accomplish specific goals. Typically, DAX functions have arguments that allow passing in variables. Formulas can use many function calls and will often nest functions within other functions.

In a formula, function names must be followed by parentheses. Within the parentheses, variables are passed in.

Note

Some functions don't take arguments, or arguments might be optional.

Working with DAX functions is described later in this module.

DAX operators

Formulas also rely on operators, which can perform arithmetic calculations, compare values, work with strings, or test conditions.

DAX operators are described in more detail later in this module.

References to model objects

Formulas can only refer to three types of model objects: tables, columns, or measures. A formula can't refer to a hierarchy or a hierarchy level. (Recall that a hierarchy level is based on a column, so your formula can refer to a hierarchy level's column.)

Table references

When you reference a table in a formula, officially, the table name is enclosed within single quotation marks. In the following calculated table definition, notice that the `Date` table is enclosed within single quotation marks.

```
Ship Date = 'Date'
```

However, single quotation marks can be omitted when both of the following conditions are true:

1. The table name doesn't include embedded spaces.
2. The table name isn't a reserved word that's used by DAX. All DAX function names and operators are reserved words. *Date* is a DAX function name, which explains why, when you're referencing a table named `Date`, that you must enclose it within single quotation marks.

In the following calculated table definition, it's possible to omit the single quotation marks when referencing the `Airport` table:

```
Arrival Airport = Airport
```

Column references

When you reference a column in a formula, the column name must be enclosed within square brackets. Optionally, it can be preceded by its table name. For example, the following measure definition refers to the `Sales Amount` column.

```
Revenue = SUM([Sales Amount])
```

Because column names are unique within a table but not necessarily unique within the model, you can disambiguate the column reference by preceding it with its table name. This disambiguated column is known as a *fully qualified column*. Some DAX functions require passing in fully qualified columns.

Tip

To improve the readability of your formulas, we recommend that you always precede a column reference with its table name.

The previous example measure definition can be rewritten as:

```
Revenue = SUM(Sales[Sales Amount])
```

Measure references

When you reference a measure in a formula, like column name references, the measure name must be enclosed within square brackets. For example, the following measure definition refers to the `Revenue` and `Cost` measures.

```
Profit = [Revenue] - [Cost]
```

If you're a DAX beginner, the fact that column and measure references are always enclosed within square brackets can cause confusion when you're trying to understand a formula. However, as you become proficient with DAX fundamentals, you're able to determine which type of object it is because, in DAX formulas, columns, and measures are used in different ways.

Tip

It's possible to precede a measure reference with its table name. However, measures are a model-level object. While they're assigned to a home table, it's only a cosmetic relationship to logically organize measures in the **Data** pane.

Therefore, while we recommend that you always precede a column reference with its table name, the inverse is true for measures: We recommend that you never precede a measure reference with its table name.

For more information, see [Column and measure references](#).

DAX variables

Formulas can declare DAX variables to store results.

How and when to use DAX variables is described later in this module.

Whitespace

Whitespace refers to characters that you can use to format your formulas in a way that's quick and simple to understand. Whitespace characters include:

- Spaces
- Tabs
- Carriage returns

Whitespace is optional and it doesn't modify your formula logic or negatively affect performance. We strongly recommend that you adopt a format style and apply it consistently, and consider the following recommendations:

- Use spaces between operators.
- Use tabs to indent nested function calls.
- Use carriage returns to separate function arguments, especially when it's too long to fit on a single line. Formatting in this way makes it simpler to troubleshoot, especially when the formula is missing a parenthesis.
- Err on the side of too much whitespace than too little.

Tip

In the formula bar, to enter a carriage return, press **Shift+Enter**. Pressing **Enter** alone commits your formula.

Notice how the following measure definition is written in a single line and that includes five DAX function calls:

```
Revenue YoY % = DIVIDE([Revenue] - CALCULATE([Revenue], SAMEPERIODLASTYEAR('Date'[Date])),
```

The following example is the same measure definition but now formatted, which helps make it easier to read and understand:

```
Revenue YoY % =  
DIVIDE(  
    [Revenue]  
    - CALCULATE(  
        [Revenue],  
        SAMEPERIODLASTYEAR('Date'[Date])  
    ),  
    CALCULATE(  
        [Revenue],  
        SAMEPERIODLASTYEAR('Date'[Date])  
    )  
)
```

Try formatting the measure on your own. Open the [Adventure Works DW 2020 M02.pbix](#) Power BI Desktop file and then, in the **Data** pane, expand the **Sales** table and then select the **Revenue YoY %** measure. In the formula bar, use tab and carriage return characters to produce the same result as the previous example. When you add a carriage return, remember to press **Shift+Enter**.

This measure definition can be further improved for readability and performance, which is explained later in this module.

Tip

An excellent formatting tool from another source that can help you format your calculations is [DAX Formatter](#). This tool allows you to paste in your calculation and format it. You can then copy the formatted calculation to the clipboard and paste it back into Power BI Desktop.

4. DAX data types

DAX data types

Completed

Each column in a semantic model has a data type, which controls what kind of values are stored. You can set the data type in Power Query when connecting to data or creating columns. If you add a calculated column, DAX determines its data type based on the formula you write. Measures also have data types, but they're determined by the result of their calculation and can change depending on filter context.

Model data types and DAX data types are related but not always the same. The table below shows how they correspond and the range of values each supports.

Model data type	DAX data type	Description
Whole number	64-bit integer	-2^{63} through $2^{63}-1$
Decimal number	64-bit real	Negative: -1.79×10^{308} through -2.23×10^{-308} - zero (0) - positive: 2.23×10^{-308} through 1.79×10^{308} - Limited to 17 decimal digits
Boolean	Boolean	TRUE or FALSE
Text	String	Unicode character string
Date	Date/time	Valid dates are all dates after January 1, 1900
Currency	Currency	-9.22×10^{14} through 9.22×10^{14} - limited to four decimal digits of fixed precision
N/A	BLANK	In some cases, it's the equivalent of a database (SQL) NULL

BLANK data type

The BLANK data type deserves a special mention. DAX uses BLANK for both database NULL and for blank cells in Excel. BLANK doesn't mean zero. Perhaps it might be simpler to think of it as the *absence of a value*.

Two DAX functions are related to the BLANK data type: the **BLANK** function returns BLANK, while the **ISBLANK** function tests whether an expression evaluates to BLANK.

5. Work with DAX functions

Work with DAX functions

Completed

The DAX function library consists of hundreds of functions, each designed to accomplish a specific goal.

Because DAX originated with the Power Pivot add-in for Microsoft Excel 2010, over 80 functions are available that can also be found in Excel. It was a deliberate design strategy by Microsoft to ensure that Excel users can quickly become productive with DAX.

However, many functions exist in Power BI, but not Excel because they're specific to data modeling:

- Relationship navigation functions
- Filter context modification functions
- Iterator functions
- Time intelligence functions
- Path functions

Tip

To search for documentation that is related to a DAX function, in a web search, enter the keyword *DAX* followed by the function name.

For more information, see the [DAX function reference](#).

Functions that originate from Excel

The following sections consider several useful functions that you might already be familiar with because they exist in Excel.

The **IF** function tests whether a condition that's provided as the first argument is met. It returns one value if the condition is **TRUE** and returns the other value if the condition is **FALSE**. The function's syntax is:

```
IF(<logical_test>, <value_if_true>[, <value_if_false>])
```

Tip

A function argument is optional when documentation shows it enclosed within square brackets.

If **logical_test** evaluates to **FALSE** and **value_if_false** isn't provided, the function returns BLANK.

Many Excel summarization functions are available, including **SUM**, **COUNT**, **AVERAGE**, **MIN**, **MAX**, and many others. The only difference is that in DAX, you pass in a column reference, whereas in Excel, you pass in a range of cells.

Many Excel mathematic, text, date and time, information, and logical functions are available as well. For example, a small sample of Excel functions that are available in DAX include **ABS**, **ROUND**, **SQRT**, **LEN**, **LEFT**, **RIGHT**, **UPPER**, **DATE**, **YEAR**, **MONTH**, **NOW**, **ISNUMBER**, **TRUE**, **FALSE**, **AND**, **OR**, **NOT**, and **IFERROR**.

Functions that don't originate from Excel

Two useful DAX functions that aren't specific to modeling and that don't originate from Excel are **DISTINCTCOUNT** and **DIVIDE**.

DISTINCTCOUNT function

You can use the **DISTINCTCOUNT** DAX function to count the number of distinct values in a column. This function is especially powerful in an analytics solution. Consider that the count of customers is different from the count of *distinct* customers. The latter doesn't count repeat customers, so the difference is "How many customers" compared with "How many *different* customers."

DIVIDE function

You can use the **DIVIDE** DAX function to achieve division. You must pass in numerator and denominator expressions. Optionally, you can pass in a value that represents an *alternate result*. The **DIVIDE** function's syntax is:

```
DIVIDE(<numerator>, <denominator>[, <alternate_result>])
```

The **DIVIDE** function automatically handles division by zero cases. If an alternate result isn't passed in, and the denominator is zero or BLANK, the function returns BLANK. When an alternate result is passed in, it's returned instead of BLANK.

This function is convenient because it saves your expression from having to first test the denominator value. The function is also better optimized for testing the denominator value than the **IF** function. The performance gain is significant because checking for division by zero is expensive. What's more, using the **DIVIDE** function results in a more concise and elegant expression.

Tip

We recommend that you use the **DIVIDE** function whenever the denominator is an expression that could return zero or BLANK. In the case that the denominator is a constant value, we recommend that you use the divide operator (/), which is introduced later in this module. In this

case, the division is guaranteed to succeed, and your expression performs better because it avoids unnecessary testing.

6. Use DAX operators

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-write-formulas/5-operators>

Use DAX operators

Completed

Your DAX formulas can use operators to create expressions that perform arithmetic calculations, compare values, work with strings, or test conditions.

Tip

Many DAX operators and precedence order are the same as those found in Excel.

Arithmetic operators

The following table lists the arithmetic operators.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation

Remember, when you're dividing two expressions, and when the denominator could return zero or BLANK, it's more efficient and safer to use the `DIVIDE` function.

Comparison operators

The following table lists the comparison operators, which are used to compare two values. The result is either TRUE or FALSE.

Operator	Description
=	Equal to
==	Strict equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to

All comparison operators, except **strict equal to** (`==`), treat BLANK as equal to the number zero, an empty string (""), the date December 30, 1899, or `FALSE`. It means that the expression `[Revenue] = 0` is `TRUE` when the value of `[Revenue]` is either zero or BLANK. In contrast, `[Revenue] == 0` is `TRUE` only when the value of `[Revenue]` is zero.

Text concatenation operator

Use the ampersand (&) character to connect, or concatenate, two text values to produce one continuous text value. For example, consider the following calculated column definition:

```
Model Color = 'Product'[Model] & "-" & 'Product'[Color]
```

Logical operators

Use logical operators to combine expressions that produce a single result. The following table lists all logical operators.

Operator	Description
&&	Creates an AND condition between two expressions where each has a Boolean result. If both expressions return TRUE, the combination of the expressions also returns TRUE; otherwise the combination returns FALSE.
(double pipe)	Creates an OR condition between two logical expressions. If either expression returns TRUE, the result is TRUE; only when both expressions are FALSE is the result FALSE.
IN	Creates a logical OR condition between each row that is being compared to a table. Note: The table constructor syntax uses braces.
NOT	Inverts the state of a Boolean expression (FALSE to TRUE, and vice versa).

An example that uses the `IN` logical operator is the `ANZ Revenue` measure definition, which uses the `CALCULATE` function to enforce a specific filter of two countries/regions: Australia and New Zealand.

Note

You're introduced to the powerful `CALCULATE` function when you learn how to modify the filter context.

```
ANZ Revenue =  
CALCULATE(  
    [Revenue],  
    Customer[Country-Region] IN {  
        "Australia",  
        "New Zealand"  
    }  
)
```

Operator precedence

When your DAX formula includes multiple operators, DAX uses rules to determine the evaluation order, which is known as an *operator precedence*. Operations are ordered according to the following table.

Operator	Description
<code>^</code>	Exponentiation
<code>-</code>	Sign (as in -1)
<code>*</code> and <code>/</code>	Multiplication and division
<code>NOT</code>	NOT
<code>+</code> and <code>-</code>	Addition and subtraction
<code>&</code>	Concatenation of two strings of text
<code>=</code> , <code>=</code> , <code><</code> , <code>></code> , <code><=</code> , <code>>=</code> , <code><></code>	Comparison

When the operators have equal precedence value, they're ordered from left to right.

In general, operator precedence is the same as what's found in Excel. If you need to override the evaluation order, then group operations within parentheses.

For example, consider the following calculated column definition:

```
Extended Amount = Sales[Order Quantity] * Sales[Unit Price] * 1 - [Unit Price Disc
```

This sample calculated column definition produces an incorrect result because multiplication happens before the subtraction. The following correct calculated column definition uses parentheses to ensure that the subtractions happen before the multiplications.

```
Extended Amount = Sales[Order Quantity] * Sales[Unit Price] * (1 - [Unit Price Disc
```

Tip

Remembering operator precedence rules can be challenging, especially for DAX beginners. We recommend that you test your formulas thoroughly. When the formulas don't produce the correct result due to an incorrect evaluation order, you can experiment by adding parentheses to adjust the evaluation order. You can also add parentheses to improve the readability of your formulas.

For more information about DAX operators and precedence order, see [DAX operators](#).

Implicit conversion

When writing a DAX formula that uses operators to combine different data types, you don't need to explicitly convert types. Usually, DAX automatically identifies the data types of referenced model objects and performs implicit conversions where necessary to complete the specified operation.

However, some limitations might exist on the values that can be successfully converted. If a value or a column has a data type that's incompatible with the current operation, DAX returns an error. For example, the attempt to multiply a date value creates an error because it isn't logical.

BLANK is handled differently, depending on the operator that is used. It's handled similar to how Excel treats BLANK, but differently to how databases (SQL) treat NULL. BLANK is treated as zero when acted on by arithmetic operators and as an empty string when concatenated to a string.

Tip

Remembering how BLANK is handled can be challenging, especially for DAX beginners. We recommend that you test your formulas thoroughly. When BLANKs create unexpected results, consider using the [IF](#) and [ISBLANK](#) functions to test for BLANK, and then respond in an appropriate way.

7. Use DAX variables

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-write-formulas/6-variables>

Use DAX variables

Completed

You can declare DAX variables in your formula expressions. When you declare at least one variable, a **RETURN** clause is used to define the expression, which then refers to the variables.

We recommend that you use variables because they offer several benefits:

- They improve the readability and maintenance of your formulas.
- They improve performance because variables are evaluated once and only when or if they're needed.
- They allow (at design time) straightforward testing of a complex formula by returning the variable of interest.

The following example shows a formula that declares a variable. The `Revenue YoY %` measure definition is rewritten to declare a variable that's assigned the value of the prior year's revenue.

```
Revenue YoY % =  
VAR RevenuePriorYear =  
    CALCULATE(  
        [Revenue],  
        SAMEPERIODLASTYEAR('Date'[Date])  
    )  
RETURN  
    DIVIDE(  
        [Revenue] - RevenuePriorYear,  
        RevenuePriorYear  
    )
```

Notice that the `RETURN` clause refers to the variable twice. This improved measure definition formula runs in at least half the time because it doesn't need to evaluate the prior year's revenue twice.

In the [Adventure Works DW 2020 M02.pbix](#) Power BI Desktop file, refactor the `Revenue YoY %` measure to produce the same result as the previous example.

For more information on using DAX variables, see [Use variables to improve your formulas](#).

8. Check your knowledge

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-write-formulas/7-check>

Check your knowledge

Completed

9. Summary

Summary

Completed

In this module, you learned how to use DAX to add calculations to your model. You explored three types of calculations:

- Calculated tables
- Calculated columns
- Measures

You also learned the basics of DAX, including:

- Writing formulas
- Using different data types
- Applying functions, operators, and variables
- Referencing model objects and constants

Create DAX calculations in semantic models

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-create-calculations/1-introduction>

Introduction

Completed

When you first create a semantic model by applying Power Query queries, the model consists of tables and columns. At this point, the model probably isn't ready for use. That's because you might need to create or adjust model relationships, create other tables or columns, add hierarchies, or set model properties. You might also identify the need for calculations to summarize the model data, especially when the requirement can't be achieved by summarizing a single column. For example, you might want to calculate year-to-date (YTD) sales revenue, which requires special time filters.

You can use Data Analysis Expressions (DAX) to add three types of calculations to your semantic model to complete your model design:

- **Calculated tables**, which can be useful to create date tables or role-playing dimensions, or to enable what-if analysis.
- **Calculated columns**, which can be added to tables in your model.
- **Measures**, which achieve summarization over model data.

In this module, you learn why and how to create each of these calculation types.

2. Create calculated tables

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-create-calculations/2-calculated-tables>

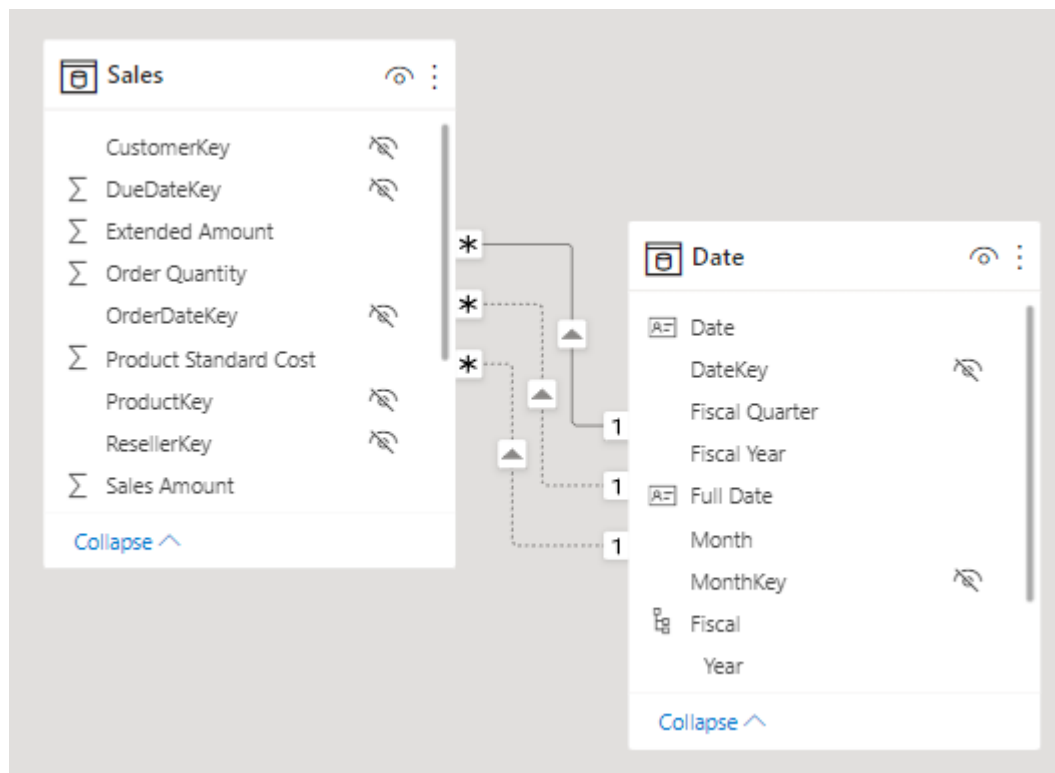
Create calculated tables

Completed

A calculated table is created with a DAX formula that returns a table object, allowing you to duplicate or transform existing model data to produce a new table.

Duplicate a table

A common design challenge in data modeling is handling multiple relationships between tables. For example, the **Sales** table might have three relationships to the **Date** table, as shown in the following diagram:



The **Sales** table stores sales data by order date, ship date, and due date, resulting in three relationships to the **Date** table. However, only one relationship can be active at a time, as indicated by a solid line in the diagram. The other relationships are inactive and shown as dashed lines.

In our example, the active relationship filters the **OrderDateKey** column in the **Sales** table, so filters applied to the **Date** table affect sales by order date only.

To enable filtering sales by ship date, a new table can be created by duplicating the **Date** table with the following formula:

```
Ship Date = 'Date'
```

This formula creates a new table named **Ship Date** with the same columns and rows as the original **Date** table. When the **Date** table is refreshed, the **Ship Date** table is also recalculated to remain synchronized. Now you can create an active relationship between **Ship Date** and **Sales** tables to allow filtering by ship dates too.

Configure duplicate tables

When you create a calculated table, you need to apply any custom configurations to the new duplicate table you want carried over. For instance, it's a good idea to rename columns so that they better describe their purpose.

In our example, the `Fiscal Year` column in the `Ship Date` table can be renamed as `Ship Fiscal Year`. The `Ship Full Date` column should be sorted by the `Ship Date` column and the `MonthKey` column can be hidden to improve sorting and reporting.

Calculated tables are useful to work in scenarios when multiple relationships between two tables exist, as previously described. However, calculated tables increase the model's storage size and can prolong data refresh times, especially when they depend on other tables.

Note

While duplicate tables solve this challenge, there are other more performant solutions. We cover this concept again when discussing measures later in this module.

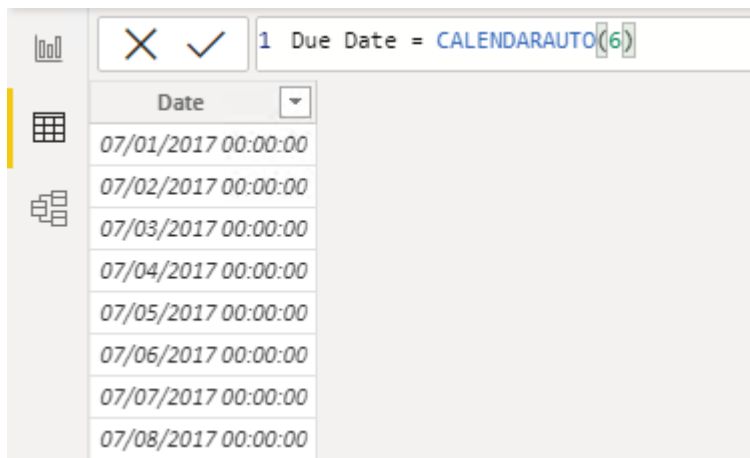
Create a date table

A good use case for calculated tables is to add a date table to your model. Date tables are required to apply special time filters known as *time intelligence*.

You can create a calculated date table using the `CALENDARAUTO` function. The `CALENDARAUTO` function scans all date and date/time columns in the model to determine the earliest and latest dates, then generates a complete set of dates that span all years in the data. The argument specifies the last month of the financial year; for example, passing `6` sets June as the year end.

```
Due Date = CALENDARAUTO(6)
```

The resulting `Due Date` table contains a single column of dates. The following image shows the `Due Date` table in data view, with dates sorted from earliest to latest:



Date
07/01/2017 00:00:00
07/02/2017 00:00:00
07/03/2017 00:00:00
07/04/2017 00:00:00
07/05/2017 00:00:00
07/06/2017 00:00:00
07/07/2017 00:00:00
07/08/2017 00:00:00

Tip

The `CALENDAR` function can also be used to create a date table by specifying a start and end date, either as static values or as expressions based on model data.

If the earliest date in the model is October 15, 2021, and the latest is June 15, 2022, the function returns dates from July 1, 2021, to June 30, 2022. This ensures the table includes complete years, which is required for marking a date table.

Mark as a date table

Once you create a date table, you need to *Mark it as a date table* in Power BI Desktop. This setting allows you to use time intelligence functions in DAX calculations. When you specify your own date table, Power BI Desktop performs the following validations of that column and its data, to ensure that the data contains:

- Unique values.
- No null values.
- Contiguous date values (from beginning to end).
- Same timestamp across each value for Date/Time data types.

This setting applies to any date tables, either imported or created in Power Query or calculated tables.

Note

You must either use a custom date table or use the built-in auto/date time feature in Power BI to use time intelligence. The auto/date time feature has limited values and can't be customized, which is one reason to consider using a custom date table.

3. Create calculated columns

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-create-calculations/3-calculated-columns>

Create calculated columns

Completed

Occasionally, you need more columns than exist in your data. Ideally, you add these columns directly to the data source, which supports easier maintenance and allows the column logic to be reused in

other models or reports. However, if you need to add the column once connected to the data in Power BI, you can either create a custom column in Power Query Editor or add a calculated column to the semantic model. No matter how you add the column, the result is the same from a report user's perspective.

In general, custom columns in Power Query are preferred because they're loaded into the model in a more compact and optimal way. Calculated columns are recommended only when adding columns to a calculated table or when the formula:

- Depends on summarized model data.
- Requires specialized modeling functions available only in DAX, such as `RELATED`, `RELATEDTABLE`, or functions for parent-child hierarchies.

Create calculated columns

A DAX formula can be used to add a *calculated column* to any table in the model. The formula must return a single value (scalar) for each row.

Calculated columns in import models increase storage size and can prolong data refresh times, especially when they depend on other tables that are refreshed. Therefore, be cautious when using too many calculated columns and consider if a measure can be used instead.

In the following examples, several calculated columns are created to support fiscal analysis to demonstrate the versatility of calculated columns.

Fiscal Year

This column determines the fiscal year for each date. The fiscal year starts in July, so dates from July to December are assigned to the next calendar year. The formula concatenates "FY" with the year, incrementing the year by one for dates in the second half of the year.

```
Due Fiscal Year =  
"FY"  
    & YEAR('Due Date'[Due Date])  
    + IF(  
        MONTH('Due Date'[Due Date]) > 6,  
        1  
    )
```

The following steps describe how Microsoft Power BI evaluates the calculated column formula:

1. The addition operator (+) is evaluated before the text concatenation operator (&).
2. The `YEAR` function returns the whole number value of the due date year.
3. The `IF` function returns the value when the due date month number is 7-12 (July to December); otherwise, it returns BLANK. (For example, because the Adventure Works financial year is July-June, the last six months of the calendar year will use the next calendar year as their financial year.)

4. The year value is added to the value that is returned by the `IF` function, which is the value one or BLANK. If the value is BLANK, it's implicitly converted to zero (0) to allow the addition to produce the fiscal year value.
5. The literal text value `"FY"` concatenated with the fiscal year value, which is implicitly converted to text.

Fiscal Quarter

This column assigns a fiscal quarter to each date, based on the fiscal year structure where Quarter 1 is July–September. The formula appends `Q` and the quarter number to the fiscal year label.

```
Due Fiscal Quarter =  
'Due Date'[Due Fiscal Year] & " Q"  
  & IF(  
    MONTH('Due Date'[Due Date]) <= 3,  
    3,  
    IF(  
      MONTH('Due Date'[Due Date]) <= 6,  
      4,  
      IF(  
        MONTH('Due Date'[Due Date]) <= 9,  
        1,  
        2  
      )  
    )  
  )  
)
```

Month Key

The `MonthKey` formula multiplies the due date year by the value 100 and then adds the month number of the due date. It produces a numeric value that can be used to sort the `Due Month` text values in chronological order.

```
MonthKey =  
(YEAR('Due Date'[Due Date]) * 100) + MONTH('Due Date'[Due Date])
```

Full Date Label

The `FORMAT` function converts the `Due Date` column value to text by using a format string. In this case, the format string produces a label that describes the year, abbreviated month name, and day:

```
Due Full Date =  
FORMAT('Due Date'[Due Date], "yyyy mmm, dd")
```

Note

Many user-defined date/time formats exist. For more information, see [Custom date and time formats for the FORMAT function](#).

