

Domain 3: Work with semantic models in Microsoft Fabric

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>

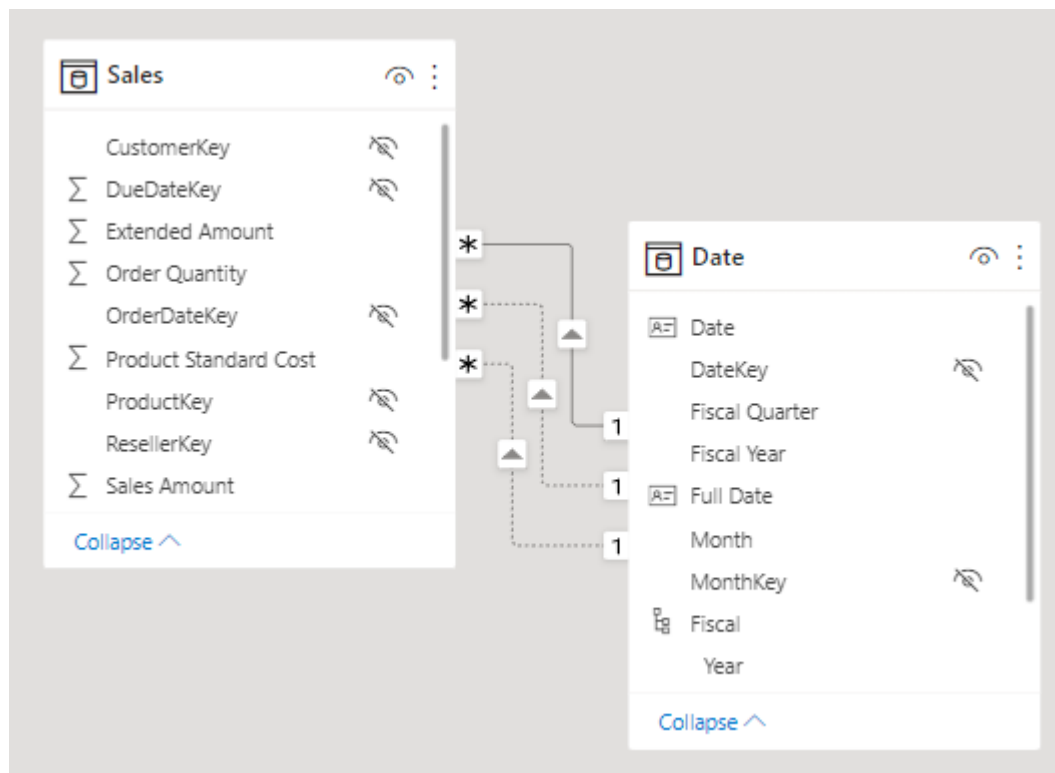
Create calculated tables

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.