After these additions, the `Due Date` table contains six columns: the original date column and five calculated columns. These columns support both time intelligence functions and readability for report consumers.

1 MonthKey =

2 (YEAR('Due Date'[Due Date]) * 100) + MONTH('Due Date'[Due Date]))

Due Date	Due Fiscal Year	Due Fiscal Quarter	Due Month	Due Full Date	MonthKey
07/01/2017 00:00:00	FY2018	FY2018 Q1	2017 Jul	2017 Jul, 01	201707
07/02/2017 00:00:00	FY2018	FY2018 Q1	2017 Jul	2017 Jul, 02	201707
07/03/2017 00:00:00	FY2018	FY2018 Q1	2017 Jul	2017 Jul, 03	201707
07/04/2017 00:00:00	FY2018	FY2018 Q1	2017 Jul	2017 Jul, 04	201707
07/05/2017 00:00:00	FY2018	FY2018 Q1	2017 Jul	2017 Jul, 05	201707
07/06/2017 00:00:00	FY2018	FY2018 Q1	2017 Jul	2017 Jul, 06	201707
07/07/2017 00:00:00	FY2018	FY2018 Q1	2017 Jul	2017 Jul, 07	201707

Understand row context

Power BI evaluates a calculated column's formula for each row in the table, which is called row context. Row context just means "the current row." For example, here's the `Due Fiscal Year` calculated column:

```
Due Fiscal Year =  
"FY"  
    & YEAR('Due Date'[Due Date])  
    + IF(  
        MONTH('Due Date'[Due Date]) <= 6,  
        1  
    )
```

When Power BI runs this formula, `'Due Date'[Due Date]` gives the value from the current row. If you've used Excel tables, this idea might feel familiar.

Row context only applies to the current table. If you need values from another table, you have two main options:

- If a relationship exists between the two tables, use the `RELATED` or `RELATEDTABLE` function. `RELATED` gets a value from the one-side of a relationship. `RELATEDTABLE` gets a table of values from the many-side.
- If there's no relationship, use the `LOOKUPVALUE` function.

Try to use `RELATED` when you can. It usually works faster than `LOOKUPVALUE` because of how Power BI stores and indexes data.

Here's an example. This formula adds a `Discount Amount` column to the `Sales` table:

```
Discount Amount =  
(  
    Sales[Order Quantity]  
    * RELATED('Product'[List Price])  
) - Sales[Sales Amount]
```

Power BI calculates this formula for each row in the `Sales` table. It gets `Order Quantity` and `Sales Amount` from the current row. To get `List Price` from the `Product` table, it uses the `RELATED` function to find the right value for each sale.

Row context always applies when Power BI evaluates calculated column formulas. It also comes into play with iterator functions, which let you create more advanced summaries. You learn about iterator functions later in this module.

4. Understand implicit measures

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-create-calculations/4-implicit-measures>

Understand implicit measures

Completed

Measures in Power BI models are either *implicit* or *explicit*. Implicit measures are automatic behaviors that let visuals summarize column data. Explicit measures, often called *measures*, are calculations you add to your model. This section explains how implicit measures work and how you can use them.

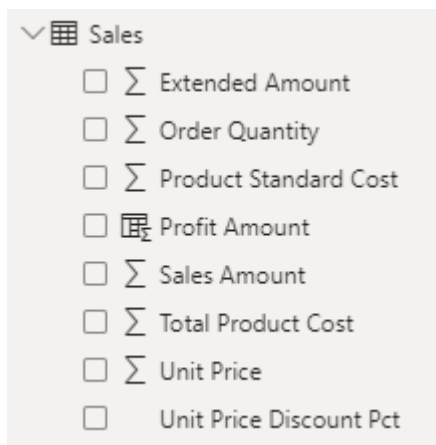
Identify implicit measures

As a data modeler, you control how a column summarizes by setting the **Summarization** property. You can choose **Don't summarize** or select a specific aggregation function. If you set a column to **Don't summarize**, the sigma symbol disappears in the **Data** pane.

In the **Data** pane, a column with the sigma symbol (Σ) shows two facts:

- The column is numeric.
- Values are summarized when used in a visual that supports summarization.

The following image shows the `Sales` table with implicit measures, a calculated column, and one column that can't be summarized.



Notice in the example with the **Sales** table, if you add the **Sales Amount** field from the **Sales** table to a matrix visual that groups by fiscal year and month, Power BI summarizes the values implicitly. The **Sum** aggregation function is selected by default.

Fiscal Year	Sales Amount
FY2018	\$23,860,891.17
2017 Jul	\$1,423,357.32
2017 Aug	\$2,057,902.45
2017 Sep	\$2,523,947.55
2017 Oct	\$561,681.48
2017 Nov	\$4,764,920.16
2017 Dec	\$596,746.56

If you add the **Unit Price** field to the matrix visual, Power BI uses **Average** as the default summarization, because summing unit prices doesn't make sense since they're rates, not totals.

Fiscal Year	Sales Amount	Unit Price
FY2018	\$23,860,891.17	\$1,273.63
2017 Jul	\$1,423,357.32	\$1,817.15
2017 Aug	\$2,057,902.45	\$1,181.42
2017 Sep	\$2,523,947.55	\$1,030.56
2017 Oct	\$561,681.48	\$3,228.05
2017 Nov	\$4,764,920.16	\$1,009.71
2017 Dec	\$596,746.56	\$3,174.18

The default summarization is now set to **Average** (the modeler knows that it's inappropriate to sum unit price values together because they're rates, which are non-additive).

Implicit measures allow the report author to start with a default summarization technique and lets them modify it to suit their visual requirements. Numeric columns support the widest range of aggregation functions to choose from, including:

- Sum
- Average
- Minimum

- Maximum
- Count (Distinct)
- Count
- Standard deviation
- Variance
- Median

Summarize non-numeric columns

Non-numeric columns like text, dates, and boolean (true/false) values can also be summarized in your visuals. While the sigma symbol (Σ) only appears next to numeric fields in the Data pane, these columns are still being aggregated.

- Text columns: First, Last, Count, Count (Distinct)
- Date columns: Earliest, Latest, Count, Count

This flexibility is useful when you want to answer questions such as:

- "How many unique products are there?" (Count distinct on a text column)
- "What was the earliest order date?" (Earliest on a date column)
- "How many orders were marked as complete?" (Count on a boolean column)

You can choose the aggregation option that best fits your analysis when you add a non-numeric field to your visual.

Considerations for implicit measures

Implicit measures are easy to use and flexible. They let report authors start with a default summarization and change it to fit their needs. Implicit measures let report authors quickly visualize data without needing to write calculations. As a data modeler, you spend less time creating explicit measures.

However, even if you set a default summarization, report authors can change it to something that might not make sense. For example, they could set **Unit Price** to **Sum**, which produces misleading results, as shown in the following image. The Unit Price values are large because they're the sum of unit prices instead of the static unit price per product.

Fiscal Year	Sales Amount	Sum of Unit Price
FY2018	\$23,860,891.17	\$13,905,519.77
2017 Jul	\$1,423,357.32	\$1,164,795.52
2017 Aug	\$2,057,902.45	\$1,115,258.56
2017 Sep	\$2,523,947.55	\$1,291,296.17
2017 Oct	\$561,681.48	\$561,681.48
2017 Nov	\$4,764,920.16	\$2,159,760.38
2017 Dec	\$596,746.56	\$596,746.56

The biggest limitation is that implicit measures only work for simple scenarios. They can summarize column values using a single aggregation function, but they can't handle more complex calculations. For example, if you need to calculate the ratio of each month's sales amount over the yearly sales amount, you must create an explicit measure with a DAX formula.

5. Create explicit measures

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-create-calculations/5-explicit-measures>

Create explicit measures

Completed

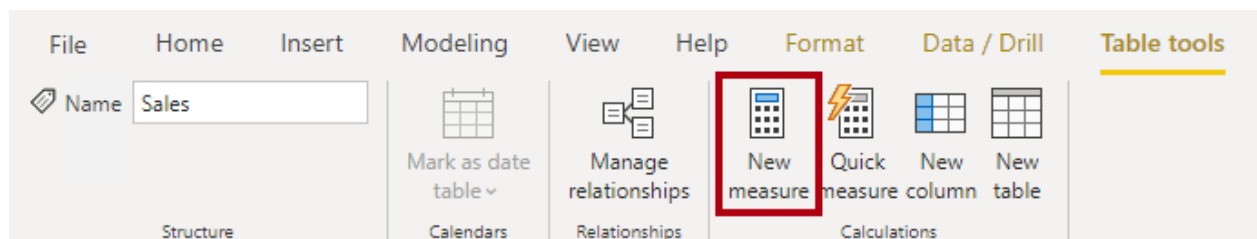
You can add a measure to any table in your model by writing a DAX formula. A measure formula must return a single value. This section explains how to create explicit measures.

Measures don't store values in the model. Instead, Power BI calculates them at query time to summarize model data. Measures can't reference a table or column directly, so you must use a function to summarize the data.

Simple measures

A *simple* measure aggregates the values of a single column, just like an implicit measure.

For example, you can add a measure to the **Sales** table. In the **Data** pane, select the **Sales** table. On the **Table Tools** ribbon, select **New measure**.



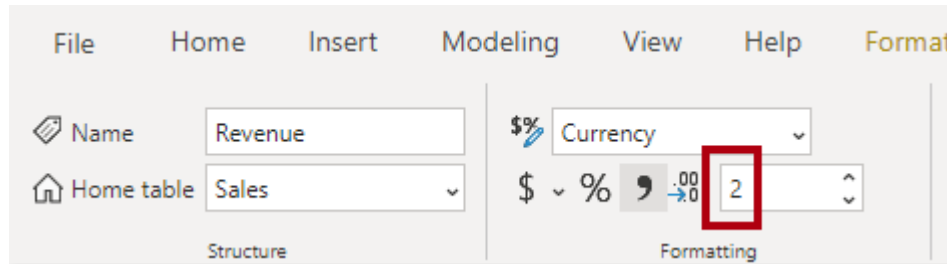
The following formula creates a measure called *Revenue*. This measure uses the **SUM** function to total the values in the **Sales Amount** column. If you add this measure to a table alongside the **Sales Amount** implicit measure, the results are the same.

```
Revenue =  
SUM(Sales[Sales Amount])
```

Tip

Adding measures and hiding columns helps report authors use the explicit measures instead.

In the **Measure tools** contextual ribbon, you can format the measure, set data type, and change the home table. The home table refers to where the measure shows when looking at the data pane.



Tip

It's a good practice to set the formatting options right after you create a measure. This ensures your values look consistent in all report visuals.

Consider you might need a measure to calculate **Profit** as shown in the following code:

```
Profit =  
SUM(Sales[Profit Amount])
```

In this example, the **Profit Amount** column is a calculated column. This approach isn't optimal because you don't need that column. In the next section, you see how to create a measure that calculates profit directly, which reduces model size and improves refresh times.

The following code creates two different measures to return **Order Line Count** and **Order Count**. The **COUNT** function counts non-BLANK values in a column. The **DISTINCTCOUNT** function counts unique values. Since an order can have multiple order lines, the **Sales Order** column has duplicates. Using **DISTINCTCOUNT** gives you the correct order count.

```
Order Line Count =  
COUNT(Sales[SalesOrderLineKey])  
  
Order Count =  
DISTINCTCOUNT('Sales Order'[Sales Order])
```

You can also write the **Order Line Count** measure using **COUNTROWS**, which counts the number of rows in a table:

```
Order Line Count =  
COUNTROWS(Sales)
```

All the measures referenced are considered simple measures because they aggregate a single column or single table.

Fiscal Year	Order Count	Order Line Count	Cost	Minimum Price	Average Price	Maximum Price	Profit	Quantity	Revenue
FY2018	3,198	10,918	\$20,824,958	\$4.75	\$1,274	\$3,578.27	\$3,035,934	24,349	\$23,860,891
FY2019	4,738	26,050	\$30,362,108	\$2.29	\$567	\$2,443.35	\$3,708,000	83,339	\$34,070,109
FY2020	23,519	84,285	\$46,070,842	\$1.33	\$329	\$2,443.35	\$5,807,433	167,088	\$51,878,275
Total	31,455	121,253	\$97,257,908	\$1.33	\$465	\$3,578.27	\$12,551,366	274,776	\$109,809,274

Create compound measures

A *compound measure* references one or more other measures. For example, you can redefine the **Profit** measure by referencing other measures. This measure can be used instead of the calculated column previously referenced.

```
Profit =
[Revenue] - [Cost]
```

This change to the model presents an important lesson: By removing the calculated column, you optimize the semantic model because it results in a decreased semantic model size and shorter data refresh times. The **Profit Amount** calculated column wasn't required after all because the **Profit** measure can directly produce the required result by using existing measures.

Note

Sometimes, it makes sense to define measures that depend on other measures. Always test changes carefully, because updates can affect all dependent measures.

Use Quick measures

Quick measures let you perform common calculations without writing DAX yourself. Power BI generates the DAX expression for you, which helps you learn and build your DAX skills.

For example, you can use a Quick measure to create a **Profit Margin** measure through the following steps:

1. Select Quick measure in the Table tools ribbon.
2. Choose Mathematical operations > Division.
3. Add the **Profit** measure into **Numerator** and **Revenue** into **Denominator**.

6. Use iterator functions

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-create-calculations/6-iterator-functions>

Use iterator functions

Completed

Iterator functions evaluate an expression for each row in a table. They give you flexibility and control over how your model summarizes data.

Single-column summarization functions, such as `SUM`, `COUNT`, `MIN`, and `MAX`, have equivalent iterator functions with an "X" suffix, like `SUMX`, `COUNTX`, `MINX`, and `MAXX`. Specialized iterator functions also exist for filtering, ranking, and semi-additive calculations over time.

Every iterator function requires a table and an expression. The table can be a model table or any expression that returns a table. The expression must return a single value for each row.

Single-column summarization functions, like `SUM`, act as shorthand. Power BI internally converts `SUM` to `SUMX`. For example, both of the following measures return the same result and have the same performance:

```
Revenue = SUM(Sales[Sales Amount])
```

```
Revenue =  
SUMX(  
    Sales,  
    Sales[Sales Amount]  
)
```

Iterator functions evaluate the expression for each row in a table, using row context—meaning they process one row at a time to compute the final result. Then the table is evaluated in filter context. For example, if a report visual filters by fiscal year FY2020, the `Sales` table contains only sales rows from that year.

Important

Using iterator functions with large tables and complex expressions can slow performance. Functions like `SEARCH` and `LOOKUPVALUE` can be expensive. When possible, use `RELATED` for better performance.

Iterator functions for complex summarization

Iterator functions let you aggregate more than a single column. For example, a revenue measure can multiply order quantity, unit price, and a discount factor for each row, then sum the results.

```
Revenue =  
SUMX(  
    Sales,  
    Sales[Order Quantity] * Sales[Unit Price] * (1 - Sales[Unit Price Discount Pct])  
)
```

Iterator functions can also reference related tables. The discount measure can use the `RELATED` function to access the list price from the product table:

```
Discount =  
SUMX(  
    Sales,  
    Sales[Order Quantity]  
    * (  
        RELATED('Product'[List Price]) - Sales[Unit Price]  
    )  
)
```

The following image shows a table visual with the Month, Revenue, and Discount columns. Revenue and Discount are the measures previously created.

Month	Revenue	Discount
2017 Jul	\$1,423,357.32	\$326,219.06
2017 Aug	\$2,057,902.45	\$1,031,506.86
2017 Sep	\$2,523,947.55	\$1,342,355.67
2017 Oct	\$561,681.48	\$0.00
2017 Nov	\$4,764,920.16	\$2,692,205.61
2017 Dec	\$596,746.56	\$0.00
2018 Jan	\$1,327,674.63	\$475,813.07
2018 Feb	\$3,936,463.31	\$2,237,412.77
2018 Mar	\$700,873.18	\$0.00
2018 Apr	\$1,519,275.24	\$588,995.26
2018 May	\$2,960,378.09	\$1,514,891.49
2018 Jun	\$1,487,671.19	\$1,659,893.12

Iterator functions for higher grain summarization

Iterator functions can also summarize data at different levels of detail (grain). For example, you might want to calculate an average at the line item level or at the sales order level.

In this example, the **Sales** table contains one row for each line item in a sales order. Each row includes details such as the sales order number, product, quantity sold, unit price, and discount. Multiple rows can have the same sales order number, representing different items within the same order.

To calculate the average revenue per order line (line item), you can use the **AVERAGEX** function to iterate over each row in the **Sales** table. The formula calculates revenue for each line item, then averages the result across all line items in the current filter context:

```
Revenue Avg Order Line =
AVERAGEX(
    Sales,
    Sales[Order Quantity] * Sales[Unit Price] * (1 - Sales[Unit Price Discount Pct.])
)
```

Month	Revenue	Discount	Revenue Avg
2017 Jul	\$1,423,357.32	\$326,219.06	\$2,220.53
2017 Aug	\$2,057,902.45	\$1,031,506.86	\$2,179.98
2017 Sep	\$2,523,947.55	\$1,342,355.67	\$2,014.32
2017 Oct	\$561,681.48	\$0.00	\$3,228.05
2017 Nov	\$4,764,920.16	\$2,692,205.61	\$2,227.64
2017 Dec	\$596,746.56	\$0.00	\$3,174.18
2018 Jan	\$1,327,674.63	\$475,813.07	\$2,329.25
2018 Feb	\$3,936,463.31	\$2,237,412.77	\$2,321.03
2018 Mar	\$700,873.18	\$0.00	\$3,200.33
2018 Apr	\$1,519,275.24	\$588,995.26	\$2,182.87
2018 May	\$2,960,378.09	\$1,514,891.49	\$2,219.17
2018 Jun	\$1,487,671.19	\$1,659,893.12	\$1,398.19

If you want to calculate the average revenue per sales order (rather than per line item), you can use the **VALUES** function to get a list of unique sales order numbers first. Then, **AVERAGEX** iterates over each sales order and averages the total revenue for each order:

```
Revenue Avg Order =
AVERAGEX(
    VALUES('Sales Order'[Sales Order]),
    [Revenue]
)
```

The **VALUES** function returns the unique sales orders based on the current filter context, so **AVERAGEX** iterates over each sales order for each month.

Month	Revenue	Discount	Revenue Avg Order Line	Revenue Avg Order
2017 Jul	\$1,423,357.32	\$326,219.06	\$2,220.53	\$4,352.77
2017 Aug	\$2,057,902.45	\$1,031,506.86	\$2,179.98	\$8,794.45
2017 Sep	\$2,523,947.55	\$1,342,355.67	\$2,014.32	\$9,670.30
2017 Oct	\$561,681.48	\$0.00	\$3,228.05	\$3,228.05
2017 Nov	\$4,764,920.16	\$2,692,205.61	\$2,227.64	\$12,441.04
2017 Dec	\$596,746.56	\$0.00	\$3,174.18	\$3,174.18
2018 Jan	\$1,327,674.63	\$475,813.07	\$2,329.25	\$5,698.17
2018 Feb	\$3,936,463.31	\$2,237,412.77	\$2,321.03	\$12,301.45
2018 Mar	\$700,873.18	\$0.00	\$3,200.33	\$3,200.33
2018 Apr	\$1,519,275.24	\$588,995.26	\$2,182.87	\$6,356.80
2018 May	\$2,960,378.09	\$1,514,891.49	\$2,219.17	\$9,642.93
2018 Jun	\$1,487,671.19	\$1,659,893.12	\$1,398.19	\$4,752.94

Ranking with iterator functions

The `RANKX` function calculates ranks by iterating over a table and evaluating an expression for each row.

Order direction can be ascending or descending. Ranking revenue usually uses descending order, so the highest value ranks first. Ranking something like complaints might use ascending order, so the lowest value ranks first. By default, `RANKX` uses descending order and skips ranks for ties.

For example, a product quantity rank measure can use `RANKX` and the `ALL` function to rank products by quantity:

```
Product Quantity Rank =
RANKX(
    ALL('Product'[Product]),
    [Quantity]
)
```

The `ALL` function removes filters, so `RANKX` ranks all products. In the following image, two products tie for tenth place, so the next product is ranked twelfth and rank 11 is skipped.

Bike Sales

Product	Quantity	Product Quantity Rank
Mountain-200 Black, 38	2,977	1
Mountain-200 Black, 42	2,664	2
Mountain-200 Silver, 38	2,394	3
Road-650 Black, 52	2,265	4
Road-650 Red, 44	2,244	5
Mountain-200 Silver, 42	2,234	6
Road-650 Red, 60	2,221	7
Mountain-200 Silver, 46	2,216	8
Mountain-200 Black, 46	2,111	9
Road-650 Red, 48	1,886	10
Road-650 Red, 62	1,886	10
Road-650 Black, 58	1,865	12
Road-550-W Yellow, 48	1,763	13
Road-550-W Yellow, 38	1,744	14
Road-250 Black, 44	1,642	15



You can also use dense ranking, which assigns the next rank after a tie without skipping numbers. To use dense ranking, the measure can include the `DENSE` argument:

```
Product Quantity Rank =
RANKX(
    ALL('Product'[Product]),
    [Quantity],
    ,
    ,
    DENSE
)
```

Now, after two products tie for tenth place, the next product is ranked eleventh and numbering continues sequentially without skipping rank 11.

Bike Sales		
Product	Quantity	Product Quantity Rank
Mountain-200 Black, 38	2,977	1
Mountain-200 Black, 42	2,664	2
Mountain-200 Silver, 38	2,394	3
Road-650 Black, 52	2,265	4
Road-650 Red, 44	2,244	5
Mountain-200 Silver, 42	2,234	6
Road-650 Red, 60	2,221	7
Mountain-200 Silver, 46	2,216	8
Mountain-200 Black, 46	2,111	9
Road-650 Red, 48	1,886	10
Road-650 Red, 62	1,886	10
Road-650 Black, 58	1,865	11
Road-550-W Yellow, 48	1,763	12
Road-550-W Yellow, 38	1,744	13
Road-250 Black, 44	1,642	14

In this visual, the total row for the **Product Quantity Rank** measure shows one, because the total for all products is also ranked and there's only one value.

Road-350-W Yellow, 40	1,477	19
Road-750 Black, 52	1,338	20
Total	90,220	1

To avoid ranking the total, the measure can use the **HASONEVALUE** function to return BLANK unless a single product is filtered:

```
Product Quantity Rank =
IF(
    HASONEVALUE('Product'[Product]),
    RANKX(
        ALL('Product'[Product]),
        [Quantity],
        ,
        DENSE
    )
)
```

Now, the total for **Product Quantity Rank** is blank.

Road-350-W Yellow, 40	1,477	19
Road-750 Black, 52	1,338	20
Total	90,220	

The `HASONEVALUE` function checks if the product column has a single value in filter context. This is true for each product group, but not for the total, which represents all products.

Iterator functions provide powerful ways to summarize, aggregate, and rank data in Power BI models. They support complex calculations and let you control the level of detail in your reports.

7. Exercise - Create DAX calculations

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-create-calculations/7-lab>

Exercise - Create DAX calculations

Completed

In this exercise, you learn how to use DAX to:

- Create calculated tables.
- Create calculated columns.
- Create measures.

This lab takes approximately **45** minutes to complete.

Note

A virtual machine containing the client tools you need is provided, along with the exercise instructions. Use the "Launch lab" button to launch the virtual machine.

A limited number of concurrent sessions are available. If the hosted environment is unavailable, please try again later.

Alternatively, you can [open the instructions in a separate window](#).

Access your environment

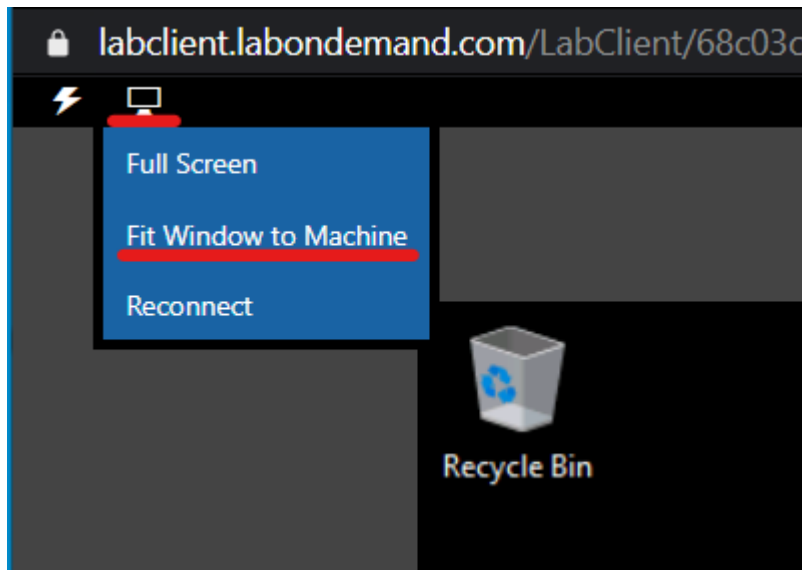
Before you start this lab (unless you are continuing from a previous lab), select **Launch lab** above.

You are automatically logged in to your lab environment as data-ai\student.

You can now begin your work on this lab.

Tip

To dock the lab environment so that it fills the window, select the PC icon at the top and then select **Fit Window to Machine**.



8. Check your knowledge

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-create-calculations/8-check>

Check your knowledge

Completed

9. Summary

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-create-calculations/9-summary>

Summary

Completed

This module covered using DAX calculations to extend your semantic model. You learned the differences between calculated columns and measures, including when and how Power BI evaluates

them and how they store data. Explicit measures are important because they allow you to create complex DAX formulas to achieve the precise calculations that your report visuals need.

You also learned how calculated columns are evaluated within row context, and iterator functions are available in measures for advanced row-by-row calculations.

Understanding how to create and use DAX calculations is fundamental for building effective, flexible, and maintainable semantic models in Power BI. These concepts help you design reports that deliver accurate insights and support a wide range of analytical requirements.

For more information, see [Introduction to DAX in Power BI](#).