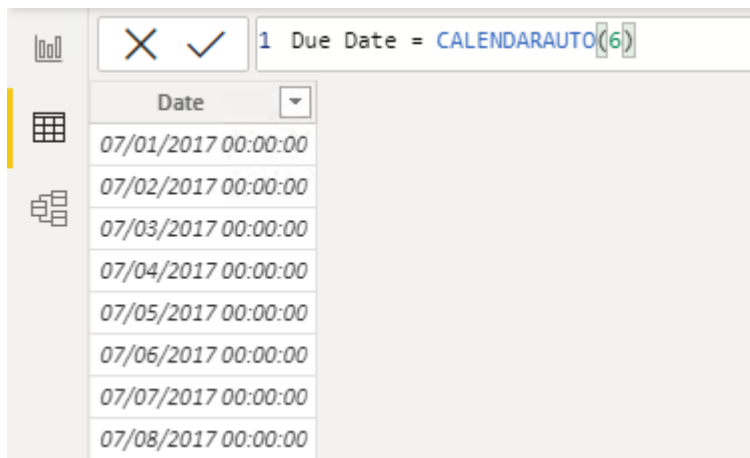
Create a date table

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

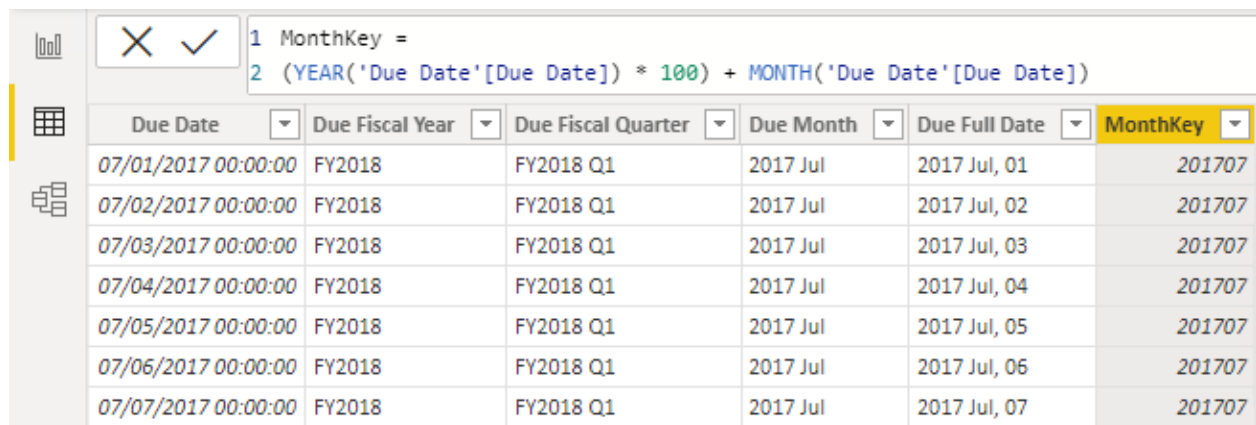
The `FORMAT` function converts the `Due Date` column value to text by using a format string. In this case, the format string produces a label that describes the year, abbreviated month name, and day:

```
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.



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.

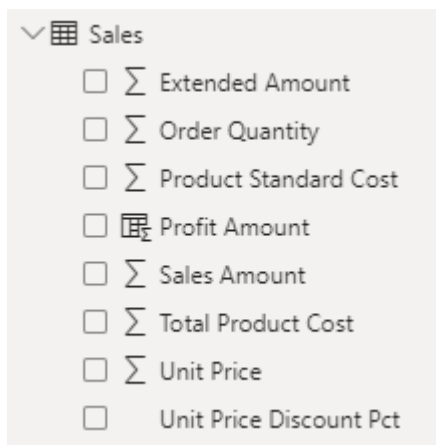
Identify implicit measures

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

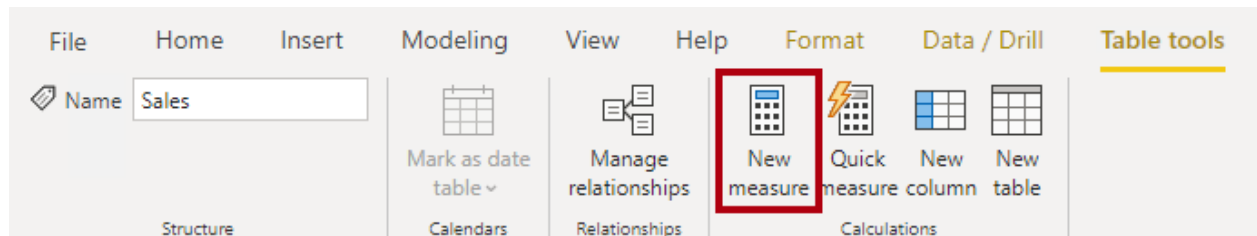
You can add a measure to any table in your model by writing a DAX formula. A measure formula must return a single value. This section explains how to create explicit measures.

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**.



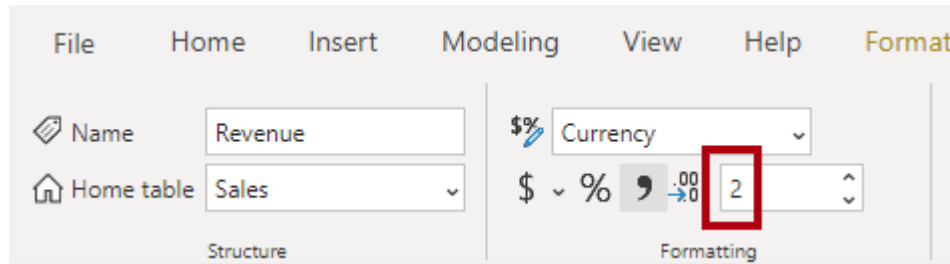
The following formula creates a measure called *Revenue*. This measure uses the **SUM** function to total the values in the **Sales Amount** column. If you add this measure to a table alongside the **Sales Amount** implicit measure, the results are the same.