Modify DAX filter context in semantic models

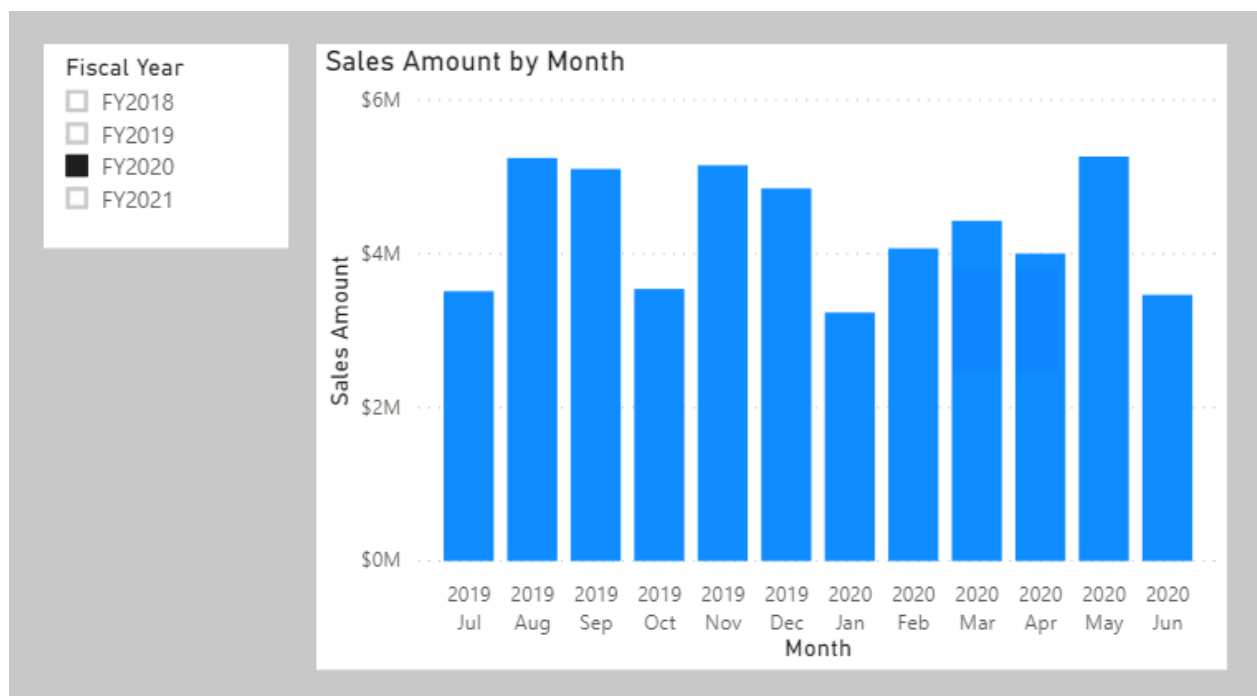
1. Understand filter context

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-modify-filter/1a-understand-filter-context>

Understand filter context

Completed

At report design time, filters are applied in the **Filters** pane or to report visuals. The slicer visual is an example of a visual whose only purpose is to filter the report page (and other pages when it's configured as a synced slicer). Report visuals, which perform grouping, also apply filters. They're implied filters; the difference is that the filter result is visible in the visual. For example, a stacked column chart visual can filter by fiscal year FY2020, group by month, and summarize sales amount. The fiscal year filter isn't visible in the visual result, yet the grouping, which results in a column for each month, behaves as a filter.



Not all filters are applied at report design time. Filters can be added when a report user interacts with the report. They can modify filter settings in the **Filters** pane, and they can cross-filter or cross-highlight visuals by selecting visual elements like columns, bars, or pie chart segments. These interactions apply other filters to report page visuals (unless interactions have been disabled).

It's important to understand how filter context works. It guides you in defining the correct formula for your calculations. As you write more complex formulas, you learn to identify when you need to add, modify, or remove filters to achieve the desired result.

Consider an example that requires your formula to modify the filter context. Your objective is to produce a report visual that shows each sales region together with its revenue and revenue *as a percentage of total revenue*.

Region	Revenue	Revenue % Total Region
Australia	\$10,655,335.96	9.70%
Canada	\$16,355,770.46	14.89%
Central	\$7,909,009.01	7.20%
France	\$7,251,555.65	6.60%
Germany	\$4,878,300.38	4.44%
Northeast	\$6,939,374.48	6.32%
Northwest	\$16,084,942.55	14.65%
Southeast	\$7,879,655.07	7.18%
Southwest	\$24,184,609.60	22.02%
United Kingdom	\$7,670,721.04	6.99%
Total	\$109,809,274.20	100.00%

The **Revenue % Total Region** measure result is achieved by defining a measure expression that's the ratio of revenue divided by revenue *for all regions*. Therefore, for Australia, the ratio is 10,655,335.96 dollars divided by 109,809,274.20 dollars, which is 9.7 percent.

The numerator expression doesn't need to modify filter context; it should use the current filter context (a visual that groups by region applies a filter for that region). The denominator expression, however, needs to remove any region filters to achieve the result for all regions.

Tip

The key to writing complex measures is understanding these concepts:

- How filter context works.
- When and how to modify or remove filters to achieve a required result.
- How to write a formula to accurately and efficiently modify filter context.

These concepts take practice and time to fully understand. Rarely will students understand the concepts from the beginning of training. Therefore, be patient and persevere with the theory

and activities. We recommend that you repeat this module at a later time to help reinforce key lessons.

The next unit introduces the `CALCULATE` function. It's one of the most powerful DAX functions, allowing you to modify filter context when your formulas are evaluated.

2. Introduction

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-modify-filter/1-introduction>

Introduction

Completed

Filter context describes the filters that are applied during the evaluation of a measure or measure expression.

To introduce this important concept, watch the following video to learn about filter context.

Filters can be applied directly to columns, like a filter on the `Fiscal Year` column in the `Date` table for the value `FY2020`. Additionally, filters can be applied indirectly, which happens when model relationships propagate filters to other tables. For example, the `Sales` table receives a filter through its relationship with the `Date` table, filtering the `Sales` table rows to return rows with an `OrderDateKey` column value in `FY2020`.

It's important to understand that calculated tables and calculated columns aren't evaluated within filter context. Calculated columns are evaluated in row context, though the formula can transition the row context to filter context, if it needs to summarize model data.

Note

Many units in this module present inline activities to follow along and create calculations. These activities are optional. If you want to complete the activities, you can download the Power BI

Desktop file by using the links provided in the units.

At the end of this module, you have the opportunity to complete an exercise in a virtual machine to apply the lessons on how to create calculated tables, calculated columns, and simple measures using DAX.

3. Modify filter context

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-modify-filter/2-modify-filter-context>

Modify filter context

Completed

You can use the `CALCULATE` function to modify filter context in your formulas. The syntax for the `CALCULATE` function is as follows:

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

The function requires passing in an expression that returns a scalar value and as many filters as you need. The expression can be a measure (which is a named expression) or any expression that can be evaluated in filter context.

Filters can be Boolean expressions or table expressions. It's also possible to pass in filter modification functions that provide more control when you're modifying filter context.

When you have multiple filters, they're evaluated by using the `AND` logical operator, which means that all conditions must be `TRUE` at the same time.

Note

The `CALCULATETABLE` function performs exactly the same functionality as the `CALCULATE` function, except that it modifies the filter context that's applied to an expression that returns a table object. In this module, the explanations and examples use the `CALCULATE` function, but keep in mind that these scenarios could also apply to the `CALCULATETABLE` function.

Apply Boolean expression filters

A Boolean expression filter is an expression that evaluates to `TRUE` or `FALSE`. Boolean filters must abide by the following rules:

- They can reference only a single column.
- They cannot reference measures.
- They cannot use functions that scan or return a table that includes aggregation functions like `SUM`.

To perform the tasks in this example, download and open the [Adventure Works DW 2020 M06.pbix](#) file. Then add the following measure to the `Sales` table that filters the `Revenue` measure by using a Boolean expression filter for red products.

```
Revenue Red =
CALCULATE(
    [Revenue],
    'Product'[Color] = "Red"
)
```

Add the `Revenue Red` measure to the table visual that is found on **Page 1** of the report.

Region	Revenue	Revenue Red
Australia	\$10,655,335.96	\$2,681,324.79
Canada	\$16,355,770.46	\$3,573,412.99
Central	\$7,909,009.01	\$1,585,997.34
France	\$7,251,555.65	\$1,051,014.15
Germany	\$4,878,300.38	\$670,607.30
Northeast	\$6,939,374.48	\$1,876,016.33
Northwest	\$16,084,942.55	\$2,292,905.61
Southeast	\$7,879,655.07	\$1,457,221.07
Southwest	\$24,184,609.60	\$5,345,637.47
United Kingdom	\$7,670,721.04	\$1,063,753.75
Total	\$109,809,274.20	\$21,597,890.81

In this next example, the following measure filters the `Revenue` measure by multiple colors. Notice the use of the `IN` operator followed by a list of color values.

```
Revenue Red or Blue =
CALCULATE(
    [Revenue],
    'Product'[Color] IN {"Red", "Blue"}
)
```

The following measure filters the `Revenue` measure by expensive products. Expensive products are those with a list price greater than 1,000 US dollars.

```
Revenue Expensive Products =
CALCULATE(
    [Revenue],
```

```
'Product'[List Price] > 1000  
)
```

Apply table expression filters

A table expression filter applies a table object as a filter. It could be a reference to a model table; however, it's likely a DAX function that returns a table object.

Commonly, the `FILTER` function is used to apply complex filter conditions, including those that can't be defined by a Boolean filter expression. The `FILTER` function is classed as an iterator function, and so you would pass in a table, or table expression, and an expression to evaluate for each row of that table.

The `FILTER` function returns a table object with exactly the same structure as one that the table passed in. Its rows are a subset of those rows that were passed in, meaning the rows where the expression evaluated as `TRUE`.

The following example shows a table filter expression that uses the `FILTER` function:

```
Revenue High Margin Products =  
CALCULATE(  
    [Revenue],  
    FILTER(  
        'Product',  
        'Product'[List Price] > 'Product'[Standard Cost] * 2  
    )  
)
```

In this example, the `FILTER` function filters all rows of the `Product` table that are in filter context. Each row for a product where its list price exceeds double its standard cost is displayed as a row of the filtered table. Therefore, the `Revenue` measure is evaluated for all products that are returned by the `FILTER` function.

All filter expressions that are passed in to the `CALCULATE` function are table filter expressions. A Boolean filter expression is a shorthand notation to improve the writing and reading experience. Internally, Microsoft Power BI translates Boolean filter expressions to table filter expressions, which is how it translates your `Revenue Red` measure definition.

```
Revenue Red =  
CALCULATE(  
    [Revenue],  
    FILTER(  
        'Product',  
        'Product'[Color] = "Red"  
    )  
)
```

Filter behavior

Two possible standard outcomes occur when you add filter expressions to the `CALCULATE` function:

- If the columns (or tables) aren't in filter context, then new filters are added to the filter context to evaluate the `CALCULATE` expression.
- If the columns (or tables) are already in filter context, the existing filters are overwritten by the new filters to evaluate the `CALCULATE` expression.

The following examples show how adding filter expressions to the `CALCULATE` function works.

Note

In each of the examples, no filters are applied to the table visual.

As in the previous activity, the `Revenue Red` measure was added to a table visual that groups by region and displays revenue.

Region	Revenue	Revenue Red
Australia	\$10,655,335.96	\$2,681,324.79
Canada	\$16,355,770.46	\$3,573,412.99
Central	\$7,909,009.01	\$1,585,997.34
France	\$7,251,555.65	\$1,051,014.15
Germany	\$4,878,300.38	\$670,607.30
Northeast	\$6,939,374.48	\$1,876,016.33
Northwest	\$16,084,942.55	\$2,292,905.61
Southeast	\$7,879,655.07	\$1,457,221.07
Southwest	\$24,184,609.60	\$5,345,637.47
United Kingdom	\$7,670,721.04	\$1,063,753.75
Total	\$109,809,274.20	\$21,597,890.81

Because no filter is applied on the `Color` column in the `Product` table, the evaluation of the measure adds a new filter to filter context. In the first row, the value of \$2,681,324.79 is for red products that were sold in the Australian region.

Switching the first column of the table visual from `Region` to `Color` produces a different result because the `Color` column in the `Product` table is now in filter context.

Color	Revenue	Revenue Red Filter
Black	\$38,236,124.06	
Blue	\$9,602,850.97	
Multi	\$649,030.25	
NA	\$1,099,303.91	
Red	\$21,597,890.81	\$21,597,890.8068
Silver	\$19,777,339.95	
Silver/Black	\$147,483.91	
White	\$29,745.13	
Yellow	\$18,669,505.22	
Total	\$109,809,274.20	\$21,597,890.8068

The `Revenue Red` measure formula evaluates the `Revenue` measure by adding a filter on the `Color` column (to red) in the `Product` table. In this visual that groups by color, the measure formula overwrites the filter context with a new filter.

This result might or might not be what you want. The next unit introduces the `KEEPFILTERS` function, which is a filter modification function that you can use to preserve filters rather than overwrite them.

4. Use filter modifier functions

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-modify-filter/3-filter-modifier-functions>

Use filter modifier functions

Completed

When using the `CALCULATE` function, you can pass in filter modification functions, which allow you to accomplish more than adding filters alone.

Remove filters

Use the `REMOVEFILTERS` function as a `CALCULATE` filter expression to remove filters from filter context. It can remove filters from one or more columns or from all columns of a single table.

Note

The `REMOVEFILTERS` function is relatively new. In previous versions of DAX, you removed filters by using the `ALL` function or variants including the `ALLEXCEPT` and the `ALLNOBLANKROW` functions. These functions behave as both filter modifiers and as functions that return table objects of distinct values. These functions are mentioned now because you'll possibly find documentation and formula examples that remove filters by using them.

In the following example, you add a new measure to the `Sales` table that evaluates the `Revenue` measure but does so by removing filters from the `Sales Territory` table. Format the measure as currency with two decimal places.

```
Revenue Total Region = CALCULATE([Revenue], REMOVEFILTERS('Sales Territory'))
```

Now, add the `Revenue Total Region` measure to the matrix visual that is found on **Page 2** of the report. The matrix visual will group by three columns from the `Sales Territory` table on the rows: `Group`, `Country`, and `Region`.

Reseller Revenue

Group	Country	Region	Revenue	Revenue Total Region
Corporate HQ	Corporate HQ	Corporate HQ		\$80,450,596.98
		Total		\$80,450,596.98
	Total			\$80,450,596.98
Europe	France	France	\$4,607,537.94	\$80,450,596.98
		Total	\$4,607,537.94	\$80,450,596.98
	Germany	Germany	\$1,983,988.04	\$80,450,596.98
		Total	\$1,983,988.04	\$80,450,596.98
	United Kingdom	United Kingdom	\$4,279,008.83	\$80,450,596.98
		Total	\$4,279,008.83	\$80,450,596.98
	Total		\$10,870,534.80	\$80,450,596.98
North America	Canada	Canada	\$14,377,925.60	\$80,450,596.98
		Total	\$14,377,925.60	\$80,450,596.98
	United States	Central	\$7,906,008.18	\$80,450,596.98
		Northeast	\$6,932,842.01	\$80,450,596.98
		Northwest	\$12,435,076.00	\$80,450,596.98
		Southeast	\$7,867,416.23	\$80,450,596.98
		Southwest	\$18,466,458.79	\$80,450,596.98
		Total	\$53,607,801.21	\$80,450,596.98
	Total		\$67,985,726.81	\$80,450,596.98
Pacific	Australia	Australia	\$1,594,335.38	\$80,450,596.98
		Total	\$1,594,335.38	\$80,450,596.98
	Total		\$1,594,335.38	\$80,450,596.98
Total			\$80,450,596.98	\$80,450,596.98

Notice that each `Revenue Total Region` measure value is the same. It's the value of total revenue.

While this result on its own isn't useful, when it's used as a denominator in a ratio, it calculates a percent of grand total. Therefore, you'll now overwrite the `Revenue Total Region` measure definition with the following definition. This new definition changes the measure name and declares two variables. Be sure to format the measure as a percentage with two decimal places.

```
Revenue % Total Region =
VAR CurrentRegionRevenue = [Revenue]
VAR TotalRegionRevenue =
    CALCULATE(
        [Revenue],
        REMOVEFILTERS('Sales Territory')
    )
RETURN
    DIVIDE(
        CurrentRegionRevenue,
```

TotalRegionRevenue

)

Verify that the matrix visual now displays the **Revenue % Total Region** values.

Reseller Revenue

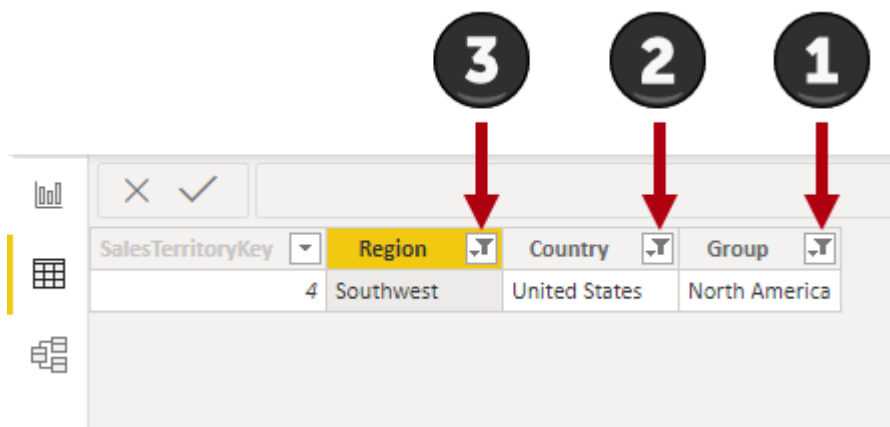
Group	Country	Region	Revenue	Revenue % Total Region
Europe	France	France	\$4,607,537.94	5.73%
		Total	\$4,607,537.94	5.73%
	Germany	Germany	\$1,983,988.04	2.47%
		Total	\$1,983,988.04	2.47%
	United Kingdom	United Kingdom	\$4,279,008.83	5.32%
		Total	\$4,279,008.83	5.32%
	Total		\$10,870,534.80	13.51%
North America	Canada	Canada	\$14,377,925.60	17.87%
		Total	\$14,377,925.60	17.87%
	United States	Central	\$7,906,008.18	9.83%
		Northeast	\$6,932,842.01	8.62%
		Northwest	\$12,435,076.00	15.46%
		Southeast	\$7,867,416.23	9.78%
		Southwest	\$18,466,458.79	22.95%
		Total	\$53,607,801.21	66.63%
	Total		\$67,985,726.81	84.51%
Pacific	Australia	Australia	\$1,594,335.38	1.98%
		Total	\$1,594,335.38	1.98%
	Total		\$1,594,335.38	1.98%
Total			\$80,450,596.98	100.00%

You'll now create another measure, but this time, you calculate the ratio of revenue for a region divided by its country's or region's revenue.

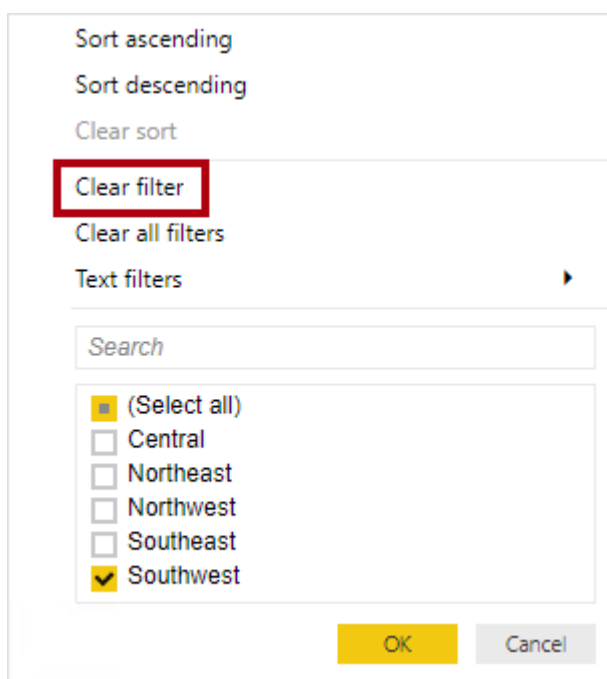
Before you complete this task, notice that the **Revenue % Total Region** value for the Southwest region is 22.95 percent. Investigate the filter context for this cell. Switch to data view and then, in the **Data** pane, select the **Sales Territory** table.

Apply the following column filters:

- **Group** - North America
- **Country** - United States
- **Region** - Southwest



Notice that the filters reduce the table to only one row. Now, while thinking about your new objective to create a ratio of the region revenue over its country's revenue, clear the filter from the **Region** column.



Notice that five rows now exist, each row belonging to the country United States. Accordingly, when you clear the **Region** column filters, while preserving filters on the **Country** and **Group** columns, you have a new filter context that's for the region's country.

In the following measure definition, notice how you can clear or remove a filter from a column. In DAX logic, it's a small and subtle change that's made to the **Revenue % Total Region** measure formula: The **REMOVEFILTERS** function now removes filters from the **Region** column instead of all columns of the **Sales Territory** table.

```
Revenue % Total Country =
VAR CurrentRegionRevenue = [Revenue]
VAR TotalCountryRevenue =
    CALCULATE(
        [Revenue],
```

```

    REMOVEFILTERS('Sales Territory'[Region])
  )
RETURN
  DIVIDE(
    CurrentRegionRevenue,
    TotalCountryRevenue
  )

```

Add the **Revenue % Total Country** measure to the **Sales** table and then format it as a percentage with two decimal places. Add the new measure to the matrix visual.

Reseller Revenue

Group	Country	Region	Revenue	Revenue % Total Region	Revenue % Total Country
Europe	France	France	\$4,607,537.94	5.73%	100.00%
		Total	\$4,607,537.94	5.73%	100.00%
	Germany	Germany	\$1,983,988.04	2.47%	100.00%
		Total	\$1,983,988.04	2.47%	100.00%
	United Kingdom	United Kingdom	\$4,279,008.83	5.32%	100.00%
		Total	\$4,279,008.83	5.32%	100.00%
	Total		\$10,870,534.80	13.51%	100.00%
	North America	Canada	Canada	\$14,377,925.60	17.87%
Total			\$14,377,925.60	17.87%	100.00%
United States		Central	\$7,906,008.18	9.83%	14.75%
		Northeast	\$6,932,842.01	8.62%	12.93%
		Northwest	\$12,435,076.00	15.46%	23.20%
		Southeast	\$7,867,416.23	9.78%	14.68%
		Southwest	\$18,466,458.79	22.95%	34.45%
		Total	\$53,607,801.21	66.63%	100.00%
Total		\$67,985,726.81	84.51%	100.00%	
Pacific	Australia	Australia	\$1,594,335.38	1.98%	100.00%
		Total	\$1,594,335.38	1.98%	100.00%
Total		\$1,594,335.38	1.98%	100.00%	
Total			\$80,450,596.98	100.00%	100.00%

Notice that all values, except those values for United States regions, are 100 percent. The reason is because, at the Adventure Works company, the United States has regions, while all other countries/regions don't.

Note

Tabular models don't support ragged hierarchies, which are hierarchies with variable depths. Therefore, it's a common design approach to repeat parent (or other ancestor) values at lower levels of the hierarchy. For example, Australia doesn't have a region, so the country/region value is repeated as the region name. It's always better to store a meaningful value instead of BLANK.

The next example is last measure that you'll create. Add the **Revenue % Total Group** measure, and then format it as a percentage with two decimal places. Then, add the new measure to the matrix visual.

```

Revenue % Total Group =
VAR CurrentRegionRevenue = [Revenue]
VAR TotalGroupRevenue =
    CALCULATE(
        [Revenue],
        REMOVEFILTERS(
            'Sales Territory'[Region],
            'Sales Territory'[Country]
        )
    )
RETURN
    DIVIDE(
        CurrentRegionRevenue,
        TotalGroupRevenue
    )

```

Reseller Revenue

Group	Country	Region	Revenue	Revenue % Total Region	Revenue % Total Country	Revenue % Total Group
Europe	France	France	\$4,607,537.94	5.73%	100.00%	42.39%
		Total	\$4,607,537.94	5.73%	100.00%	42.39%
	Germany	Germany	\$1,983,988.04	2.47%	100.00%	18.25%
		Total	\$1,983,988.04	2.47%	100.00%	18.25%
	United Kingdom	United Kingdom	\$4,279,008.83	5.32%	100.00%	39.36%
		Total	\$4,279,008.83	5.32%	100.00%	39.36%
	Total		\$10,870,534.80	13.51%	100.00%	100.00%
North America	Canada	Canada	\$14,377,925.60	17.87%	100.00%	21.15%
		Total	\$14,377,925.60	17.87%	100.00%	21.15%
	United States	Central	\$7,906,008.18	9.83%	14.75%	11.63%
		Northeast	\$6,932,842.01	8.62%	12.93%	10.20%
		Northwest	\$12,435,076.00	15.46%	23.20%	18.29%
		Southeast	\$7,867,416.23	9.78%	14.68%	11.57%
		Southwest	\$18,466,458.79	22.95%	34.45%	27.16%
		Total	\$53,607,801.21	66.63%	100.00%	78.85%
	Total		\$67,985,726.81	84.51%	100.00%	100.00%
Pacific	Australia	Australia	\$1,594,335.38	1.98%	100.00%	100.00%
		Total	\$1,594,335.38	1.98%	100.00%	100.00%
	Total		\$1,594,335.38	1.98%	100.00%	100.00%
Total			\$80,450,596.98	100.00%	100.00%	100.00%

When you remove filters from the **Region** and **Country** columns in the **Sales Territory** table, the measure calculates the region revenue as a ratio of its group's revenue.

Preserve filters

You can use the **KEEPFILTERS** function as a filter expression in the **CALCULATE** function to preserve filters.

To observe how to accomplish this task, switch to **Page 1** of the report. Then, modify the **Revenue Red** measure definition to use the **KEEPFILTERS** function.

```

Revenue Red =
CALCULATE(
    [Revenue],

```

```
KEEPFILTERS('Product'[Color] = "Red")
)
```

Color	Revenue	Revenue Red
Black	\$38,236,124.06	
Blue	\$9,602,850.97	
Multi	\$649,030.25	
NA	\$1,099,303.91	
Red	\$21,597,890.81	\$21,597,890.81
Silver	\$19,777,339.95	
Silver/Black	\$147,483.91	
White	\$29,745.13	
Yellow	\$18,669,505.22	
Total	\$109,809,274.20	\$21,597,890.81

In the table visual, notice that only one **Revenue Red** measure value exists. The reason is because the Boolean filter expression preserves existing filters on the **Color** column in the **Product** table. The reason why colors other than red are BLANK is because the filter contexts and the filter expressions are combined for these two filters. The color black and color red are intersected, and because both can't be **TRUE** at the same time, the expression is filtered by no product rows. It's only possible that both red filters can be **TRUE** at the same time, which explains why the one **Revenue Red** measure value is shown.

Use inactive relationships

An inactive model relationship can only propagate filters when the **USERELATIONSHIP** function is passed as a filter expression to the **CALCULATE** function. When you use this function to engage an inactive relationship, the active relationship will automatically become inactive.

Review an example of a measure definition that uses an inactive relationship to calculate the **Revenue** measure by shipped dates:

```
Revenue Shipped =
CALCULATE (
    [Revenue],
    USERELATIONSHIP('Date'[DateKey], Sales[ShipDateKey])
)
```

Modify relationship behavior

You can modify the model relationship behavior when an expression is evaluated by passing the `CROSSFILTER` function as a filter expression to the `CALCULATE` function. It's an advanced capability.

The `CROSSFILTER` function can modify filter directions (from both to single or from single to both) and even disable a relationship.

5. Examine filter context

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-modify-filter/4-examine-filter-context>

Examine filter context

Completed

The `VALUES` function lets your formulas determine what values are in filter context.

The `VALUES` function syntax is as follows:

```
VALUES(<TableNameOrColumnName>)
```

The function requires passing in a table reference or a column reference. When you pass in a table reference, it returns a table object with the same columns that contain rows for what's in filter context. When you pass in a column reference, it returns a single-column table of unique values that are in filter context.

The function always returns a table object and it's possible for a table to contain multiple rows. Therefore, to test whether a specific value is in filter context, your formula must first test that the `VALUES` function returns a single row. Two functions can help you accomplish this task: the `HASONEVALUE` and the `SELECTEDVALUE` functions.

The `HASONEVALUE` function returns `TRUE` when a given column reference has been filtered down to a single value.

The `SELECTEDVALUE` function simplifies the task of determining what a single value could be. When the function is passed a column reference, it returns a single value, or when more than one value is in filter context, it returns BLANK (or an alternate value that you pass to the function).

In the following example, you use the `HASONEVALUE` function. Add the following measure, which calculates sales commission, to the `Sales` table. At Adventure Works, the commission rate is 10 percent of revenue for all countries/regions except the United States. In the United States, salespeople earn 15 percent commission. Format the measure as currency with two decimal places, and then add it to the table that is found on **Page 3** of the report.

```

Sales Commission =
[Revenue]
* IF(
    HASONEVALUE('Sales Territory'[Country]),
    IF(
        VALUES('Sales Territory'[Country]) = "United States",
        0.15,
        0.1
    )
)

```

Region	Revenue	Sales Commission
Australia	\$10,655,335.96	\$1,065,533.60
Canada	\$16,355,770.46	\$1,635,577.05
Central	\$7,909,009.01	\$1,186,351.35
France	\$7,251,555.65	\$725,155.56
Germany	\$4,878,300.38	\$487,830.04
Northeast	\$6,939,374.48	\$1,040,906.17
Northwest	\$16,084,942.55	\$2,412,741.38
Southeast	\$7,879,655.07	\$1,181,948.26
Southwest	\$24,184,609.60	\$3,627,691.44
United Kingdom	\$7,670,721.04	\$767,072.10
Total	\$109,809,274.20	

Notice that the total **Sales Commission** measure result is BLANK. The reason is because multiple values are in filter context for the **Country** column in the **Sales Territory** table. In this case, the **HASONEVALUE** function returns **FALSE**, which results in the **Revenue** measure being multiplied by BLANK (a value multiplied by BLANK is BLANK). To produce a total, you need to use an iterator function, which is explained later in this module.

Three other functions that you can use to test filter state are:

- **ISFILTERED** - Returns **TRUE** when a passed-in column reference is *directly* filtered.
- **ISCROSSFILTERED** - Returns **TRUE** when a passed-in column reference is *indirectly* filtered. A column is cross-filtered when a filter that is applied to another column in the same table, or in a related table, affects the reference column by filtering it.
- **ISINSCOPE** - Returns **TRUE** when a passed-in column reference is the level in a hierarchy of levels.

Return to **Page 2** of the report, and then modify the **Revenue % Total Country** measure definition to test that the **Region** column in the **Sales Territory** table is in scope. If it's not in scope, the measure result should be BLANK.


```

Revenue % Total Country =
VAR CurrentRegionRevenue = [Revenue]
VAR TotalCountryRevenue =
    CALCULATE(
        [Revenue],
        REMOVEFILTERS('Sales Territory'[Region])
    )
RETURN
    IF(
        ISINSCOPE('Sales Territory'[Region]),
        DIVIDE(
            CurrentRegionRevenue,
            TotalCountryRevenue
        )
    )

```

Reseller Revenue

Group	Country	Region	Revenue	Revenue % Total Region	Revenue % Total Country	Revenue % Total Group
Europe	France	France	\$4,607,537.94	5.73%	100.00%	42.39%
		Total	\$4,607,537.94	5.73%		42.39%
	Germany	Germany	\$1,983,988.04	2.47%	100.00%	18.25%
		Total	\$1,983,988.04	2.47%		18.25%
	United Kingdom	United Kingdom	\$4,279,008.83	5.32%	100.00%	39.36%
		Total	\$4,279,008.83	5.32%		39.36%
	Total		\$10,870,534.80	13.51%		100.00%
North America	Canada	Canada	\$14,377,925.60	17.87%	100.00%	21.15%
		Total	\$14,377,925.60	17.87%		21.15%
	United States	Central	\$7,906,008.18	9.83%	14.75%	11.63%
		Northeast	\$6,932,842.01	8.62%	12.93%	10.20%
		Northwest	\$12,435,076.00	15.46%	23.20%	18.29%
		Southeast	\$7,867,416.23	9.78%	14.68%	11.57%
		Southwest	\$18,466,458.79	22.95%	34.45%	27.16%
		Total	\$53,607,801.21	66.63%		78.85%
	Total		\$67,985,726.81	84.51%		100.00%
Pacific	Australia	Australia	\$1,594,335.38	1.98%	100.00%	100.00%
		Total	\$1,594,335.38	1.98%		100.00%
	Total		\$1,594,335.38	1.98%		100.00%
Total			\$80,450,596.98	100.00%		100.00%

In the matrix visual, notice that 'Revenue % Total Country' measure values are now only displayed when a region is in scope.

6. Exercise - Modify DAX filter context

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-modify-filter/5a-lab>

Exercise - Modify DAX filter context

Completed

In this exercise, you learn how to:

- Use the `CALCULATE` function to manipulate filter context.

This lab takes approximately **30** minutes to complete.

Note

A virtual machine containing the client tools you need is provided, along with the exercise instructions. Use the "Launch lab" button to launch the virtual machine.

A limited number of concurrent sessions are available. If the hosted environment is unavailable, please try again later.

Alternatively, you can [open the instructions in a separate window](#).

Access your environment

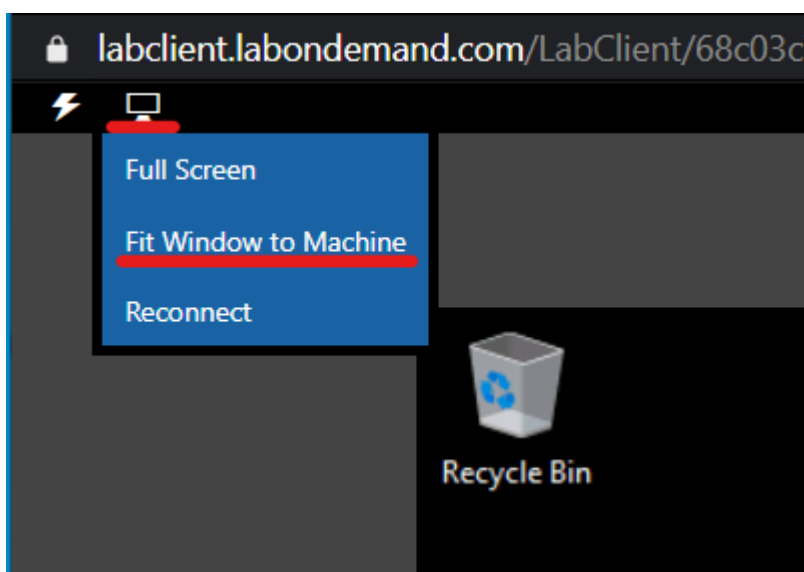
Before you start this lab (unless you are continuing from a previous lab), select **Launch lab** above.

You are automatically logged in to your lab environment as data-ai\student.

You can now begin your work on this lab.

Tip

To dock the lab environment so that it fills the window, select the PC icon at the top and then select **Fit Window to Machine**.



7. Perform context transition

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-modify-filter/5-context-transition>

Perform context transition

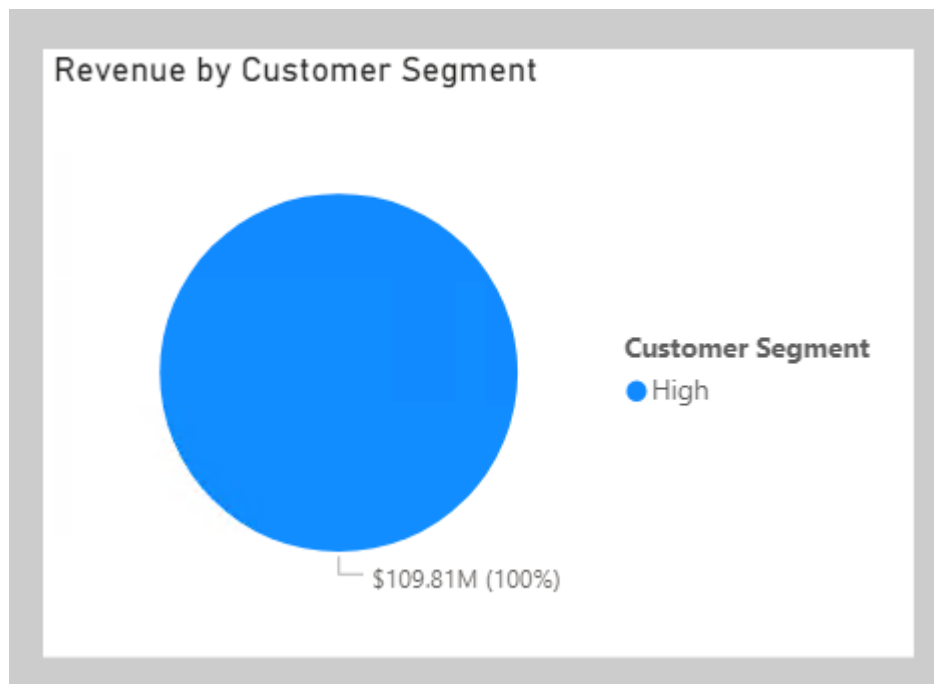
Completed

What happens when a measure or measure expression is evaluated within row context? This scenario can happen in a calculated column formula or when an expression in an iterator function is evaluated.

In the following example, you add a calculated column to the **Customer** table to classify customers into a loyalty class. The scenario is simple: When the revenue that is produced by the customer is less than \$2,500, the customer is classified as *Low*; otherwise they're classified as *High*.

```
Customer Segment =  
VAR CustomerRevenue = SUM(Sales[Sales Amount])  
RETURN  
    IF(CustomerRevenue < 2500, "Low", "High")
```

On **Page 4** of the report, add the **Customer Segment** column as the legend of the pie chart.



Notice that only one **Customer Segment** value exists. The reason is because the calculated column formula produces an incorrect result: Each customer is assigned the value of *High* because the

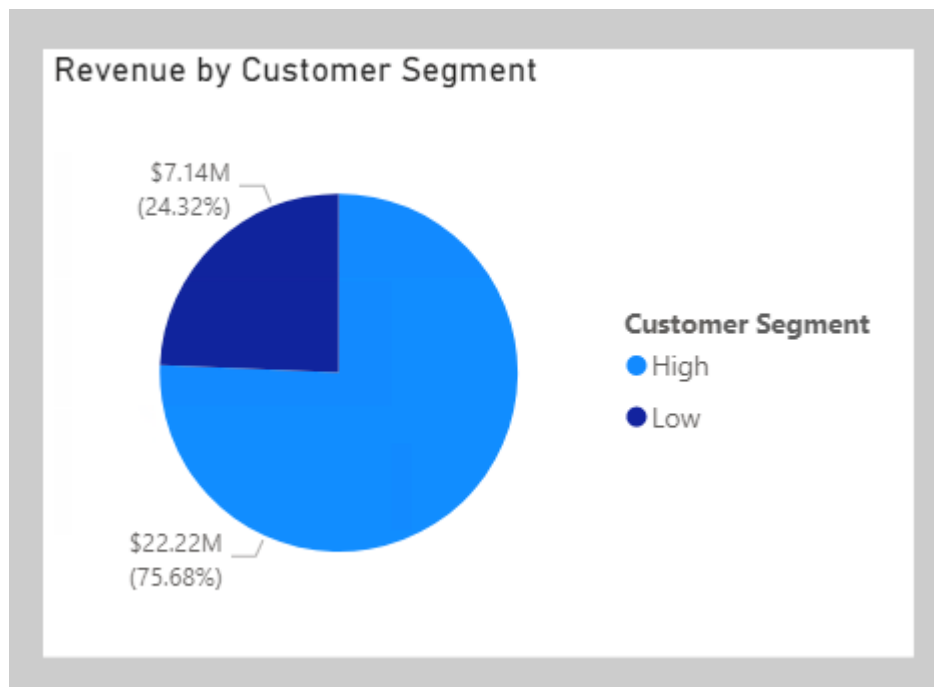
expression `SUM(Sales[Sales Amount])` isn't evaluated in a filter context. So, each customer is assessed on the sum of every `Sales Amount` column value in the `Sales` table.

To force the evaluation of the `SUM(Sales[Sales Amount])` expression *for each customer*, a context transition must take place that applies the row context column values to filter context. You can accomplish this transition by using the `CALCULATE` function without passing in filter expressions.

Modify the calculated column definition so that it produces the correct result.

```
Customer Segment =  
VAR CustomerRevenue = CALCULATE(SUM(Sales[Sales Amount]))  
RETURN  
    IF(CustomerRevenue < 2500, "Low", "High")
```

In the pie chart visual add the new calculated column to the Legend well, verify that two pie segments now display.



In this case, the `CALCULATE` function applies row context values as filters, known as *context transition*. To be accurate, the process doesn't quite work that way when a unique column is on the table. When a unique column is on the table, you only need to apply a filter on that column to make the transition happen. In this case, Power BI applies a filter on the `CustomerKey` column for the value in row context.

If you reference measures in an expression that's evaluated in row context, context transition is automatic. Thus, you don't need to pass measure references to the `CALCULATE` function.

Modify the calculated column definition, which references the `Revenue` measure, and notice that it continues to produce the correct result.

```
Customer Segment =
VAR CustomerRevenue = [Revenue]
RETURN
    IF(CustomerRevenue < 2500, "Low", "High")
```

Now, you can complete the **Sales Commission** measure formula. To produce a total, you need to use an iterator function to iterate over all regions in filter context. The iterator function expression must use the **CALCULATE** function to transition the row context to the filter context. Notice that it no longer needs to test whether a single **Country** column value in the **Sales Territory** table is in filter context because it's known to be filtering by a single country/region (because it's iterating over the regions in filter context and a region belongs to only one country/region).

Switch to **Page 3** of the report, and then modify the **Sales Commission** measure definition to use the **SUMX** iterator function:

```
Sales Commission =
SUMX(
    VALUES('Sales Territory'[Region]),
    CALCULATE(
        [Revenue]
        * IF(
            VALUES('Sales Territory'[Country]) = "United States",
            0.15,
            0.1
        )
    )
)
```

The table visual now displays a sales commission total for all regions.

Region	Revenue	Sales Commission
Australia	\$10,655,335.96	\$1,065,533.60
Canada	\$16,355,770.46	\$1,635,577.05
Central	\$7,909,009.01	\$1,186,351.35
France	\$7,251,555.65	\$725,155.56
Germany	\$4,878,300.38	\$487,830.04
Northeast	\$6,939,374.48	\$1,040,906.17
Northwest	\$16,084,942.55	\$2,412,741.38
Southeast	\$7,879,655.07	\$1,181,948.26
Southwest	\$24,184,609.60	\$3,627,691.44
United Kingdom	\$7,670,721.04	\$767,072.10
Total	\$109,809,274.20	\$14,130,806.96

8. Check your knowledge

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-modify-filter/6-check>

Check your knowledge

Completed

9. Summary

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-modify-filter/7-summary>

Summary

Completed

In this module, you learned about filter context. You also learned why and how to work with the `CALCULATE` function to modify filter context by passing in filters. These filters can add or overwrite the filter context, and they can modify filter context when you're passing in special functions like `REMOVEFILTERS` or `KEEPFILTERS`.

Additionally, you learned that the `CALCULATE` function can transition row context to filter context, which can be required when you're using measure expressions in calculated columns or iterator functions.

Use DAX time intelligence functions in semantic models

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-time-intelligence/1-introduction>

Introduction

Completed

Time intelligence relates to calculations over time. Specifically, it relates to calculations over dates, months, quarters, or years, and possibly time. Rarely would you need to calculate over time in the sense of hours, minutes, or seconds.

In Data Analysis Expressions (DAX) calculations, time intelligence refers to *modifying the filter context for date filters*.

For example, at the Adventure Works company, their financial year begins on July 1 and ends on June 30 of the following year. They produce a table visual that displays monthly revenue and year-to-date (YTD) revenue.

Year	Revenue	Revenue YTD
FY2018	\$23,860,891.17	\$23,860,891.17
2017 Jul	\$1,423,357.32	\$1,423,357.32
2017 Aug	\$2,057,902.45	\$3,481,259.78
2017 Sep	\$2,523,947.55	\$6,005,207.32
2017 Oct	\$561,681.48	\$6,566,888.80
2017 Nov	\$4,764,920.16	\$11,331,808.96
2017 Dec	\$596,746.56	\$11,928,555.52
2018 Jan	\$1,327,674.63	\$13,256,230.15
2018 Feb	\$3,936,463.31	\$17,192,693.45
2018 Mar	\$700,873.18	\$17,893,566.64
2018 Apr	\$1,519,275.24	\$19,412,841.88
2018 May	\$2,960,378.09	\$22,373,219.97
2018 Jun	\$1,487,671.19	\$23,860,891.17
FY2019	\$34,070,108.50	\$34,070,108.50
2018 Jul	\$2,939,691.00	\$2,939,691.00
2018 Aug	\$3,964,801.20	\$6,904,492.20
2018 Sep	\$3,287,605.93	\$10,192,098.13

The filter context for *2017 August* contains each of the 31 dates of August, which are stored in the **Date** table. However, the calculated year-to-date revenue for *2017 August* applies a different filter context. It's the first date of the year through to the last date in filter context. In this example, that's July 1, 2017 through to August 31, 2017.

Time intelligence calculations modify date filter contexts. They can help you answer these time-related questions:

- What's the accumulation of revenue for the year, quarter, or month?
- What revenue was produced for the same period last year?
- What growth in revenue has been achieved over the same period last year?
- How many new customers made their first order in each month?
- What's the inventory stock on-hand value for the company's products?

This module describes how to create time intelligence measures to answer these questions.

Note

Many units in this module present inline activities to follow along and create calculations. These activities are optional. If you want to complete the activities, you can download the Power BI Desktop file by using the links provided in the units.

At the end of this module, you have the opportunity to complete an exercise in a virtual machine to apply the lessons on how to create calculated tables, calculated columns, and simple measures using DAX.

2. Use DAX time intelligence functions

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-time-intelligence/2-functions>

Use DAX time intelligence functions

Completed

DAX includes several time intelligence functions to simplify the task of modifying date filter context. You can write many of these intelligence formulas by using a `CALCULATE` function that modifies date filters, but that creates more work.

Note

Many DAX time intelligence functions work with standard date periods such as years, quarters, and months. If you have irregular time periods, like financial months that do not match the calendar, or if you need to work with weeks or shorter time periods, the built-in DAX functions will not help. In these cases, use the `CALCULATE` function with custom date or time filters.

Date table requirement

To work with time intelligence DAX functions, you need to meet the prerequisite model requirement of having at least one *date table* in your model. A date table is a table that is marked as a Date table. That table must have a column of data type Date (or date/time), known as the *date column*. The date column must:

- Contain unique values.
- Span full years.
- Not contain BLANKs.
- Not have any missing dates.

For more information, see [Create date tables in Power BI Desktop](#).

Summarizations over time

One group of DAX time intelligence functions is concerned with summarizations over time:

- `DATESYTD` - Returns a single-column table that contains dates for the year-to-date (YTD) in the current filter context. This group also includes the `DATESMTD` and `DATESQTD` functions for month-to-date (MTD) and quarter-to-date (QTD). You can pass these functions as filters into the `CALCULATE` function.

- **TOTALYTD** - Evaluates an expression for YTD in the current filter context. The equivalent QTD and MTD DAX functions of **TOTALQTD** and **TOTALMTD** are also included.
- **DATESBETWEEN** - Returns a table that contains a column of dates that begins with a given start date and continues until a given end date.
- **DATESINPERIOD** - Returns a table that contains a column of dates that begins with a given start date and continues for the specified number of intervals.

Note

While the **TOTALYTD** function is simple to use, you're limited to passing in one filter expression. If you need to apply multiple filter expressions, use the **CALCULATE** function and then pass the **DATESYTD** function in as one of the filter expressions.

In the following example, you'll create your first time intelligence calculation that uses the **TOTALYTD** function. The syntax is as follows:

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

The function requires an expression and, as is common to all time intelligence functions, a reference to the date column of a marked date table. Optionally, a single filter expression or the year-end date can be passed in (required only when the year doesn't finish on December 31).

Download and open the [Adventure Works DW 2020 M07.pbix](#) file. Then, add the following measure definition to the **Sales** table that calculates YTD revenue. Format the measure as currency with two decimal places.

```
Revenue YTD =
TOTALYTD(
    [Revenue],
    'Date'[Date],
    "6-30"
)
```

The year-end date value of **"6-30"** represents June 30.

On **Page 1** of the report, add the **Revenue YTD** measure to the matrix visual. Notice that it produces a summarization of the revenue amounts from the beginning of the year through to the filtered month.

Year	Revenue	Revenue YTD
☒ FY2018	\$23,860,891.17	\$23,860,891.17
2017 Jul	\$1,423,357.32	\$1,423,357.32
2017 Aug	\$2,057,902.45	\$3,481,259.78
2017 Sep	\$2,523,947.55	\$6,005,207.32
2017 Oct	\$561,681.48	\$6,566,888.80
2017 Nov	\$4,764,920.16	\$11,331,808.96
2017 Dec	\$596,746.56	\$11,928,555.52
2018 Jan	\$1,327,674.63	\$13,256,230.15
2018 Feb	\$3,936,463.31	\$17,192,693.45
2018 Mar	\$700,873.18	\$17,893,566.64
2018 Apr	\$1,519,275.24	\$19,412,841.88
2018 May	\$2,960,378.09	\$22,373,219.97
2018 Jun	\$1,487,671.19	\$23,860,891.17

Comparisons over time

Another group of DAX time intelligence functions is concerned with shifting time periods:

- **DATEADD** - Returns a table that contains a column of dates, shifted either forward or backward in time by the specified number of intervals from the dates in the current filter context.
- **PARALLELPERIOD** - Returns a table that contains a column of dates that represents a period that is parallel to the dates in the specified dates column, in the current filter context, with the dates shifted a number of intervals either forward in time or back in time.
- **SAMEPERIODLASTYEAR** - Returns a table that contains a column of dates that are shifted one year back in time from the dates in the specified dates column, in the current filter context.
- Many helper DAX functions for navigating backward or forward for specific time periods, all of which returns a table of dates. These helper functions include **NEXTDAY**, **NEXTMONTH**, **NEXTQUARTER**, **NEXTYEAR**, and **PREVIOUSDAY**, **PREVIOUSMONTH**, **PREVIOUSQUARTER**, and **PREVIOUSYEAR**.

Now, you'll add a measure to the **Sales** table that calculates revenue for the prior year by using the **SAMEPERIODLASTYEAR** function. Format the measure as currency with two decimal places.

```
Revenue PY =
VAR RevenuePriorYear = CALCULATE(
    [Revenue],
    SAMEPERIODLASTYEAR('Date' [Date])
)
RETURN
    RevenuePriorYear
```

Add the **Revenue PY** measure to the matrix visual. Notice that it produces results that are similar to the previous year's revenue amounts.

Year	Revenue	Revenue YTD	Revenue PY
☒ FY2018	\$23,860,891.17	\$23,860,891.17	
2017 Jul	\$1,423,357.32	\$1,423,357.32	
2017 Aug	\$2,057,902.45	\$3,481,259.78	
2017 Sep	\$2,523,947.55	\$6,005,207.32	
2017 Oct	\$561,681.48	\$6,566,888.80	
2017 Nov	\$4,764,920.16	\$11,331,808.96	
2017 Dec	\$596,746.56	\$11,928,555.52	
2018 Jan	\$1,327,674.63	\$13,256,230.15	
2018 Feb	\$3,936,463.31	\$17,192,693.45	
2018 Mar	\$700,873.18	\$17,893,566.64	
2018 Apr	\$1,519,275.24	\$19,412,841.88	
2018 May	\$2,960,378.09	\$22,373,219.97	
2018 Jun	\$1,487,671.19	\$23,860,891.17	
☒ FY2019	\$34,070,108.50	\$34,070,108.50	\$23,860,891.17
2018 Jul	\$2,939,691.00	\$2,939,691.00	\$1,423,357.32
2018 Aug	\$3,964,801.20	\$6,904,492.20	\$2,057,902.45
2018 Sep	\$3,287,605.93	\$10,192,098.13	\$2,523,947.55
2018 Oct	\$2,157,287.40	\$12,349,385.53	\$561,681.48
2018 Nov	\$3,611,092.23	\$15,960,477.76	\$4,764,920.16
2018 Dec	\$2,624,078.39	\$18,584,556.15	\$596,746.56
2019 Jan	\$1,847,691.91	\$20,432,248.06	\$1,327,674.63
2019 Feb	\$2,829,361.64	\$23,261,609.70	\$3,936,463.31
2019 Mar	\$2,092,434.35	\$25,354,044.05	\$700,873.18
2019 Apr	\$2,405,970.99	\$27,760,015.05	\$1,519,275.24
2019 May	\$3,459,444.04	\$31,219,459.08	\$2,960,378.09
2019 Jun	\$2,850,649.42	\$34,070,108.50	\$1,487,671.19

Next, you'll modify the measure by renaming it to **Revenue YoY %** and then updating the **RETURN** clause to calculate the change ratio. Be sure to change the format to a percentage with two decimals places.

```

Revenue YoY % =
VAR RevenuePriorYear = CALCULATE(
    [Revenue],
    SAMEPERIODLASTYEAR('Date' [Date])
)
RETURN
    DIVIDE(
        [Revenue] - RevenuePriorYear,
        RevenuePriorYear
    )

```

Notice that the **Revenue YoY %** measure produces a ratio of change factor over the previous year's monthly revenue. For example, July 2018 represents a 106.53 percent *increase* over the previous

year's monthly revenue, and November 2018 represents a 24.22 percent *decrease* over the previous year's monthly revenue.