```
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

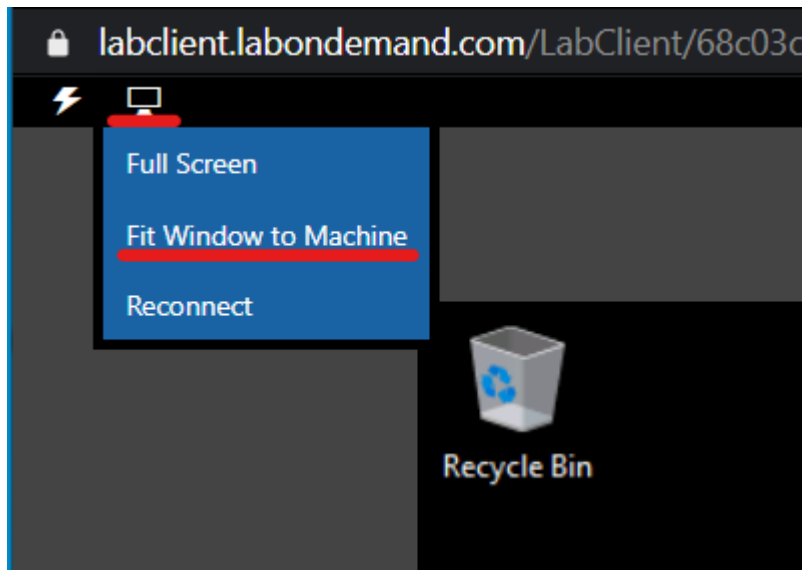
Before you start this lab (unless you are continuing from a previous lab), select **Launch lab** above.

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](#).

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

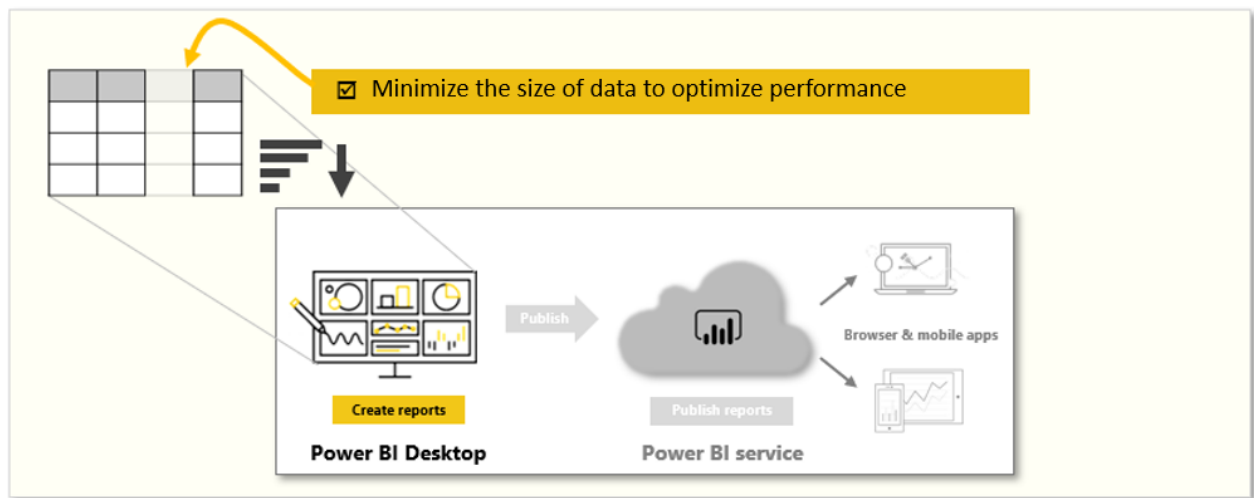
Completed

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.

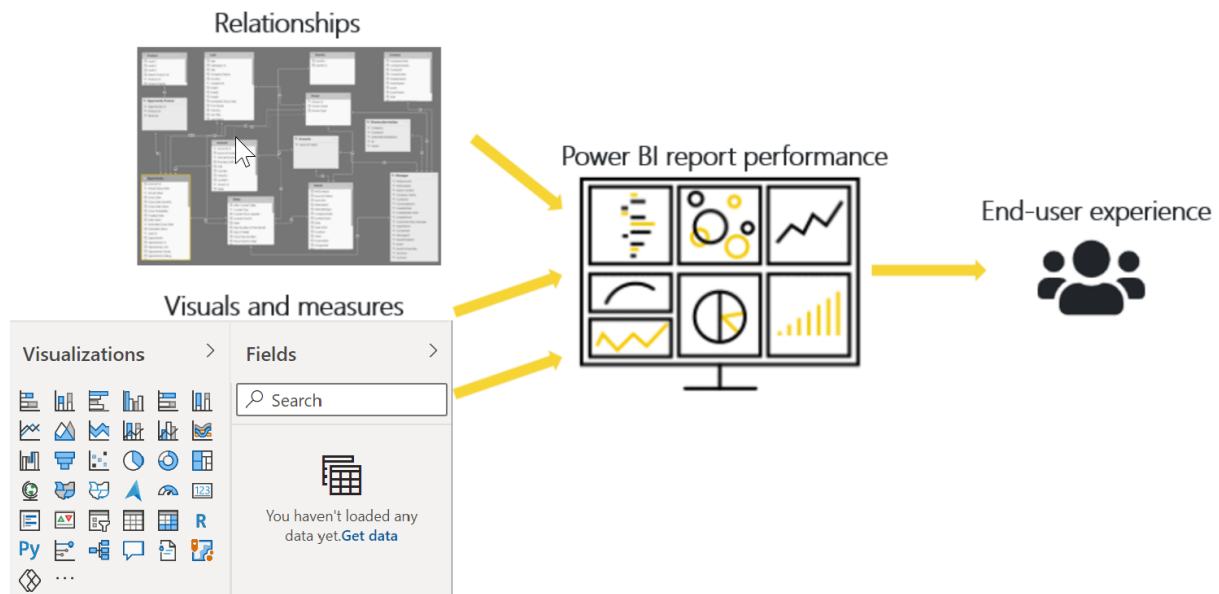
3. Review performance of measures, relationships, and visuals