Year	Revenue	Revenue YTD	Revenue YoY %
☐ FY2018	\$23,860,891.17	\$23,860,891.17	
2017 Jul	\$1,423,357.32	\$1,423,357.32	
2017 Aug	\$2,057,902.45	\$3,481,259.78	
2017 Sep	\$2,523,947.55	\$6,005,207.32	
2017 Oct	\$561,681.48	\$6,566,888.80	
2017 Nov	\$4,764,920.16	\$11,331,808.96	
2017 Dec	\$596,746.56	\$11,928,555.52	
2018 Jan	\$1,327,674.63	\$13,256,230.15	
2018 Feb	\$3,936,463.31	\$17,192,693.45	
2018 Mar	\$700,873.18	\$17,893,566.64	
2018 Apr	\$1,519,275.24	\$19,412,841.88	
2018 May	\$2,960,378.09	\$22,373,219.97	
2018 Jun	\$1,487,671.19	\$23,860,891.17	
☐ FY2019	\$34,070,108.50	\$34,070,108.50	42.79%
2018 Jul	\$2,939,691.00	\$2,939,691.00	106.53%
2018 Aug	\$3,964,801.20	\$6,904,492.20	92.66%
2018 Sep	\$3,287,605.93	\$10,192,098.13	30.26%
2018 Oct	\$2,157,287.40	\$12,349,385.53	284.08%
2018 Nov	\$3,611,092.23	\$15,960,477.76	-24.22%
2018 Dec	\$2,624,078.39	\$18,584,556.15	339.73%
2019 Jan	\$1,847,691.91	\$20,432,248.06	39.17%
2019 Feb	\$2,829,361.64	\$23,261,609.70	-28.12%
2019 Mar	\$2,092,434.35	\$25,354,044.05	198.55%
2019 Apr	\$2,405,970.99	\$27,760,015.05	58.36%
2019 May	\$3,459,444.04	\$31,219,459.08	16.86%
2019 Jun	\$2,850,649.42	\$34,070,108.50	91.62%

Note

The **Revenue YoY %** measure demonstrates a good use of DAX variables. The measure improves the readability of the formula and allows you to unit test part of the measure logic (by returning the **RevenuePriorYear** variable value). Additionally, the measure is an optimal formula because it doesn't need to retrieve the prior year's revenue value twice. Having stored it once in a variable, the **RETURN** clause uses the variable value twice.

3. Exercise - Use time intelligence functions

Exercise - Use time intelligence functions

Completed

In this exercise, you learn how to:

- Create DAX measures using time intelligence functions.

This lab takes approximately **15** minutes to complete.

Note

A virtual machine containing the client tools you need is provided, along with the exercise instructions. Use the "*Launch lab*" button to launch the virtual machine.

A limited number of concurrent sessions are available. If the hosted environment is unavailable, please try again later.

Alternatively, you can [open the instructions in a separate window](#).

Access your environment

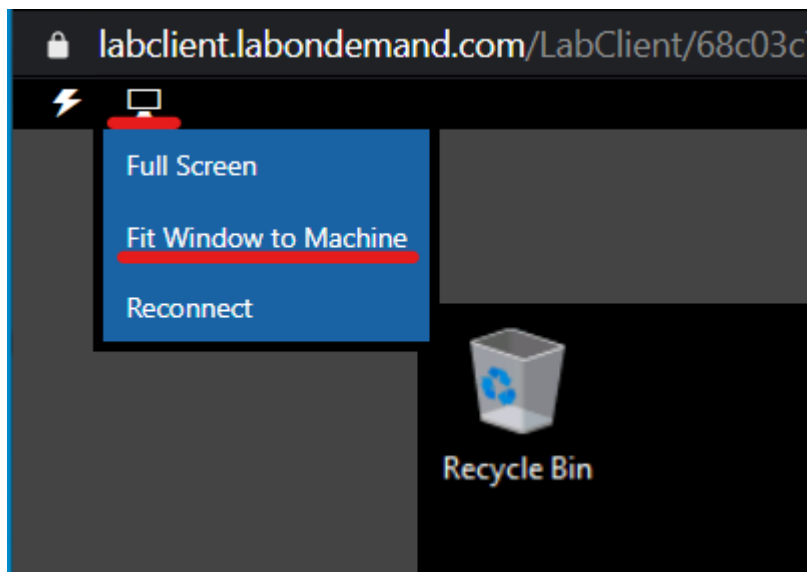
Before you start this lab (unless you are continuing from a previous lab), select **Launch lab** above.

You are automatically logged in to your lab environment as data-ai\student.

You can now begin your work on this lab.

Tip

To dock the lab environment so that it fills the window, select the PC icon at the top and then select **Fit Window to Machine**.



4. Additional time intelligence calculations

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-time-intelligence/3-calculations>

Additional time intelligence calculations

Completed

Other DAX time intelligence functions exist that are concerned with returning a single date. In this unit, you learn about these functions by applying them in two different scenarios.

The `FIRSTDATE` and the `LASTDATE` functions return the first and last date in the current filter context for the specified column of dates.

Calculate new occurrences

Another use of time intelligence functions is to count new occurrences. The following example shows how you can calculate the number of new customers for a time period. A new customer is counted in the time period in which they made their first purchase.

Your first task is to add the following measure to the `Sales` table that counts the number of distinct customers *life-to-date* (LTD). Life-to-date means from the beginning of time until the last date in filter context. Format the measure as a whole number by using the thousands separator.

```
Customers LTD =  
VAR CustomersLTD =  
    CALCULATEC
```

```

        DISTINCTCOUNT(Sales[CustomerKey]),
        DATESBETWEEN(
            'Date'[Date],
            BLANK(),
            MAX('Date'[Date])
        ),
        'Sales Order'[Channel] = "Internet"
    )
RETURN
    CustomersLTD

```

Add the `Customers LTD` measure to the matrix visual. Notice that it produces a result of distinct customers LTD until the end of each month.

Year	Revenue	Revenue YTD	Revenue YoY %	Customers LTD
FY2018	\$23,860,891.17	\$23,860,891.17		2,459
2017 Jul	\$1,423,357.32	\$1,423,357.32		289
2017 Aug	\$2,057,902.45	\$3,481,259.78		448
2017 Sep	\$2,523,947.55	\$6,005,207.32		609
2017 Oct	\$561,681.48	\$6,566,888.80		783
2017 Nov	\$4,764,920.16	\$11,331,808.96		1,013
2017 Dec	\$596,746.56	\$11,928,555.52		1,201
2018 Jan	\$1,327,674.63	\$13,256,230.15		1,394
2018 Feb	\$3,936,463.31	\$17,192,693.45		1,571
2018 Mar	\$700,873.18	\$17,893,566.64		1,790
2018 Apr	\$1,519,275.24	\$19,412,841.88		1,992
2018 May	\$2,960,378.09	\$22,373,219.97		2,214
2018 Jun	\$1,487,671.19	\$23,860,891.17		2,459

The `DATESBETWEEN` function returns a table that contains a column of dates that begins with a given start date and continues until a given end date. When the start date is `BLANK`, it uses the first date in the date column. (Conversely, when the end date is `BLANK`, it uses the last date in the date column.) In this case, the end date is determined by the `MAX` function, which returns the last date in filter context. Therefore, if the month of August 2017 is in filter context, then the `MAX` function will return August 31, 2017 and the `DATESBETWEEN` function will return all dates through to August 31, 2017.

Next, you modify the measure by renaming it to `New Customers` and by adding a second variable to store the count of distinct customers *before* the time period in filter context. The `RETURN` clause now subtracts this value from LTD customers to produce a result, which is the number of new customers in the time period.

```

New Customers =
VAR CustomersLTD =
    CALCULATE(
        DISTINCTCOUNT(Sales[CustomerKey]),
        DATESBETWEEN(
            'Date'[Date],
            BLANK(),

```



```

        MAX('Date'[Date])
    ),
    'Sales Order'[Channel] = "Internet"
)
VAR CustomersPrior =
    CALCULATE(
        DISTINCTCOUNT(Sales[CustomerKey]),
        DATESBETWEEN(
            'Date'[Date],
            BLANK(),
            MIN('Date'[Date]) - 1
        ),
        'Sales Order'[Channel] = "Internet"
    )
RETURN
    CustomersLTD - CustomersPrior

```

Year	Revenue	Revenue YTD	Revenue YoY %	New Customers
FY2018	\$23,860,891.17	\$23,860,891.17		2,459
2017 Jul	\$1,423,357.32	\$1,423,357.32		289
2017 Aug	\$2,057,902.45	\$3,481,259.78		159
2017 Sep	\$2,523,947.55	\$6,005,207.32		161
2017 Oct	\$561,681.48	\$6,566,888.80		174
2017 Nov	\$4,764,920.16	\$11,331,808.96		230
2017 Dec	\$596,746.56	\$11,928,555.52		188
2018 Jan	\$1,327,674.63	\$13,256,230.15		193
2018 Feb	\$3,936,463.31	\$17,192,693.45		177
2018 Mar	\$700,873.18	\$17,893,566.64		219
2018 Apr	\$1,519,275.24	\$19,412,841.88		202
2018 May	\$2,960,378.09	\$22,373,219.97		222
2018 Jun	\$1,487,671.19	\$23,860,891.17		245

For the `CustomersPrior` variable, notice that the `DATESBETWEEN` function includes dates until the first date in filter context *minus* one. Because Power BI internally stores dates as numbers, you can add or subtract numbers to shift a date.

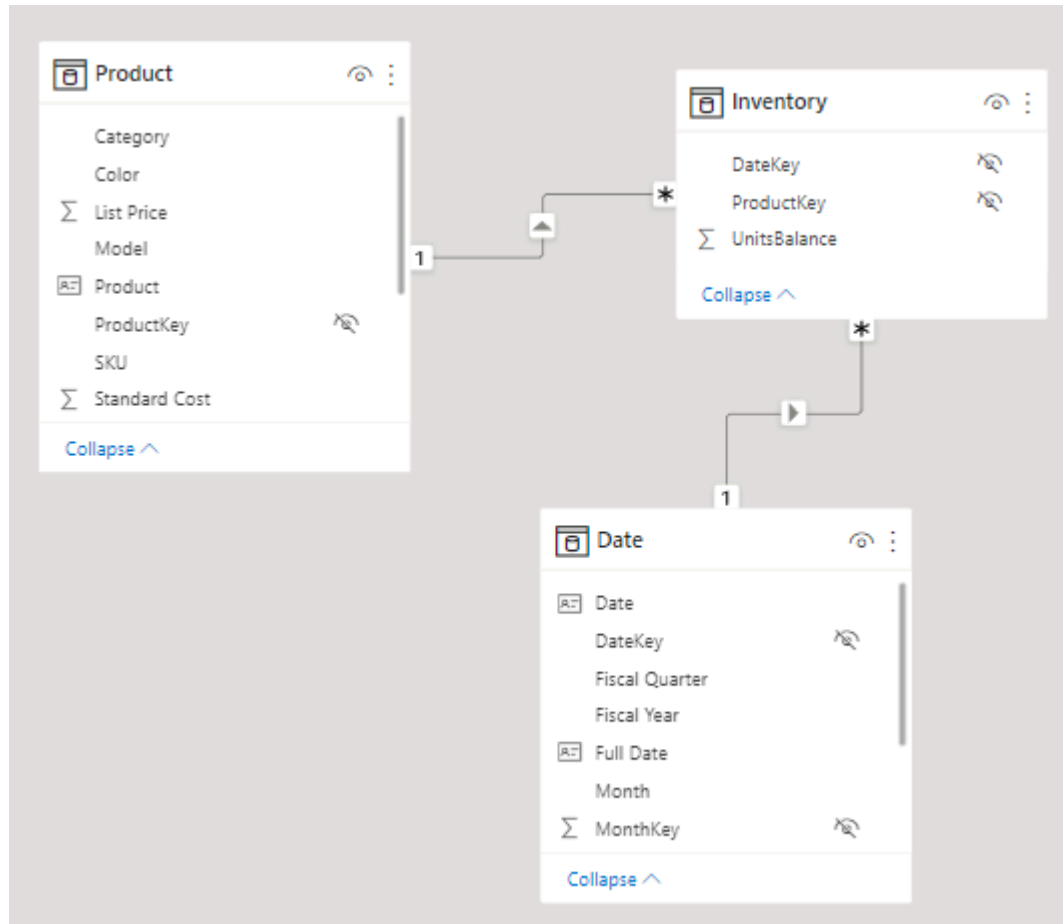
Snapshot calculations

Occasionally, fact data is stored as snapshots in time. Common examples include inventory stock levels or account balances. A snapshot of values is loaded into the table on a periodic basis.

When summarizing snapshot values (like inventory stock levels), you can summarize values across any dimension except date. Adding stock level counts across product categories produces a meaningful summary, but adding stock level counts across dates doesn't. Adding yesterday's stock level to today's stock level isn't a useful operation to perform (unless you want to average that result).

When you're summarizing snapshot tables, measure formulas can rely on DAX time intelligence functions to enforce a single date filter.

In the following example, you explore a scenario for the Adventure Works company. Switch to model view and select the **Inventory** model diagram.



Notice that the diagram shows three tables: **Product**, **Date**, and **Inventory**. The **Inventory** table stores snapshots of unit balances for each date and product. Importantly, the table contains no missing dates and no duplicate entries for any product on the same date. Also, the last snapshot record is stored for the date of June 15, 2020.

Now, switch to report view and select **Page 2** of the report. Add the **UnitsBalance** column of the **Inventory** table to the matrix visual. Its default summarization is set to sum values.

FY2020 Mountain-200 Bike Stock													
Product	2019 Jul	2019 Aug	2019 Sep	2019 Oct	2019 Nov	2019 Dec	2020 Jan	2020 Feb	2020 Mar	2020 Apr	2020 May	2020 Jun	Total
Mountain-200 Black, 38	4,884	4,649	4,784	4,873	4,605	4,943	5,208	4,872	5,208	5,040	5,208	2,520	56,794
Mountain-200 Black, 42	5,248	5,175	5,246	5,382	5,200	5,415	5,177	4,843	5,177	5,010	5,177	2,505	59,555
Mountain-200 Black, 46	5,523	5,617	5,287	5,419	5,273	5,663	5,890	5,510	5,890	5,700	5,890	2,850	64,512
Mountain-200 Silver, 38	5,412	5,190	5,114	5,245	5,220	5,406	5,890	5,510	5,890	5,700	5,890	2,850	63,317
Mountain-200 Silver, 42	5,252	5,262	5,043	5,266	5,072	5,208	5,022	4,698	5,022	4,860	5,022	2,430	58,157
Mountain-200 Silver, 46	5,004	5,019	4,960	5,012	4,847	5,231	5,053	4,727	5,053	4,890	5,053	2,445	57,294
Total	31,323	30,912	30,434	31,197	30,217	31,866	32,240	30,160	32,240	31,200	32,240	15,600	359,629

This visual configuration is an example of how not to summarize a snapshot value. Adding daily snapshot balances together doesn't produce a meaningful result. Therefore, remove the

`UnitsBalance` field from the matrix visual.

Now, you add a measure to the `Inventory` table that sums the `UnitsBalance` value *for a single date*. The date is the last date of each time period. It's achieved by using the `LASTDATE` function. Format the measure as a whole number with the thousands separator.

```
Stock on Hand =  
CALCULATE(  
    SUM(Inventory[UnitsBalance]),  
    LASTDATE('Date'[Date])  
)
```

Note

Notice that the measure formula uses the `SUM` function. An aggregate function must be used (measures don't allow direct references to columns), but given that only one row exists for each product for each date, the `SUM` function only operates over a single row.

Add the `Stock on Hand` measure to the matrix visual. The value for each product is now based on the last recorded units balance for each month.

FY2020 Mountain-200 Bike Stock													
Product	2019 Jul	2019 Aug	2019 Sep	2019 Oct	2019 Nov	2019 Dec	2020 Jan	2020 Feb	2020 Mar	2020 Apr	2020 May	2020 Jun	Total
Mountain-200 Black, 38	151	171	99	172	30	168	168	168	168	168	168		
Mountain-200 Black, 42	165	186	116	176	76	167	167	167	167	167	167		
Mountain-200 Black, 46	182	184	131	172	111	190	190	190	190	190	190		
Mountain-200 Silver, 38	171	173	129	190	85	190	190	190	190	190	190		
Mountain-200 Silver, 42	177	169	109	170	120	162	162	162	162	162	162		
Mountain-200 Silver, 46	181	158	126	178	88	163	163	163	163	163	163		
Total	1,027	1,041	710	1,058	510	1,040	1,040	1,040	1,040	1,040	1,040		

The measure returns BLANKs for June 2020 because no record exists for the last date in June. According to the data, it hasn't happened yet.

Filtering by the last date in filter context has inherent problems: A recorded date might not exist because it hasn't yet happened, or perhaps because stock balances aren't recorded on weekends.

Your next step is to adjust the measure formula to determine the last date *that has a non-BLANK result* and then filter by that date. You can achieve this task by using the `LASTNONBLANK` function.

Use the following measure definition to modify the `Stock on Hand` measure.

```
Stock on Hand =  
CALCULATE(  
    SUM(Inventory[UnitsBalance]),  
    LASTNONBLANK(  
        'Date'[Date],  
        CALCULATE(SUM(Inventory[UnitsBalance]))  
    )  
)
```

In the matrix visual, notice the values for June 2020 and the total (representing the entire year).

2020 Apr	2020 May	2020 Jun	Total
168	168	168	168
167	167	167	167
190	190	190	190
190	190	190	190
162	162	162	162
163	163	163	163
1,040	1,040	1,040	1,040

The `LASTNONBLANK` function is an iterator function. It returns the last date that produces a non-BLANK result. It achieves this result by iterating through all dates in filter context *in descending chronological order*. (Conversely, the `FIRSTNONBLANK` iterates in ascending chronological order.) For each date, it evaluates the passed in expression. When it encounters a non-BLANK result, the function returns the date. That date is then used to filter the `CALCULATE` function.

Note

The `LASTNONBLANK` function evaluates its expression in row context. The `CALCULATE` function must be used to transition the row context to filter context to correctly evaluate the expression.

You should now hide the `UnitsBalance` column in the `Inventory` table. It prevents report authors from inappropriately summarizing snapshot unit balances.

5. Check your knowledge

<https://learn.microsoft.com/en-us/training/modules/dax-power-bi-time-intelligence/4-check>

Check your knowledge

Completed

6. Summary

Summary

Completed

In this module, you learned how time intelligence calculations work by changing the filter context for date filters. You learned about several DAX time intelligence functions. These functions help you create calculations like year-to-date (YTD) and year-over-year (YoY).

You also learned about life-to-date (LTD) calculations. LTD calculations help you count new occurrences in your fact data. Additionally, you saw how to filter snapshot data to ensure that only a single snapshot value is returned.

Create visual calculations in Power BI Desktop

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/power-bi-visual-calculations/1-introduction>

Introduction

Completed

Visual calculations in Power BI are Data Analytics Expressions (DAX) calculations defined on a visual. Unlike measures, they aren't stored in the model, but on the visual you're working with. Visual calculations make it easier to create calculations that were previously hard to create, leading to simpler DAX, easier maintenance, and better performance.

For example, you're asked to add the following calculations to your report:

- A comparison of sales over years.
- A running sum of cost by category.
- A moving average of profit over time.

Visual calculations make these complex calculations easier by providing a user-friendly interface and real-time validation when applying the calculation on the visual.

In this module, you'll create visual calculations using templates and parameters, and be able to decide when to use visual calculations and when to use measures.

By the end, you'll be able to use visual calculations to provide insights in Power BI reports.

2. Understand visual calculations

Understand visual calculations

Completed

Visual calculations are a simple way to add calculations to your Power BI report. They're DAX calculations defined on a visual and are defined visually. Compared to measures, the DAX required is easier to write, understand, and maintain.

Visual calculations are defined on a visual, such as a table, bar chart, or a matrix. They provide easy to use functions for calculations that are commonly used in organizations.

While creating a visual calculation, you don't have to worry about the complexity of the model. You just work with what is on the visual, making it easier to understand and check your work.

A visual calculation can refer to any data in the visual including columns, measures, or other visual calculations. This ability removes the complexity of the semantic model and simplifies the process of adding calculations to your report. You can use visual calculations to complete common business calculations, such as running sums or moving averages.

Benefits of using a visual calculation

DAX in visual calculations works differently from other calculation options. The formula isn't stored in the model, but on the visual itself. This means they can only refer to what's on the visual. If you need to use something from the model, you must add it to the visual first. This approach simplifies things by avoiding the complexity of filter context and the model.

Visual calculations combine the straightforward context of calculated columns with the flexibility of on-demand calculations from measures. They work on aggregated data instead of detailed data, which often improves performance. When you can choose between a new measure or a visual calculation, the visual calculation usually performs better.

Since visual calculations are part of the visual, they can refer to the visual structure, offering more flexibility.

Note

While visual calculations are in preview, the limitations in this section are subject to change. Please see [Considerations and limitations](#) for the up-to-date information for visual calculations.

3. Create visual calculations

<https://learn.microsoft.com/en-us/training/modules/power-bi-visual-calculations/3-create-visual-calculations>

Create visual calculations

Completed

Creating a visual calculation is straightforward - select a visual and choose **New calculation**. The visual calculation window consists of three major sections, as shown from top to bottom in the following image:

- The **visual preview** which shows the visual you're working with.
- A **formula bar** where you can add visual calculations.
- The **visual matrix** which shows the data in the visual, and displays the results of visual calculations as you add them. Any styling or theming you apply to your visual isn't applied to the visual matrix.



To add a visual calculation, type the expression in the formula bar. For example, the following code calculates the profit by subtracting Total Product Cost from Sales Amount.

```
Profit = [Sales Amount] - [Total Product Cost]
```


The following image shows this Profit calculation added to a matrix visual that contains Sales Amount and Total Product Cost by Fiscal Year. Now you can see the individual fiscal years and a total for the three different values: Sales Amount, Total Product Cost, and Profit.

1 Profit = [Sales Amount]-[Total Product Cost]			
Fiscal Year	Sales Amount	Total Product Cost	Profit
FY2018	\$23,860,891.17	\$20,824,957.60	3,035,933.57
FY2019	\$34,070,108.50	\$30,362,108.34	3,708,000.16
FY2020	\$51,878,274.54	\$46,070,842.02	5,807,432.52
Total	\$109,809,274.20	\$97,257,907.95	12,551,366.25

By default, most visual calculations on a visual are evaluated row-by-row, like a calculated column. In the previous example, for each row of the visual matrix the current Sales Amount and Total Product Cost are subtracted, and the result is returned in the Profit column.

Although possible, there's no need to add an aggregation function like SUM as you would in a measure. In fact, it's better not to add such aggregates when they're not necessary, so you can more easily distinguish between measures and visual calculation expressions.

Hide fields from a visual

As you add visual calculations, they're shown in the list of fields on the visual and on the visual itself:

Y-axis

Fiscal Year

X
|
>

+Add data

X-axis

Sales Amount

☐
X
|
>

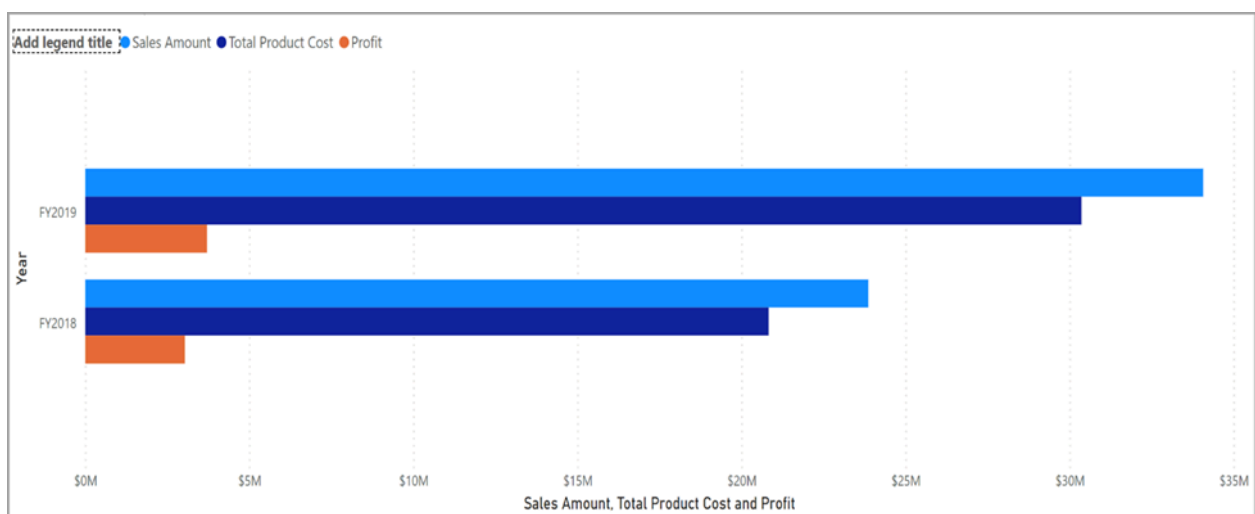
Total Product Cost

☐
X
|
>

☒
Profit

☐
X
|
>

+Add data



In visual calculations edit mode, you can hide fields from the visual just like you can hide columns and tables in the modeling view. For example, if you wanted to only show the *Profit* visual calculation, you can hide *Sales Amount* and *Total Profit* cost from view.



Hiding fields doesn't remove them from the visual or from the visual matrix, so your visual calculations can still refer to them and continue to work. A hidden field is still shown on the visual matrix but is not shown on the resulting visual. It's a recommended practice to only include hidden fields if they're necessary for your visual calculations to work.

Work with templates

Visual calculations include templates to make it easier to write common calculations. You can find templates by selecting the template button and choosing a template, such as:

- **Running sum** calculates the sum of values, adding the current value to the preceding values.
- **Moving average** calculates an average of a set of values in a given window by dividing the sum of the values by the size of the window.
- **Percent of parent** calculates the percentage of a value relative to its parent.
- **Average of children** calculates the average value of the set of child values.
- **Versus previous** compares a value to a preceding value.
- **Versus next** compares a value to a subsequent value.

Each template has a corresponding function which is added to the formula bar when you choose a template. You can also add your own expressions without relying on templates.

Available DAX functions

You can use many of the existing DAX functions in visual calculations. Since visual calculations work within the confines of the visual matrix, functions that rely on model relationships such as USERELATIONSHIP, RELATED, or RELATEDTABLE aren't available. Visual calculations also introduce a set of functions specific to visual calculations. Many of these functions are easier to use shortcuts to DAX window functions.

Note

For a full list of available functions, see the [documentation](#).

4. Use parameters in visual calculations

<https://learn.microsoft.com/en-us/training/modules/power-bi-visual-calculations/4-visual-calculation-parameters>

Use parameters in visual calculations

Completed

Visual calculations have optional parameters to help you create complex calculations with minimal code.

Use the Axis parameter

Many functions have an optional **Axis** parameter, which can only be used in visual calculations. Axis influences how the visual calculation traverses the visual matrix. The Axis parameter is set to the first axis in the visual by default. For many visuals the first axis is ROWS, which means that the visual calculation is evaluated row-by-row in the visual matrix, from top to bottom.

The following parameter values control how the data is calculated:

Value	Description	Icon
ROWS	Vertically across rows from top to bottom.	
COLUMNS	Horizontally across columns from left to right.	
ROWS COLUMNS	Vertically across rows from top to bottom, continuing column by column from left to right.	
COLUMNS ROWS	Horizontally across columns from left to right, continuing row by row from top to bottom.	

Use the Reset parameter

Many functions have an optional **Reset** parameter that is available in visual calculations only. Reset influences if and when the function resets its value to 0 or switches to a different scope while traversing the visual matrix.

The Reset parameter is set to None by default, which means the visual calculation is never restarted. The following list describes the only valid values for the Reset parameter:

- **NONE** is the default value and doesn't reset the calculation.
- **HIGHESTPARENT** resets the calculation when the value of the highest parent on the axis changes.
- **LOWESTPARENT** resets the calculations when the value of the lowest parent on the axis changes.
- **A numerical value** which tells Power BI which level in the visual's hierarchy to partition by—using positive integers to specify an absolute reset level from the top (1 = first field, 2 = second, etc.) and negative integers to specify a relative reset level above the current row—so the visual calculation knows where to restart its aggregation.

To better understand this concept, let's consider an axis that has three fields on multiple levels: Year, Quarter, and Month. As seen in the following example, **HIGHESTPARENT** is **Year** and **LOWESTPARENT** is **Quarter**, which affect how the running sum is calculated.

- `RUNNINGSUM([Sales Amount], HIGHESTPARENT)` starts from 0 **for every year**.
- `RUNNINGSUM([Sales Amount], LOWESTPARENT)` starts from 0 **for every Quarter**.

Lastly, a visual calculation that is defined as `RUNNINGSUM([Sales Amount])` doesn't reset, and continues adding the Sales Amount value for each month to the previous values without restarting.

Note

Reset expects there to be multiple levels on the axis. If there's only one level on the axis, you can use PARTITIONBY.

5. Exercise - Create visual calculations in Power BI Desktop

<https://learn.microsoft.com/en-us/training/modules/power-bi-visual-calculations/5-exercise>

Exercise - Create visual calculations in Power BI Desktop

Completed

In this exercise, you need to create visual calculations to develop a report for sales performance. You learn how to:

- Create a visual calculation
- Use the Axis parameter
- Use the Reset parameter

This lab takes approximately **30** minutes to complete.

Note

A virtual machine containing the client tools you need is provided, along with the exercise instructions. Use the "*Launch lab*" button to launch the virtual machine.

A limited number of concurrent sessions are available. If the hosted environment is unavailable, please try again later.

Alternatively, you can [open the instructions in a separate window](#).

Access your environment

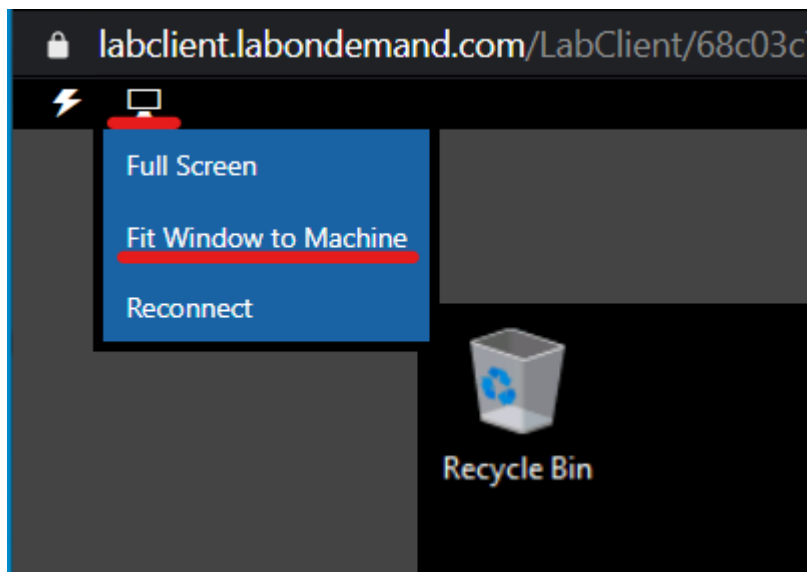
Before you start this lab (unless you are continuing from a previous lab), select **Launch lab** above.

You are automatically logged in to your lab environment as data-ai\student.

You can now begin your work on this lab.

Tip

To dock the lab environment so that it fills the window, select the PC icon at the top and then select **Fit Window to Machine**.



6. Module assessment

<https://learn.microsoft.com/en-us/training/modules/power-bi-visual-calculations/6-knowledge-check>

Module assessment

Completed

7. Summary

<https://learn.microsoft.com/en-us/training/modules/power-bi-visual-calculations/7-summary>

Summary

Completed

In this module, we explored how to use visual calculations in Power BI. Visual calculations let you add powerful calculations to your reports without needing to understand complex DAX language.

Some common problems solved with visual calculations include running sums, moving averages, and comparison calculations. Without visual calculations, users would need to rely on complex DAX functions, which can be time-consuming and difficult to manage.

Visual calculations improve report performance and flexibility, making it easier to create and manage calculations directly on the visual elements. This approach significantly reduces the time needed to add calculations to reports and enhances overall efficiency.

For a more in-depth comparison of ways of adding calculations in Power BI, see [Using calculations options in Power BI Desktop](#).

Optimize a model for performance in Power BI

1. Introduction to performance optimization

<https://learn.microsoft.com/en-us/training/modules/optimize-model-power-bi/1-introduction>

Introduction to performance optimization

Completed

Performance optimization, also known as performance tuning, involves making changes to the current state of the semantic model so that it runs more efficiently. Essentially, when your semantic model is optimized, it performs better.

In this module, you'll be introduced to the steps, processes, and concepts that are necessary to optimize a semantic model for enterprise-level performance. However, keep in mind that, while the basic performance and best practices guidance in Power BI will take you a long way, to optimize a semantic model for query performance, you'll likely have to partner with a data engineer to drive semantic model optimization in the source data systems.

By the end of this module, you'll be able to:

- Review the performance of measures, relationships, and visuals.
- Use variables to improve performance and troubleshooting.
- Improve performance by reducing column and relationship cardinality.
- Optimize DirectQuery models with table-level storage.
- Create and manage aggregations.

2. Describe semantic model optimization techniques

<https://learn.microsoft.com/en-us/training/modules/optimize-model-power-bi/1a-optimization-techniques>

Describe semantic model optimization techniques

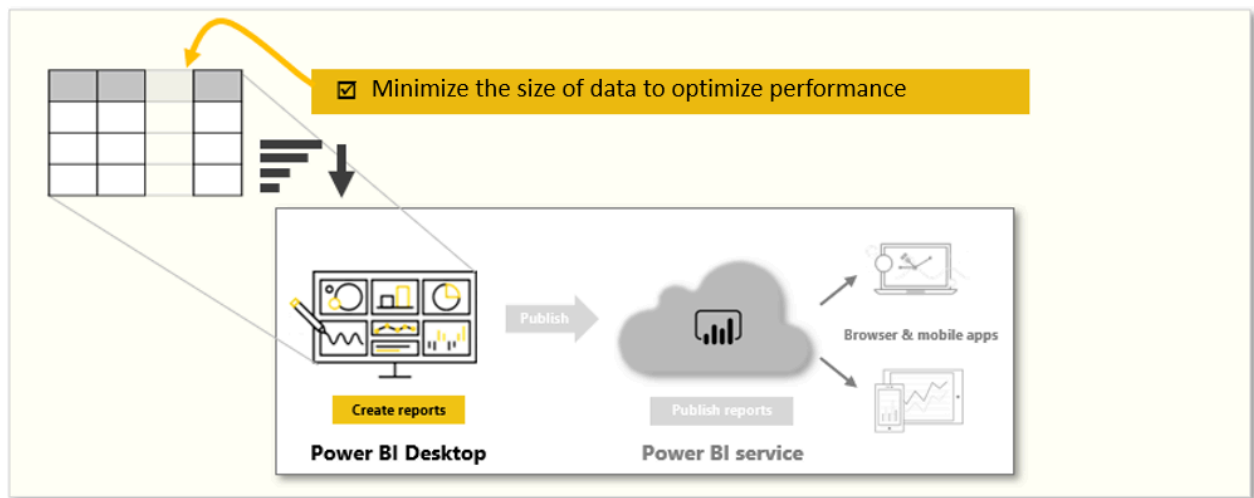
Completed

As a data analyst, you will spend approximately 90 percent of your time working with your data, and nine times out of ten, poor performance is a direct result of a poorly designed semantic model, inefficient Data Analysis Expressions (DAX) calculations, or a mix of the two. The process of designing a semantic model for performance can be tedious, and it's often underestimated.

However, if you address performance issues during development, you'll have a robust semantic model that will deliver better reporting performance and an overall more positive user experience. Ultimately, you'll also be able to maintain optimized performance. As your organization grows, the size of its data grows, and its semantic models becomes more complex. By optimizing your semantic model early, you can mitigate the negative impact that this growth might have on the performance of your semantic model.

A smaller sized semantic model uses less resources (memory) and achieves faster data refresh, calculations, and rendering of visuals in reports. Therefore, the performance optimization process involves minimizing the size of the semantic model and making the most efficient use of the data in the model. Your design decisions should:

- Ensure that the correct data types are used.
- Remove unnecessary columns and rows.
- Avoid repeated values.
- Surface numeric columns as measures.
- Reduce column cardinality.
- Analyze model metadata.
- Summarize data where possible.



For example, consider that you work as a Power BI developer for Tailwind Traders. You have been tasked to review a semantic model that was created a few years ago by another developer, a person who has since left the organization.

The semantic model produces a report that has received negative feedback from users. The users are happy with the results that they see in the report, but they aren't satisfied with the report performance. Loading the pages in the report takes too long, and tables aren't updating quickly enough when new filters are applied. In addition to this feedback, the IT team has highlighted that the file size of this particular semantic model is too large, and that it's putting a strain on the capacity resources.

You need to review the semantic model in order to identify the root cause(s) of the performance issues and make changes to optimize performance.

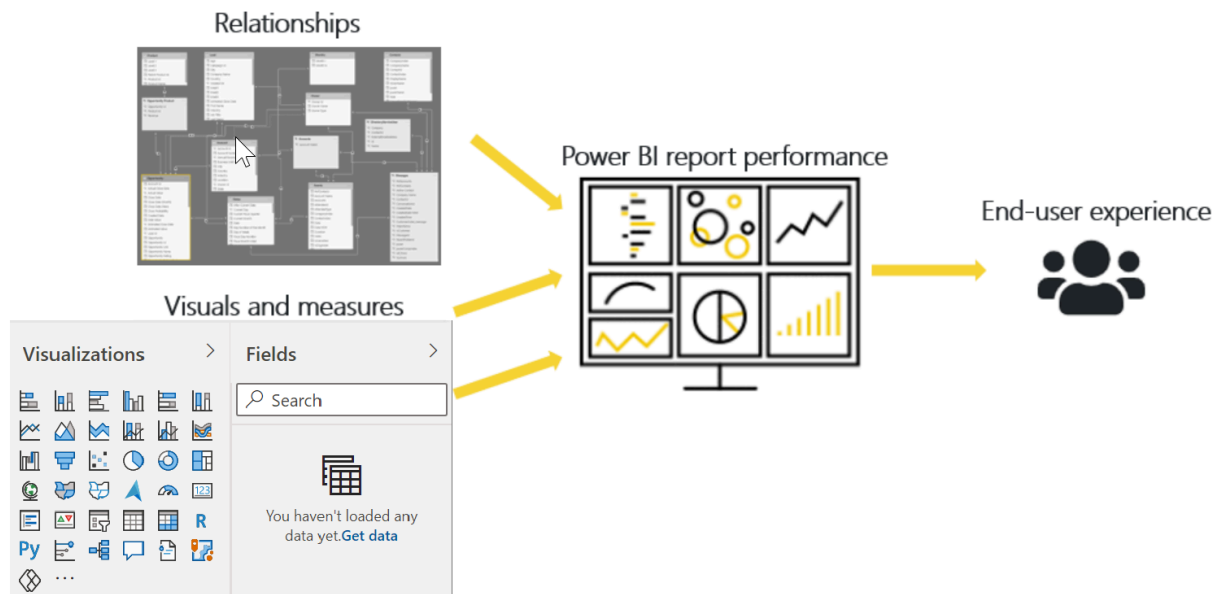
3. Review performance of measures, relationships, and visuals

<https://learn.microsoft.com/en-us/training/modules/optimize-model-power-bi/2-performance>

Review performance of measures, relationships, and visuals

Completed

If your semantic model has multiple tables, complex relationships, intricate calculations, multiple visuals, or redundant data, a potential exists for poor report performance. The poor performance of a report leads to a negative user experience.



To optimize performance, you must first identify where the source of the problem; in other words, find out which elements of your report and semantic model are causing the performance issues. Afterward, you can take action to resolve those issues and, therefore, improve performance.

Identify report performance bottlenecks

To achieve optimal performance in your reports, you need to create an efficient semantic model that supports fast-running queries and measures. When you have a good foundation, you can improve the model further by analyzing the query plans and dependencies and then making changes to further optimize performance.

You should review the measures and queries in your semantic model to ensure that you are using the most efficient way to get the results that you want. Your starting point should be to identify bottlenecks that exist in the code. When you identify the slowest query in the semantic model, you can focus on the biggest bottleneck first and then establish a priority list to work through the other issues.

Analyze performance

You can use **Performance Analyzer**, which is a tool in Power BI Desktop, to help find out how each of your report elements is performing when users interact with them. For example, you can determine how long it takes for a particular visual to refresh when it's initiated by a user interaction. The tool will help you identify any elements that contribute to your performance issues, which can be useful during troubleshooting.

Before you run **Performance Analyzer**, to ensure you get the most accurate results in your analysis (test), make sure that you start with a clear visual cache and a clear data engine cache.

- **Visual cache** - When you load a visual, you can't clear this visual cache without closing Power BI Desktop and opening it again. To avoid any caching in play, you need to start your analysis

with a clear visual cache.

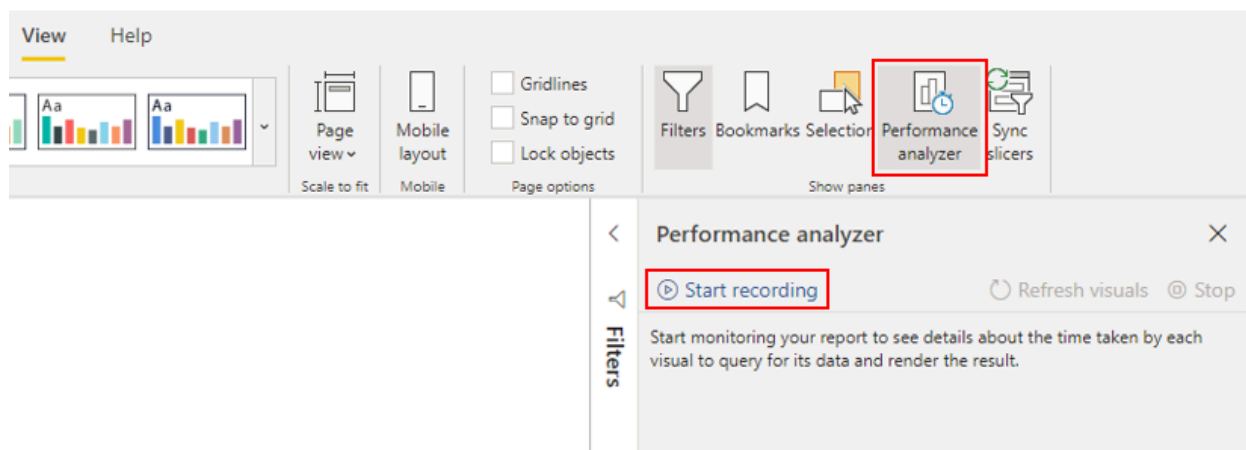
To ensure that you have a clear visual cache, add a blank page to your Power BI Desktop file and then, with that page selected, save and then close the file. Reopen the Power BI Desktop file, which will open on the blank page.

- **Data engine cache** - When a query is run, the results are cached, so the results of your analysis will be misleading. You need to clear the data cache before rerunning the visual.

To clear the data cache, you can either restart Power BI Desktop or connect DAX Studio to the semantic model and then invoke the **Clear Cache** function.

When you have cleared the caches and opened the Power BI Desktop file on the blank page, go to the **View** ribbon tab, and then select **Performance Analyzer**.

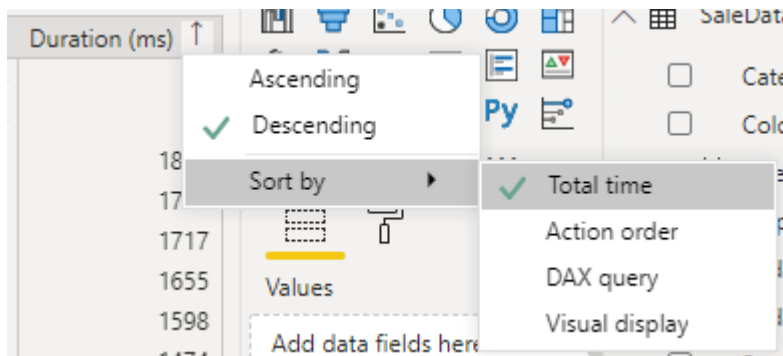
To begin the analysis process, select **Start recording**, select the page of the report that you want to analyze, and then interact with the elements of the report that you want to monitor. You will see the results of your interactions displayed in the **Performance Analyzer** pane as you work. When you are finished, select the **Stop** button.



For more detailed information, see [Use Performance Analyzer to examine report element performance](#).

Review results

You can review the results of your performance test in the **Performance Analyzer** pane. To review the tasks in order of duration, longest to shortest, right-click the **Sort** icon next to the **Duration (ms)** column header, and then select **Total time** in **Descending** order.



The log information for each visual shows how much time it took (duration) to complete the following categories of tasks:

- **DAX query** - The time it took for the visual to send the query, along with the time it took Power BI to return the results.
- **Visual display** - The time it took for the visual to render on the screen, including the time required to retrieve web images or geocoding.
- **Other** - The time it took the visual to prepare queries, wait for other visuals to complete, or perform other background processing tasks. If this category displays a long duration, the only real way to reduce this duration is to optimize DAX queries for other visuals, or reduce the number of visuals in the report.

Performance analyzer	
Start recording Refresh visuals Stop	
Clear Export	
Name	Duration (ms) ↑
Recording started (8/06/2020 10:59:48 a.m.)	-
Changed page	-
Card	1913
DAX query	94
Visual display	21
Other	1797
Copy query	
Slicer	1855
Sales by Quarter and Region	1770
Sales by Region and Color	1669
Card	1499
Card	1431
Slicer	1356
Sales by Region	1329
Sales by Quarter	1270

The results help you to understand the behavior of your semantic model and identify the elements that you need to optimize. You can compare the duration of each element in the report and identify

the elements that have a long duration. You should focus on those elements and investigate why it takes them so long to load on the report page.

To analyze your queries in more detail, you can use DAX Studio, which is a free, open-source tool that's provided by another service.

Resolve issues and optimize performance

The results of your analysis will identify areas for improvement and opportunities for performance optimization. You might find that you need to carry out improvements to the visuals, DAX calculations, or other elements in your semantic model. The following information provides guidance on what to look for and the changes that you can make.

Visuals

If you identify visuals as the bottleneck leading to poor performance, you should find a way to improve performance with minimal impact to user experience.

Consider the number of visuals on the report page; fewer visuals means better performance. Ask yourself if a visual is really necessary and whether it adds value to the end user. If the answer is no, you should remove that visual. Rather than using multiple visuals on the page, consider other ways to provide additional details, such as drill-through pages or report page tooltips.

Examine the number of fields in each visual. The more visuals you have on the report, the higher chance for performance issues. In addition, the more visuals, the more the report can appear crowded and lose clarity. The upper limit for visuals is 100 fields (measures or columns), so a visual with more than 100 fields will be slow to load. Ask yourself whether you really need all of this data in a visual. You might find that you can reduce the number of fields that you currently use.

DAX query

When you examine the results in the **Performance Analyzer** pane, you can see how long it took the Power BI Desktop engine to evaluate each query (in milliseconds). A good starting point is any DAX query that's taking longer than 120 milliseconds. In this example, you identify one particular query that has a large duration time.



☐ Sales by Year	270
DAX query	2754
Visual display	57
Other	160
📄 Copy query	

Performance Analyzer highlights potential issues, but it doesn't tell you what needs to be done to improve them. You might want to conduct further investigation into why this measure takes so long to process. You can use DAX Studio to investigate your queries in more detail.

For example, select **Copy Query** to copy the calculation formula onto the clipboard, then paste it into DAX Studio. You can then review the calculation step in more detail. In this example, you're trying to count the total number of products with order quantities greater than or equal to five.

```
Count Customers =  
CALCULATE(  
    DISTINCTCOUNT(Order[ProductID]),  
    FILTER (Order, Order[OrderQty] >= 5)  
)
```

After analyzing the query, you can use your own knowledge and experience to identify where the performance issues are. You can also try using different DAX functions to see whether they improve performance. In the following example, the `FILTER` function was replaced with the `KEEPFILTERS` function. When the test was run again in **Performance Analyzer**, the duration was shorter as a result of the replaced function.

```
Count Customers =  
CALCULATE(  
    DISTINCTCOUNT(Order[ProductID]),  
    KEEPFILTERS(Order[OrderQty] >= 5)  
)
```

In this case, you can replace the `FILTER` function with the `KEEPFILTERS` function to significantly reduce the evaluation duration time for this query. When you make this change, to check whether the duration time has improved or not, clear the data cache and then repeat the **Performance Analyzer** process.

☐ Sales by Year	270
DAX query	54
Visual display	57
Other	160
📄 Copy query	

Semantic model

If the duration of measures and visuals display low values (in other words they have a short duration time), they're not the reason for the performance issues. Instead, if the DAX query displays a high duration value, it's likely that a measure is written poorly or an issue has occurred with the semantic model. The issue might be caused by the relationships, columns, or metadata in your model, or it could be that the **Auto date/time** option is enabled, as described in the following section.

Relationships

You should review the model relationships between tables to ensure that you have established the correct relationships. Check that relationship cardinality properties are correctly set up. For example, a one-side column that contains unique values might be incorrectly configured as a many-side column. You'll learn more about how cardinality affects performance later in this module.

Columns

It's best practice to not import columns of data that you don't need. To avoid deleting columns in Power Query, you should try to deal with them at the source when loading data into Power BI Desktop. However, if it's impossible to remove redundant columns from the source query or the data has already been imported in its raw state, you can always use Power Query to examine each column. Ask yourself whether you really need each column and try to identify the benefit that each adds to your semantic model. If you find that a column adds no value, you should remove it from your semantic model. For example, suppose that you have an ID column with thousands of unique rows. You know that you won't use this particular column in a relationship, so it won't be used in a report. Therefore, you should conclude that this column is unnecessary and admit that it's wasting space in your semantic model.

When you remove an unnecessary column, you reduce the size of the semantic model which, in turn, results in a smaller file size and faster refresh time. Also, because the semantic model contains only relevant data, the overall report performance should improve.

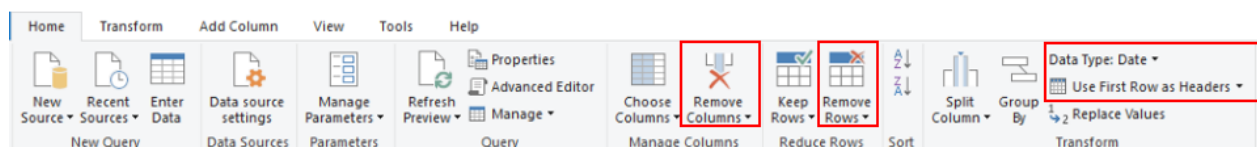
For more information, see [Data reduction techniques for Import modeling](#).

Metadata

Metadata is information about other data. Power BI metadata contains information on your semantic model, such as the name, data type and format of each of the columns, the schema of the database, the report design, when the file was last modified, the data refresh rates, and more.

When you load data into Power BI Desktop, it's good practice to analyze the corresponding metadata so you can identify any inconsistencies with your semantic model and normalize the data before you start to build reports. Running analysis on your metadata should improve semantic model performance because, while analyzing your metadata, you'll possibly identify unnecessary columns, errors within your data, incorrect data types, the volume of data being loaded (large semantic models, including transactional or historic data, will take longer to load), and more.

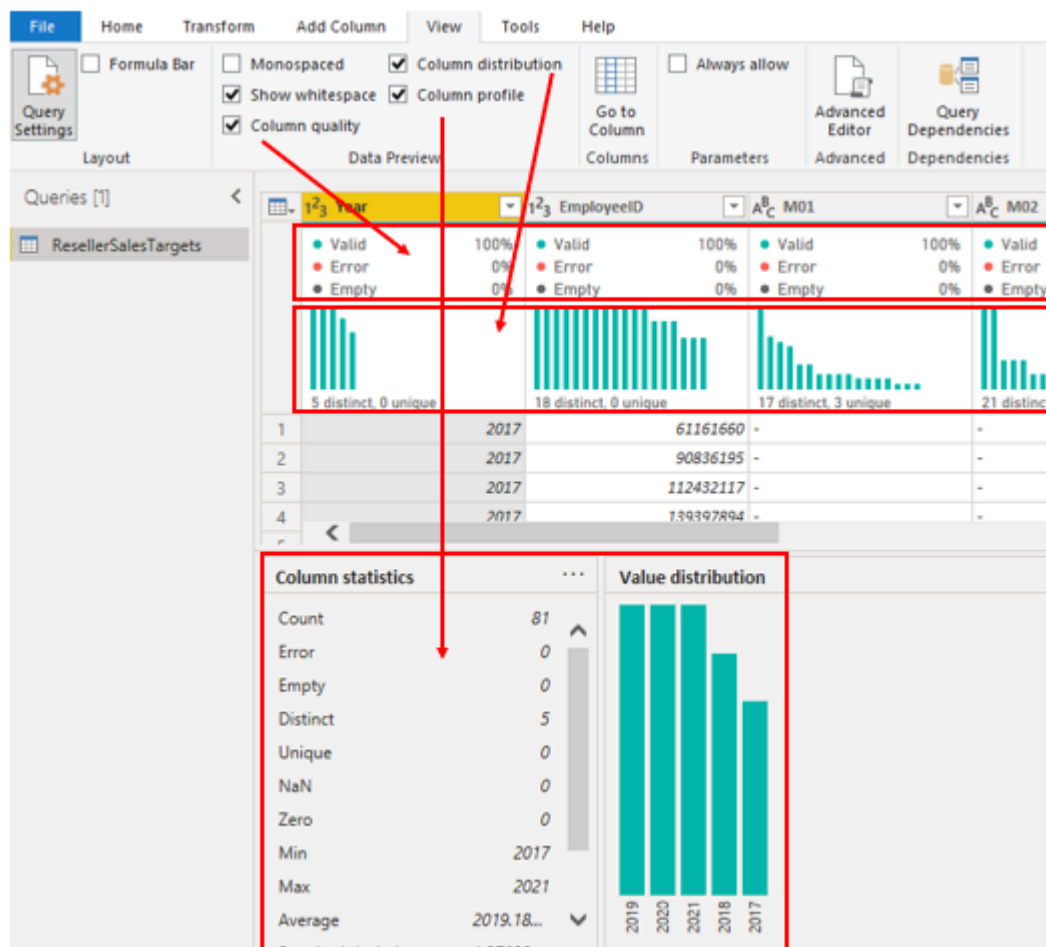
You can use Power Query in Power BI Desktop to examine the columns, rows, and values of your source data. You can then use the available tools, such as those highlighted in the following image, to make the necessary changes.



The Power Query options include:

- **Unnecessary columns** - Evaluates the need for each column. If one or more columns won't be used in the report and are therefore unnecessary, you should remove them by using the **Remove Columns** option.

- **Unnecessary rows** - Checks the first few rows in the semantic model to see whether they're empty or whether they contain data that you don't need in your reports; if so, it removes those rows by using the **Remove Rows** option.
- **Data type** - Evaluates the column data types to ensure that each one is correct. If you identify a data type that's incorrect, change it by selecting the column, selecting **Data Type** on the **Transform** ribbon tab, and then selecting the correct data type from the list.
- **Query names** - Examines the query (table) names in the **Queries** pane. Just like you did for column header names, you should change uncommon or unhelpful query names to names that are more obvious or names that the user is more familiar with. You can rename a query by right-clicking that query, selecting **Rename**, editing the name as required, and then pressing **Enter**.
- **Column details** - Power Query has the following three data preview options that you can use to analyze the metadata that's associated with your columns. You can find these options on the **View** ribbon tab, as shown in the following screenshot.
 - **Column quality** - Determines what percentage of items in the column are valid, have errors, or are empty. If the valid percentage is not 100 percent, you should investigate the reason, correct the errors, and populate empty values.
 - **Column distribution** - Displays frequency and distribution of the values in each of the columns. You'll investigate this further later in this module.
 - **Column profile** - Shows column statistics chart and a column distribution chart.



Note

If you're reviewing a query with more than 1,000 rows, and you want to analyze that whole semantic model, you need to change the default option at the bottom of the window. Select **Column profiling based on top 1000 rows** > **Column profiling based on entire data set**.



Other metadata that you should consider is the information about the semantic model as a whole, such as the file size and data refresh rates. You can find this metadata in the associated Power BI Desktop (.pbix) file. The data that you load into Power BI Desktop is compressed and stored to disk by the VertiPaq storage engine. The size of your semantic model has a direct impact on its performance; a smaller sized semantic model uses less resources (memory) and achieves faster data refresh, calculations, and rendering of visuals in reports.

Auto date/time feature

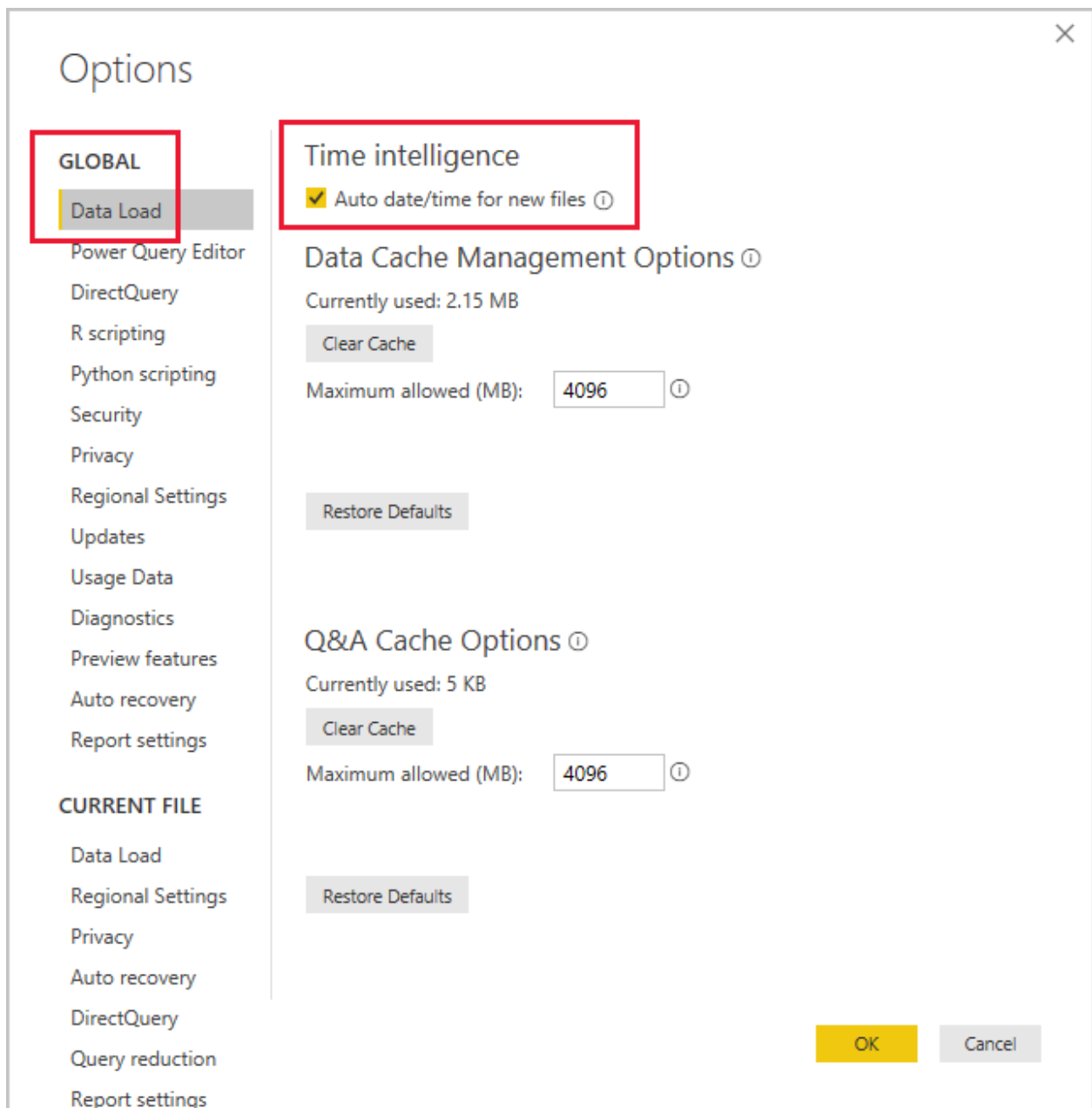
Another item to consider when optimizing performance is the **Auto date/time** option in Power BI Desktop. By default, this feature is enabled globally, which means that Power BI Desktop automatically creates a hidden calculated table for each date column, provided that certain conditions are met. The new, hidden tables are in addition to the tables that you already have in your semantic model.

The **Auto date/time** option allows you to work with time intelligence when filtering, grouping, and drilling down through calendar time periods. We recommend that you keep the **Auto date/time** option enabled only when you work with calendar time periods and when you have simplistic model requirements in relation to time.

If your data source already defines a date dimension table, that table should be used to consistently define time within your organization, and you should disable the global **Auto date/time** option. Disabling this option can lower the size of your semantic model and reduce the refresh time.

You can enable/disable the **Auto date/time** option globally so that it applies to all of your Power BI Desktop files, or you can enable/disable the option for the current file so that it applies only to an individual file.

To enable/disable the **Auto date/time** option, go to **File > Options and settings > Options**, and then select either the **Global** or **Current File** page. On either page, select **Data Load** and then, in the **Time Intelligence** section, select or clear the check box as required.



For an overview and general introduction to the **Auto date/time** feature, see [Apply auto date/time in Power BI Desktop](#).

4. Use variables to improve performance and troubleshooting

<https://learn.microsoft.com/en-us/training/modules/optimize-model-power-bi/3-variables>

Use variables to improve performance and

troubleshooting

Completed

You can use variables in your DAX formulas to help you write less complex and more efficient calculations. Variables are underused by developers who are starting out in Power BI Desktop, but they're effective and you should use them by whenever you can.

Some expressions involve the use of many nested functions and the reuse of expression logic. These expressions take a longer time to process and are difficult to read and, therefore, troubleshoot. If you use variables, you can save query processing time. This change is a step in the right direction toward optimizing the performance of a semantic model.

The use of variables in your semantic model provides the following advantages:

- **Improved performance** - Variables can make measures more efficient because they remove the need for Power BI to evaluate the same expression multiple times. You can achieve the same results in a query in about half the original processing time.
- **Improved readability** - Variables have short, self-describing names and are used in place of an ambiguous, multi-worded expression. You might find it easier to read and understand the formulas when variables are used.
- **Simplified debugging** - You can use variables to debug a formula and test expressions, which can be helpful during troubleshooting.
- **Reduced complexity** - Variables don't require the use of `EARLIER` or `EARLIEST` functions, which are difficult to understand. These functions were required before variables were introduced, and were written in complex expressions that introduced new filter contexts. Now that you can use variables instead of those functions, you can write fewer complex formulas.

Use variables to improve performance

To illustrate how you can use a variable to make a measure more efficient, the following table displays a measure definition in two different ways. Notice that the formula repeats the expression that calculates "same period last year" but in two different ways: the first instance uses the normal DAX calculation method and the second one uses variables in the calculation.

The second row of the table shows the improved measure definition. This definition uses the `VAR` keyword to introduce a variable named `SalesPriorYear`, and it uses an expression to assign the "same period last year" result to that new variable. It then uses the variable twice in the `DIVIDE` function.

Without variable

```
Sales YoY Growth =  
DIVIDE(  
    ([Sales] - CALCULATE([Sales], PARALLELPERIOD('Date'[Date], -12, MONTH))),
```

```
CALCULATE([Sales], PARALLELPERIOD('Date'[Date], -12, MONTH))
)
```

With variable

```
Sales YoY Growth =
VAR SalesPriorYear =
    CALCULATE([Sales], PARALLELPERIOD('Date'[Date], -12, MONTH))
VAR SalesVariance =
    DIVIDE(([Sales] - SalesPriorYear), SalesPriorYear)
RETURN
    SalesVariance
```

In the first measure definition, the formula is inefficient because it requires Power BI to evaluate the same expression twice. The second definition is more efficient because, thanks to the variable, Power BI only needs to evaluate the `PARALLELPERIOD` function once.

If your semantic model has multiple queries with multiple measures, the use of variables could reduce the overall query processing time in half and improve the overall performance of the semantic model. Furthermore, this solution is a simple one; imagine the savings as the formulas get more complicated, for instance, when you are working with percentages and running totals.

Use variables to improve readability

In addition to improved performance, you might notice how the use of variables makes your code simpler to read.

When using variables, it's best practice to use descriptive names for the variables. In the previous example, the variable is named `SalesPriorYear`, which clearly states what the variable is calculating. Consider the outcome of using a variable that was named `X`, `temp` or `variable1`; the purpose of the variable would not be clear at all.

Using clear, concise, meaningful names will help make it easier for you to understand and document what you're calculating, and it's much simpler for other developers to maintain in the future.

Use variables to troubleshoot multiple steps

You can use variables to help you debug a formula and identify what the issue is. Variables help simplify the task of troubleshooting your DAX calculation by evaluating each variable separately and by recalling them in the `RETURN` clause.

In the following example, you test an expression that's assigned to a variable. In order to debug you temporarily rewrite the `RETURN` clause to return the variable. The measure definition returns only the `SalesPriorYear` variable because that's what comes after the `RETURN` expression.

```
Sales YoY Growth % =
VAR SalesPriorYear = CALCULATE([Sales], PARALLELPERIOD('Date'[Date], -12, MONTH))
```

```
VAR SalesPriorYear% = DIVIDE([Sales] - SalesPriorYear, SalesPriorYear)
RETURN SalesPriorYear%
```

The `RETURN` clause returns only the `SalesPriorYear%` variable. This technique allows you to revert the expression when you have completed the debugging. It also makes calculations simpler to understand due to reduced complexity of the DAX code.

5. Reduce cardinality

<https://learn.microsoft.com/en-us/training/modules/optimize-model-power-bi/4-reduce-cardinality>

Reduce cardinality

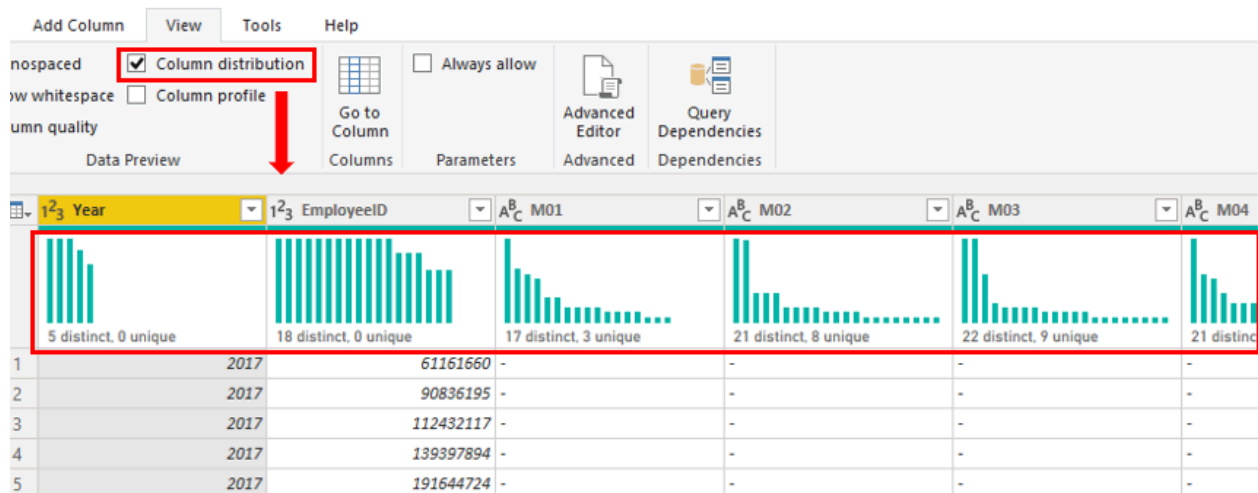
Completed

Cardinality is a term that's used to describe the uniqueness of the values in a column. Cardinality is also used in the context of model relationships, where it describes the direction of the relationship.

Identify cardinality levels in columns

Previously, when you used Power Query to analyze the metadata, the **Column distribution** option on the **View** ribbon tab displayed statistics on how many distinct and unique items were in each column in the data.

- **Distinct values count** - The total number of different values found in a given column.
- **Unique values count** - The total number of values that only appear once in a given column.



A column that has many repeated values in its range (unique count is low) has a low level of cardinality. Conversely, a column that has many unique values in its range (unique count is high) has a high level of cardinality.

Lower cardinality leads to more optimized performance, so you should try to reduce the number of high cardinality columns in your semantic model.

Reduce relationship cardinality

When you import multiple tables, it's possible that you'll do some analysis by using data from all those tables. Relationships between those tables are necessary to accurately calculate results and display the correct information in your reports. Power BI Desktop helps make creating those relationships easier. In fact, in most cases, you won't need to do anything because the autodetect feature can do it for you. However, you might occasionally need to create relationships or make changes to a relationship. Regardless, it's important to understand relationships in Power BI Desktop and how to create and edit them.

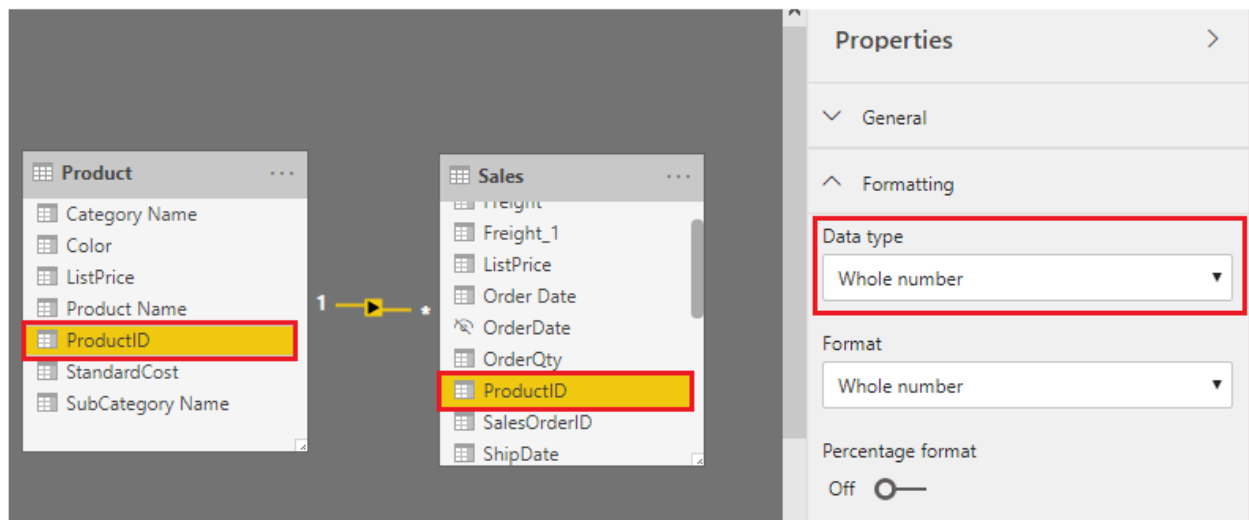
When you create or edit a relationship, you can configure other options. By default, Power BI Desktop automatically configures other options based on its assessment of the model data, which can be different for each relationship based on the data in the columns.

The relationships can have different cardinality. Cardinality is the direction of the relationship, and each model relationship must be defined with a cardinality type. The cardinality options in Power BI are:

- **Many-to-one (*:1)** - This relationship is the most common. It means that the column in one table can have more than one instance of a value, and the other related table, often known as the lookup table, has only one instance of a value.
- **One-to-one (1:1)** - The column in one table has only one instance of a particular value, and the other related table has only one instance of a particular value.
- **One-to-many (1:*)** - The column in one table has only one instance of a particular value, and the other related table can have more than one instance of a value.
- **Many-to-many (*:*)** - With composite models, you can establish a many-to-many relationship between tables, which removes requirements for unique values in tables. It also removes previous workarounds, such as introducing new tables only to establish relationships.

During development, you create and edit relationships in your model, so when you're building new relationships in your model, regardless of what cardinality you have chosen, always ensure that both of the columns that you are using to participate in a relationship have the same data type. Your model will never work if you try to build a relationship between two columns, where one column has a text data type and another column has an integer data type.

In the following example, the `ProductID` column has the **Whole number** data type in both the `Product` and `Sales` tables. The columns with data type **Integer** perform better than columns with data type **Text**.



Improve performance by reducing cardinality levels

Power BI Desktop offers different techniques that you can use to help reduce the data that's loaded into semantic models, such as summarization. Reducing the data that's loaded into your model improves the relationship cardinality of the report. For this reason, it's important that you strive to minimize the data that's loaded into your models. That's especially true for large models, or models that you anticipate will grow to become large over time.

Perhaps the most effective technique to reduce a model size is to use a summary table from the data source. Where a detail table might contain every transaction, a summary table might contain one record per day, per week, or per month. It might be a sum of all of the transaction amounts per day, for instance.

For example, a source sales fact table stores one row for each order line. Significant data reduction could be achieved by summarizing all sales metrics when you group by date, customer, and product, and individual transaction detail isn't needed.

Consider, then, that you can achieve an even more significant data reduction by summarizing at month level. It could achieve a possible 99 percent reduction in model size; but, reporting at day level or an individual order level is no longer possible. Deciding to summarize fact data always involves a tradeoff with the detail of your data. A disadvantage is that you might lose the ability to drill into data because the detail no longer exists. This tradeoff could be mitigated by using a Composite model.

In Power BI Desktop, a Composite model allows you to determine a storage mode for each table. Therefore, each table can have its **Storage Mode** property set as **Import** or **DirectQuery**.

An effective technique to reduce the model size is to set the **Storage Mode** property for larger fact tables to **DirectQuery**. This design approach can work well in conjunction with techniques that are used to summarize your data. For example, the summarized sales data could be used to achieve high performance "summary" reporting. You could then create a drillthrough page to display granular sales for a specific (and narrow) filter context, displaying all in-context sales orders. The drillthrough

page could include visuals based on a DirectQuery table to retrieve the sales order data (sales order details).

For more information, see [Data reduction techniques for Import modeling](#).

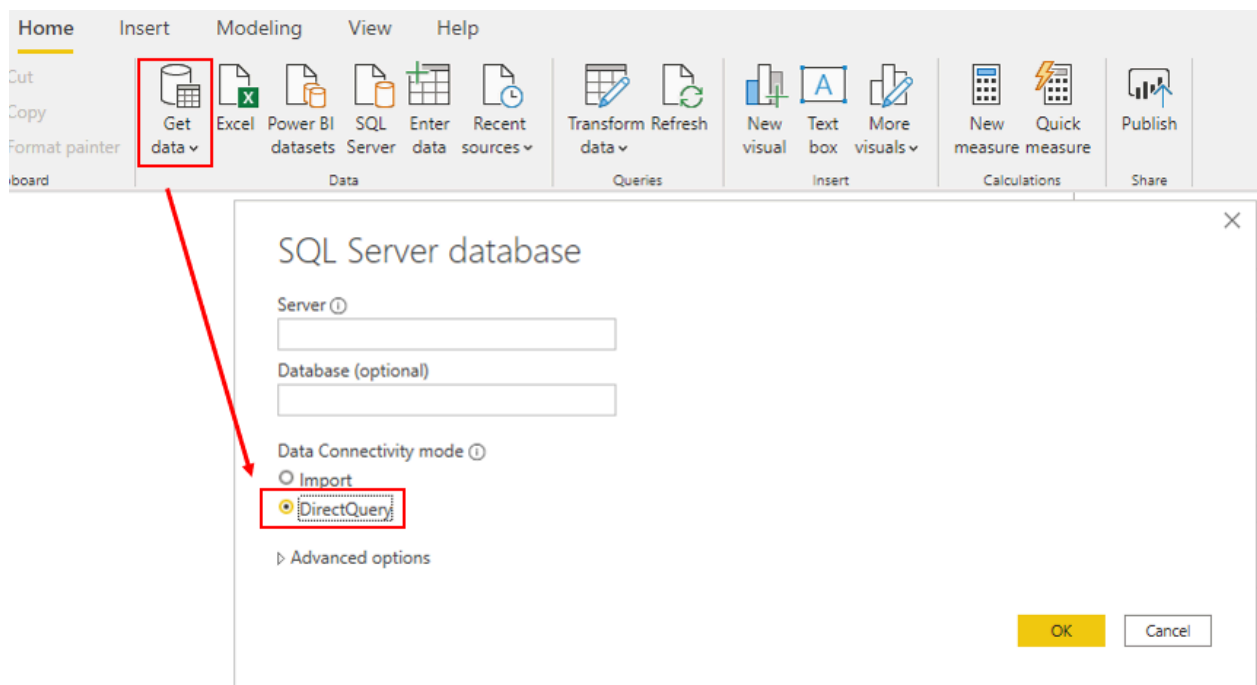
6. Optimize DirectQuery models with table level storage

<https://learn.microsoft.com/en-us/training/modules/optimize-model-power-bi/5-directquery-models>

Optimize DirectQuery models with table level storage

Completed

DirectQuery is a storage mode where Power BI connects directly to the data source. It is an alternative to importing data into model tables.



When you use the DirectQuery storage mode, query response times depend heavily on the performance of the underlying data source. Slow query response times lead to a negative user experience and, in the worst-case scenarios, queries might time out. Also, the number of users who are opening the reports at any one time impact the load that's placed on the data source. For example, if a report page has 20 visuals and 10 people opening that page, 200 queries (or more) will be sent to the data source because each visual will issue one or more queries.

Unfortunately, the performance of your semantic model is not only impacted by the performance of the underlying data source, but also by other uncontrollable factors, such as:

- Network latency; faster networks return data quicker.
- The performance of the data source's server and how many other workloads are on that server. For example, consider the implications of a server refresh taking place while hundreds of people are using the same server for different reasons.

Therefore, using DirectQuery poses a risk to the quality of your model's performance. To optimize performance in this situation, you need to have control over, or access to, the source database.

For more detailed information, see [DirectQuery model guidance in Power BI Desktop](#).

Implications of using DirectQuery

It's best practice to import data into your model tables, but your organization might need to use the DirectQuery storage mode because of one of the following reasons (benefits of DirectQuery):

- It's suitable in cases where data changes frequently and near real-time reporting is required.
- It can handle large data without the need to pre-aggregate.
- It applies data sovereignty restrictions to comply with legal requirements.
- It can be used with a multidimensional data source that contains measures such as SAP Business Warehouse (BW).

If your organization wants to use DirectQuery, you should clearly understand its behavior and be aware of its limitations. You will then be in a good position to take action to optimize the DirectQuery model as much as possible.

Behavior of DirectQuery connections

When you use DirectQuery to connect to data in Power BI Desktop, that connection behaves in the following way:

- When you initially use the **Get Data** feature in Power BI Desktop, you will select the source. If you connect to a relational source, you can select a set of tables and each one will define a query that logically returns a set of data. If you select a multidimensional source, such as SAP BW, you can only select the source.
- When you load the data, no data is imported into the Power BI Desktop, only the schema is loaded. When you add a visual to a report, queries are sent to the underlying source to retrieve the necessary data. The time it takes to refresh the visual depends on the performance of the underlying data source.
- If changes are made to the underlying data, they won't be immediately reflected in the existing visuals due to caching. You need to carry out a refresh to see those changes. The necessary queries are present for each visual, and the visuals are updated accordingly.
- When you publish the Power BI Desktop file to the Power BI service, it publishes a semantic model. However, no data is included with that semantic model.

- When you open an existing report in Power BI service (or build a new one), the underlying source is again queried to retrieve the necessary data. Depending on the location of the original source, you might have to configure an on-premises data gateway.
- You can pin visuals, or entire report pages, as dashboard tiles. The tiles are automatically refreshed on a schedule, for example, every hour. You can control the frequency of this refresh to meet your requirements. When you open a dashboard, the tiles reflect the data at the time of the last refresh and might not include the latest changes that are made to the underlying data source. You can always refresh an open dashboard to ensure that it's up to date.

Limitations of DirectQuery connections

The use of DirectQuery can have negative implications. The limitations vary, depending on the specific data source that is being used. You should take the following points into consideration:

- **Performance** - As previously discussed, your overall user experience depends heavily on the performance of the underlying data source.
- **Security** - If you use multiple data sources in a DirectQuery model, it's important to understand how data moves between the underlying data sources and the associated security implications. You should also identify whether security rules are applicable to the data in your underlying source because, in Power BI, every user can see that data.
- **Data transformation** - Compared to imported data, data that is sourced from DirectQuery has limitations when it comes to applying data transformation techniques with Power Query. For example, if you connect to an OLAP source, such as SAP BW, you can't make any transformations at all; the entire external model is taken from the data source. If you want to make any transformations to the data, you will need to do so in the underlying data source.
- **Modeling** - Some of the modeling capabilities that you have with imported data aren't available, or are limited.
- **Reporting** - Almost all the reporting capabilities that you have with imported data are also supported for DirectQuery models, provided that the underlying source offers a suitable level of performance.

For more detailed information on the limitations of using DirectQuery, see [Implications of using DirectQuery](#).

Now that you have an understanding of how DirectQuery works and its limitations, you can take action to improve the performance.

Optimize performance

Continuing with the Tailwind Traders scenario, during your review of the semantic model, you discover that the model uses DirectQuery to connect to source data. This use of DirectQuery is the reason why users are experiencing poor report performance. It's taking too long to load the pages in the report, and tables are not refreshing quickly enough when certain selections are made. You need to take action to optimize the performance of the DirectQuery model.

You can examine the queries that are being sent to the underlying source and try to identify the reason for the poor query performance. You can then make changes in Power BI Desktop and the underlying data source to optimize overall performance.

Optimize data in Power BI Desktop

When you've optimized the data source as much as possible, you can take further action within Power BI Desktop by using **Performance Analyzer**, where you can isolate queries to validate query plans.

You can analyze the duration of the queries that are being sent to the underlying source to identify the queries that are taking a long time to load. In other words, you can identify where bottlenecks exist.

You don't need to use a special approach when optimizing a DirectQuery model; you can apply the same optimization techniques that you use to tune imported data. For example, you can reduce the number of visuals on the report page or reduce the number of fields that are used in a visual. You can also remove unnecessary columns and rows.

For more detailed guidance on how to optimize a DirectQuery query, see: [DirectQuery model guidance in Power BI Desktop](#) and [Guidance for using DirectQuery successfully](#).

Optimize the underlying data source (connected database)

Your first stop is the data source. You need to tune the source database as much as possible because anything you do to improve the performance of that source database will in turn improve Power BI DirectQuery. The actions that you take in the database will do the most good.

Consider the use of the following standard database practices that apply to most situations:

- Avoid the use of complex calculated columns because the calculation expression will be embedded into the source queries. It's more efficient to push the expression back to the source because it avoids the push down. You could also consider adding surrogate key columns to dimension tables.
- Review source table indexes and verify that the current indexing is optimal. If you need to create new indexes, ensure that they're appropriate.

Refer to the guidance documents of your data source and implement their performance recommendations.

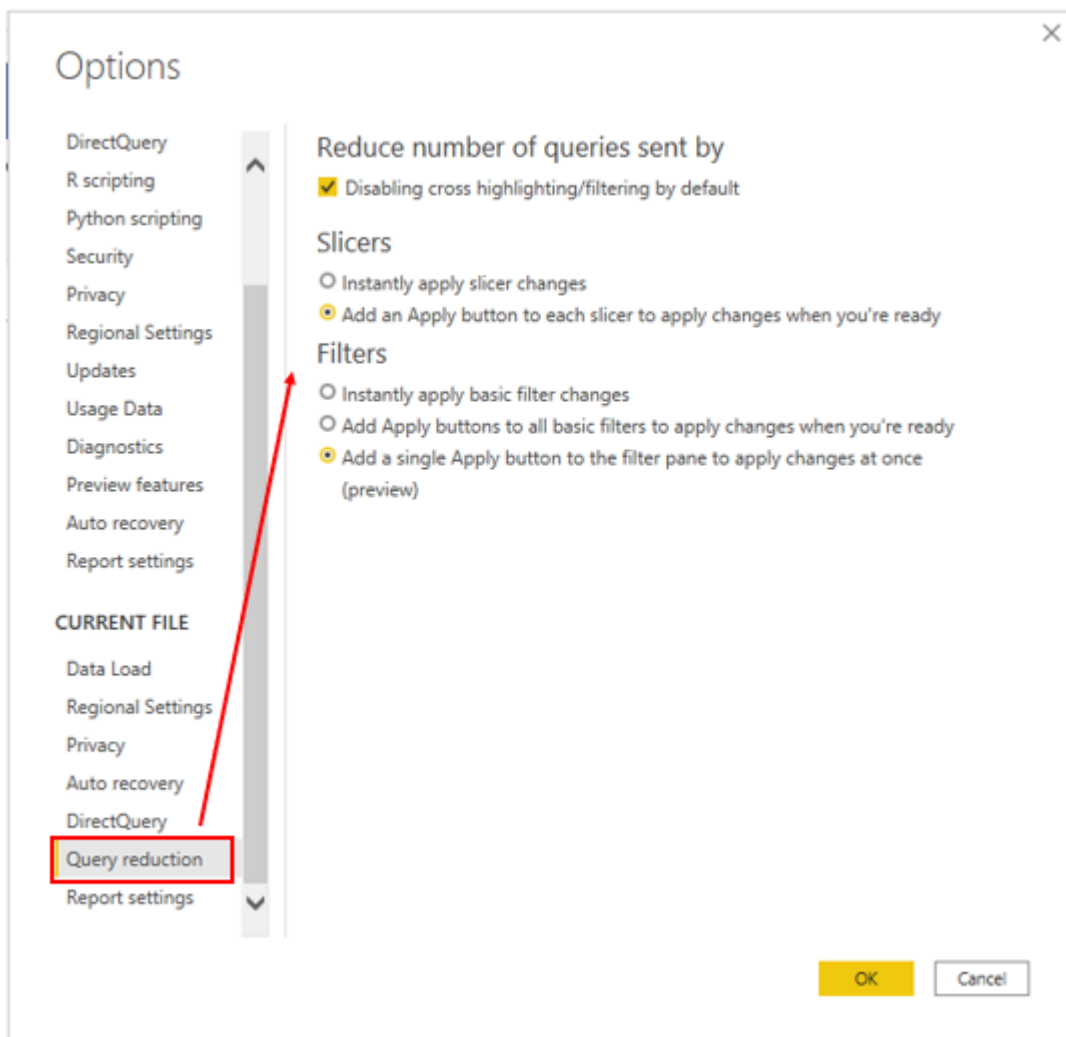
Customize the Query reduction options

Power BI Desktop gives you the option to send fewer queries and to disable certain interactions that will result in a poor experience when the resulting queries take a long time to run. Applying these options prevents queries from continuously hitting the data source, which should improve performance.

In this example, you edit the default settings to apply the available data reduction options to your report. You access the settings by selecting **File > Options and settings > Options**, scrolling down the page, and then selecting the **Query reduction** option.

The following query reduction options are available:

- **Reduce number of queries sent by** - By default, every visual interacts with every other visual. Selecting this check box disables that default interaction. You can then choose which visuals interact with each other by using the **Edit interactions** feature.
- **Slicers** - By default, the **Instantly apply slicer changes** option is selected. To force the report users to manually apply slicer changes, select the **Add an apply button to each slicer to apply changes when you're ready** option.
- **Filters** - By default, the **Instantly apply basic filter changes** option is selected. To force the report users to manually apply filter changes, select one of the alternative options:
 - **Add an apply button to all basic filters to apply changes when you're ready**
 - **Add a single apply button to the filter pane to apply changes at once (preview)**



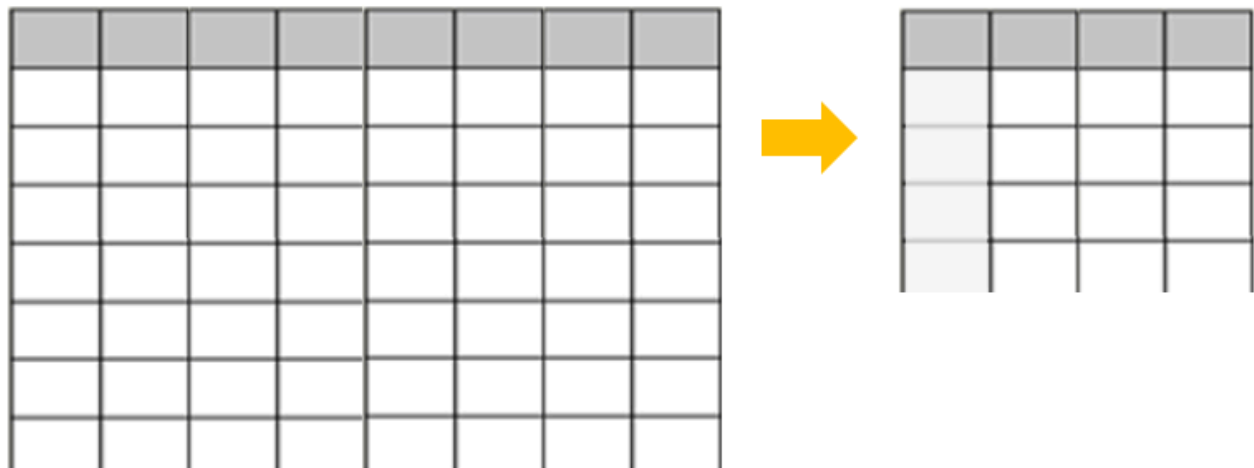
7. Create and manage aggregations

<https://learn.microsoft.com/en-us/training/modules/optimize-model-power-bi/6-aggregations>

Create and manage aggregations

Completed

When aggregating data, you summarize that data and present it in at a higher level of granularity. For example, you can summarize sales data and group it by date, customer, product, and so on. The aggregation process reduces the table sizes in the semantic model, allowing you to focus on important data and helping to improve the query performance.



Your organization might decide to use aggregations in their semantic models for the following reasons:

- You work with large volumes of data. In this case, aggregations provide better query performance and help you analyze and reveal the insights over this large data. Aggregated data is cached and, therefore, uses a fraction of the resources that are required for detailed data.
- You experience a slow data refresh. In this case, aggregations help to speed up the refresh process. The smaller cache size reduces the refresh time, so data gets to users faster. Instead of refreshing what could be millions of rows, you refresh a smaller amount of data instead.
- You have a large semantic model. In this case, aggregations help to reduce and maintain the size of your model.
- You anticipate future growth of your semantic model. In this case, you can use aggregations as a proactive step toward future proofing your semantic model by lessening the potential for performance and refresh issues and overall query problems.

Continuing with the Tailwind Traders scenario, you have taken several steps to optimize the performance of the semantic model, but the IT team has informed you that the file size is still too large. The file size is currently 1 gigabyte (GB), so you need to reduce it to around 50 megabytes (MB). During your performance review, you identified that the previous developer didn't use aggregations in the semantic model, so you now want to create aggregations for the sales data to reduce the file size and further optimize the performance.

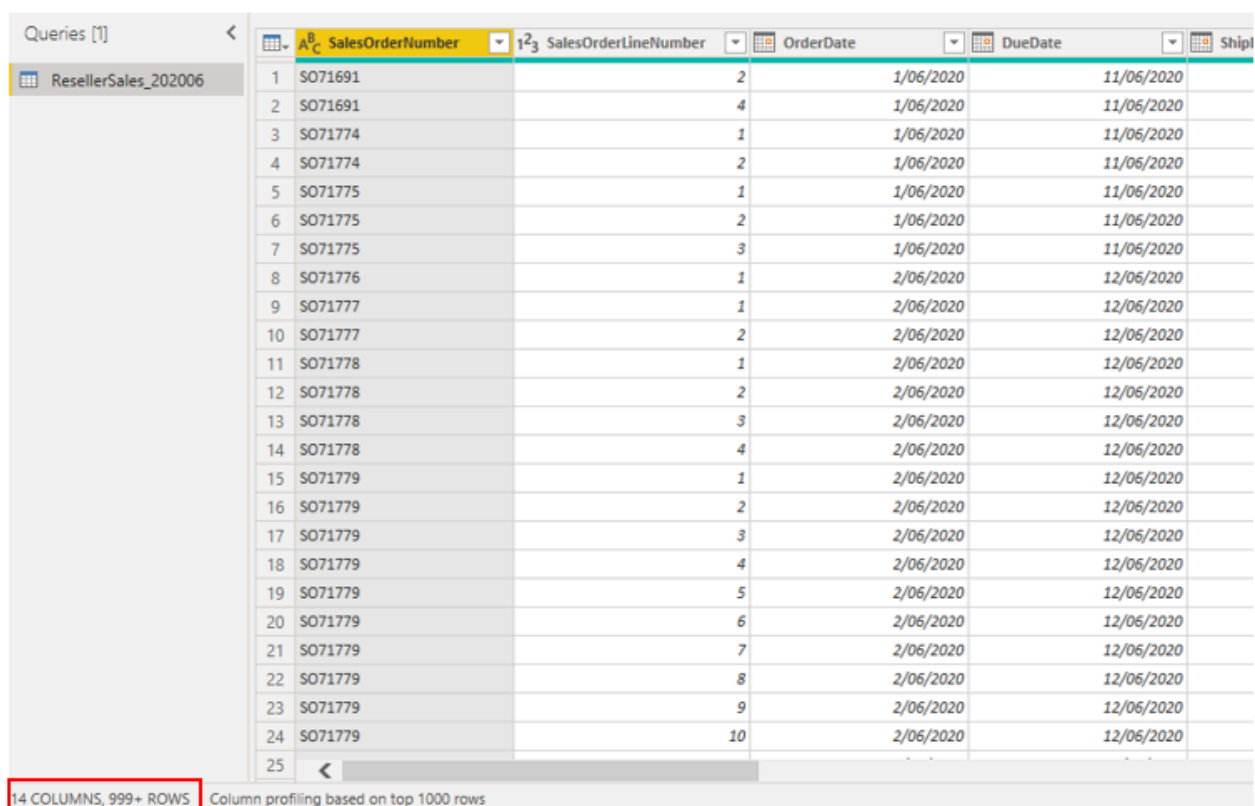
Create aggregations

Before you create aggregations, you should decide on the level of granularity on which you want to create them. In this example, you want to aggregate the sales data at the day level.

When you decide on the grain, the next step is to decide on how you want to create the aggregations. You can create aggregations in different ways and each method will yield the same results, for example:

- If you have access to the database, you can create a table (or view) and then import it into Power BI Desktop.
- In Power BI Desktop, you can use Power Query to create the aggregations step-by-step.

In this example, you open a query in Power Query and notice that the data has not been aggregated; it has over 999 rows, as illustrated the following screenshot.

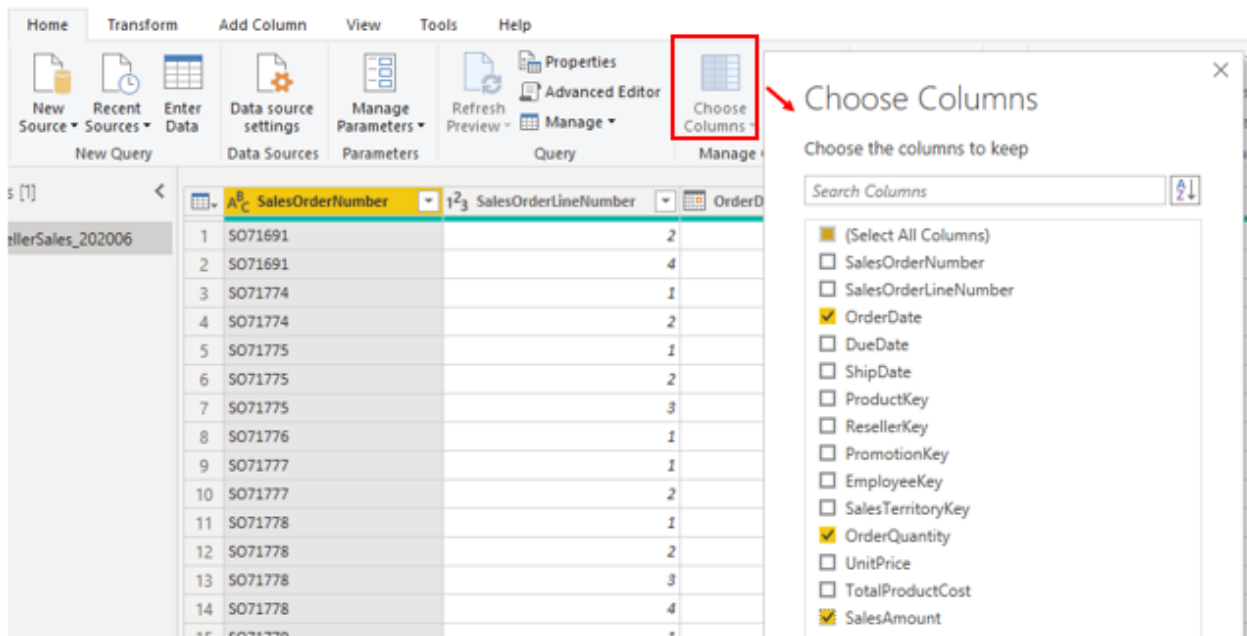


	1	2	3	4	5
	SalesOrderNumber	SalesOrderLineNumber	OrderDate	DueDate	ShipDate
1	SO71691	2	1/06/2020	11/06/2020	
2	SO71691	4	1/06/2020	11/06/2020	
3	SO71774	1	1/06/2020	11/06/2020	
4	SO71774	2	1/06/2020	11/06/2020	
5	SO71775	1	1/06/2020	11/06/2020	
6	SO71775	2	1/06/2020	11/06/2020	
7	SO71775	3	1/06/2020	11/06/2020	
8	SO71776	1	2/06/2020	12/06/2020	
9	SO71777	1	2/06/2020	12/06/2020	
10	SO71777	2	2/06/2020	12/06/2020	
11	SO71778	1	2/06/2020	12/06/2020	
12	SO71778	2	2/06/2020	12/06/2020	
13	SO71778	3	2/06/2020	12/06/2020	
14	SO71778	4	2/06/2020	12/06/2020	
15	SO71779	1	2/06/2020	12/06/2020	
16	SO71779	2	2/06/2020	12/06/2020	
17	SO71779	3	2/06/2020	12/06/2020	
18	SO71779	4	2/06/2020	12/06/2020	
19	SO71779	5	2/06/2020	12/06/2020	
20	SO71779	6	2/06/2020	12/06/2020	
21	SO71779	7	2/06/2020	12/06/2020	
22	SO71779	8	2/06/2020	12/06/2020	
23	SO71779	9	2/06/2020	12/06/2020	
24	SO71779	10	2/06/2020	12/06/2020	
25					

14 COLUMNS, 999+ ROWS. Column profiling based on top 1000 rows

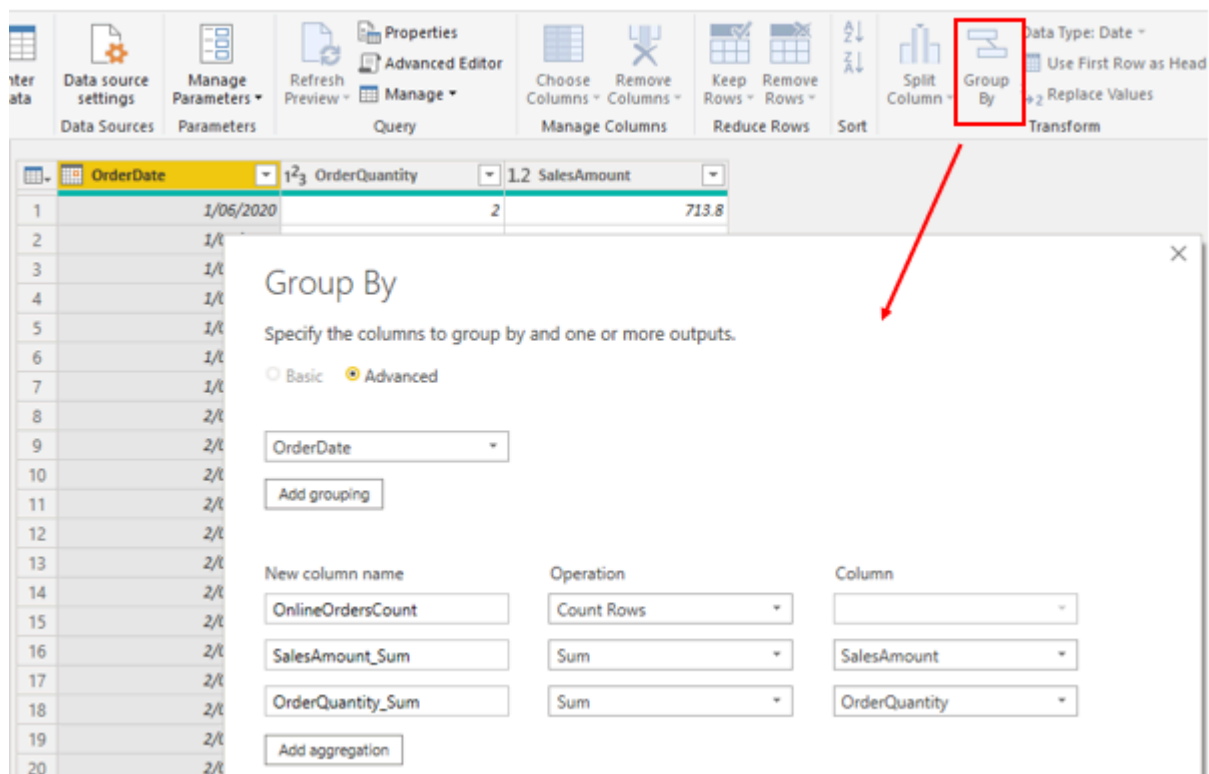
You want to group by the **OrderDate** column and summarize the **OrderQuantity** and **SalesAmount** columns. Start by selecting **Choose Columns** on the **Home** ribbon tab. In the window

that opens, select the columns that you want in the aggregation, and then select **OK**.



When the selected columns display on the page, select the **Group By** option on the **Home** ribbon tab. In the window that opens, select the column that you want to group by (**OrderDate**) and enter a name for the new column (**OnlineOrdersCount**).

Select the **Advanced** option, and then select the **Add aggregation** button to configure another column row. Enter a name for the aggregation column, select the operation of the column, and then select the column to which you want to link the aggregation. Repeat these steps until you have added all the aggregations, and then select **OK**.



It might take a few minutes for a preview of your aggregation to display, but when it does, you'll see how the data has been transformed. The data will be aggregated into each date, and you will be able to see the values for the orders count and the respective sum of the sales amount and order quantity.

	OrderDate	1.2 OnlineOrdersCount	1.2 SalesAmount_Sum	1.2 OrderQuantity_Sum
1	1/06/2020	7	3221.28	14
2	2/06/2020	78	68495.62	225
3	3/06/2020	151	213860.64	738
4	4/06/2020	89	87484.01	218
5	5/06/2020	224	292106.92	978
6	6/06/2020	163	145808.57	572
7	7/06/2020	125	110115.85	557
8	8/06/2020	156	177334.37	441

Select the **Close and Apply** button to close Power Query Editor and apply the changes to your semantic model. In Power BI Desktop, on the **Home** ribbon tab, select **Refresh**. Observe the screen because a brief message will display the number of rows that your semantic model has loaded. This number of rows should be significantly less than the number that you started with. You can also see this number when you open Power Query Editor again, as illustrated in the following screenshot. In this example, the number of rows was reduced to 30.

22	22/06/2020	55	73935.41	129
23	23/06/2020	116	191212.91	371
24	24/06/2020	20	11193.33	26
25	25/06/2020	62	65857.75	183

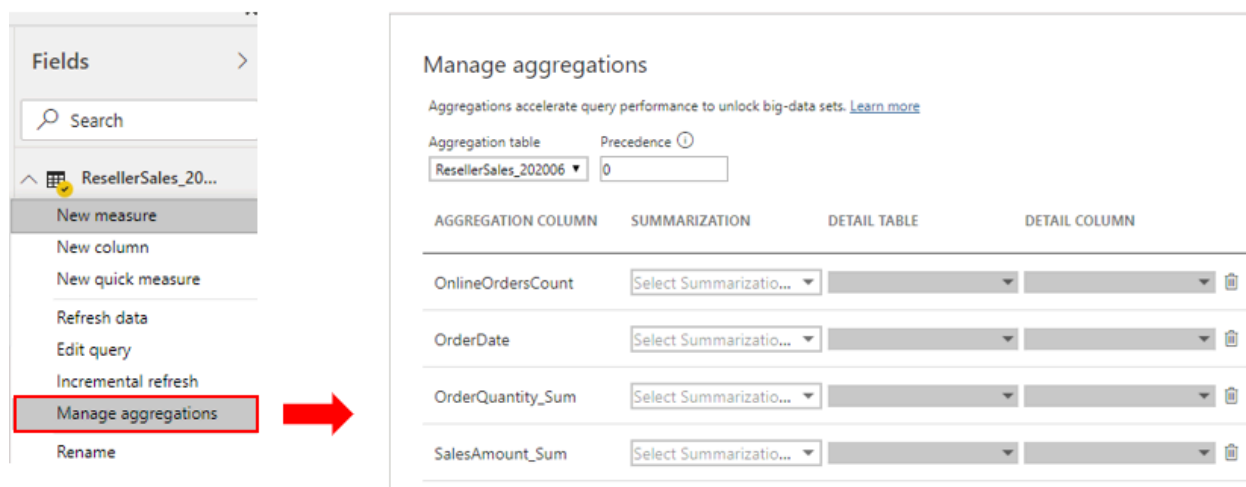
4 COLUMNS, 30 ROWS Column profiling based on top 1000 rows

Remember, you started with 999+ rows. Using aggregation has significantly reduced the number of rows in your semantic model, which means that Power BI has less data to refresh and your model should perform better.

Manage aggregations

You can later manage aggregations in Power BI Desktop to make changes to their behavior, if required.

You can open the **Manage Aggregations** window from any view in Power BI Desktop. In the **Data** pane, right-click the table and then select **Manage aggregations**.



For each aggregation column, you can select an option from the **Summarization** dropdown list and make changes to the selected detail table and column. When you are finished managing the aggregations, select **Apply All**.

For more detailed information on how to create and manage aggregations, see [Use aggregations in Power BI Desktop](#).

8. Check your knowledge

<https://learn.microsoft.com/en-us/training/modules/optimize-model-power-bi/7-check>

Check your knowledge

Completed

9. Summary

<https://learn.microsoft.com/en-us/training/modules/optimize-model-power-bi/8-summary>

Summary

Completed

In this module's scenario, one of your organization's semantic models was inefficient and causing problems. Users were dissatisfied with the report performance, and the model's file size was too large, so it was placing a strain on the organization's resources.

You were asked to review the semantic model to identify the cause of the performance issues and make changes to optimize performance and reduce the size of the model.

Power BI Desktop provides a range of tools and features for you to analyze and optimize the performance of its semantic models. You started the optimization process by using Performance analyzer and other tools to review the performance of measures, relationships, and visuals, and then made improvements based on the analysis results. Next, you used variables to write less complex and more efficient calculations. You then took a closer look at the column distribution and reduced the cardinality of your relationships. At that stage, the semantic model was more optimized. You considered how the situation would be different if your organization used a DirectQuery model, and then you identified how to optimize performance from Power BI Desktop and the source database. Finally, you used aggregations to significantly reduce the size of the semantic model.

If Power BI Desktop didn't give you the opportunity to optimize inefficient semantic models, you would have to spend a lot of time in your multiple data sources to improve the data there. In particular, without **Performance Analyzer**, you wouldn't have identified the reasons for the performance issues in your reports and the bottlenecks in the queries that need to be cleared. As a result, users would be frustrated and unmotivated and might avoid using the reports.

Now that you have optimized the report, users can access the data that they need in a faster time, so they are more productive and have greater job satisfaction. The reduction that you made to the model's file size will ease the strain on resources, bringing about a range of benefits to your organization. You have successfully accomplished the task you were given.