<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.

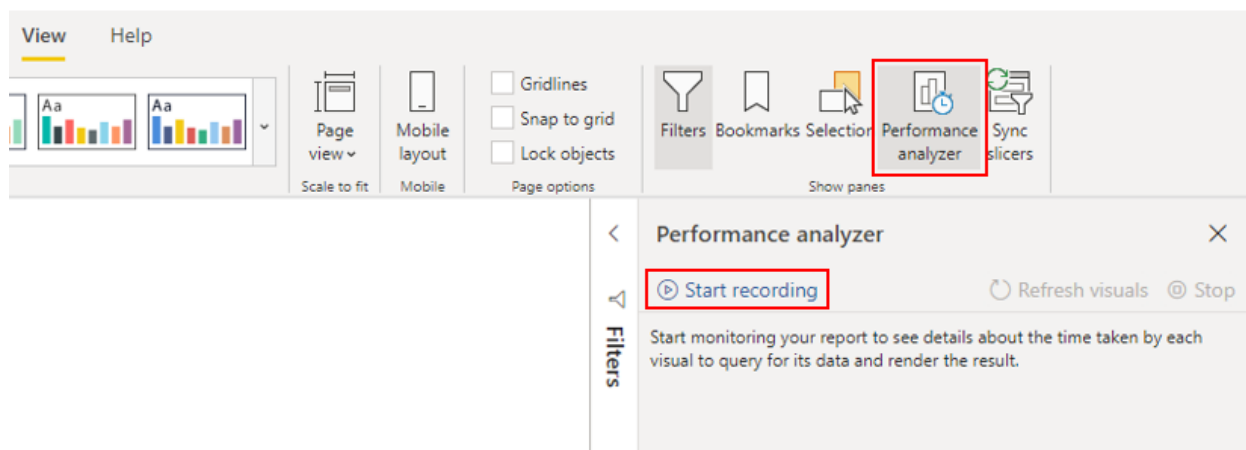
To ensure that you have a clear visual cache, add a blank page to your Power BI Desktop file and then, with that page selected, save and then close the file. Reopen the Power BI Desktop file, which will open on the blank page.

- **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**.

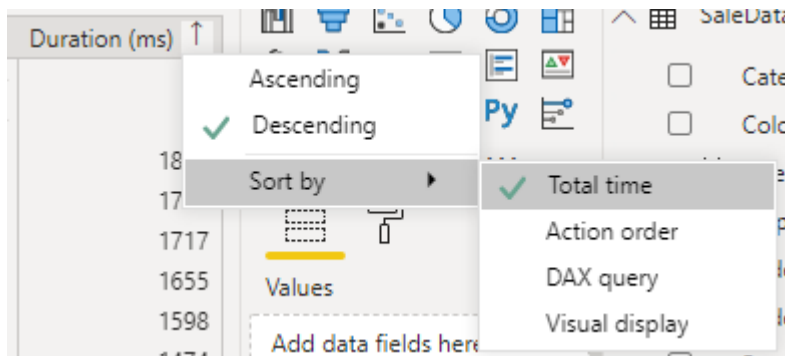
To begin the analysis process, select **Start recording**, select the page of the report that you want to analyze, and then interact with the elements of the report that you want to monitor. You will see the results of your interactions displayed in the **Performance Analyzer** pane as you work. When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

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.

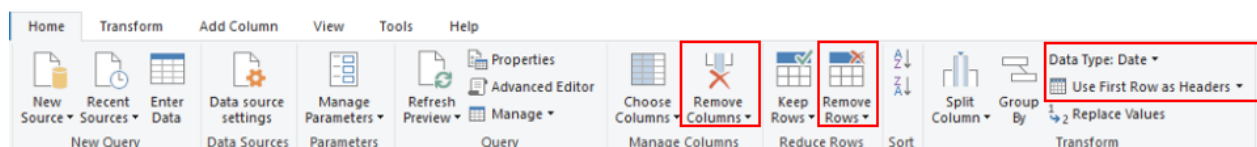
For more information, see [Data reduction techniques for Import modeling](#).

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.

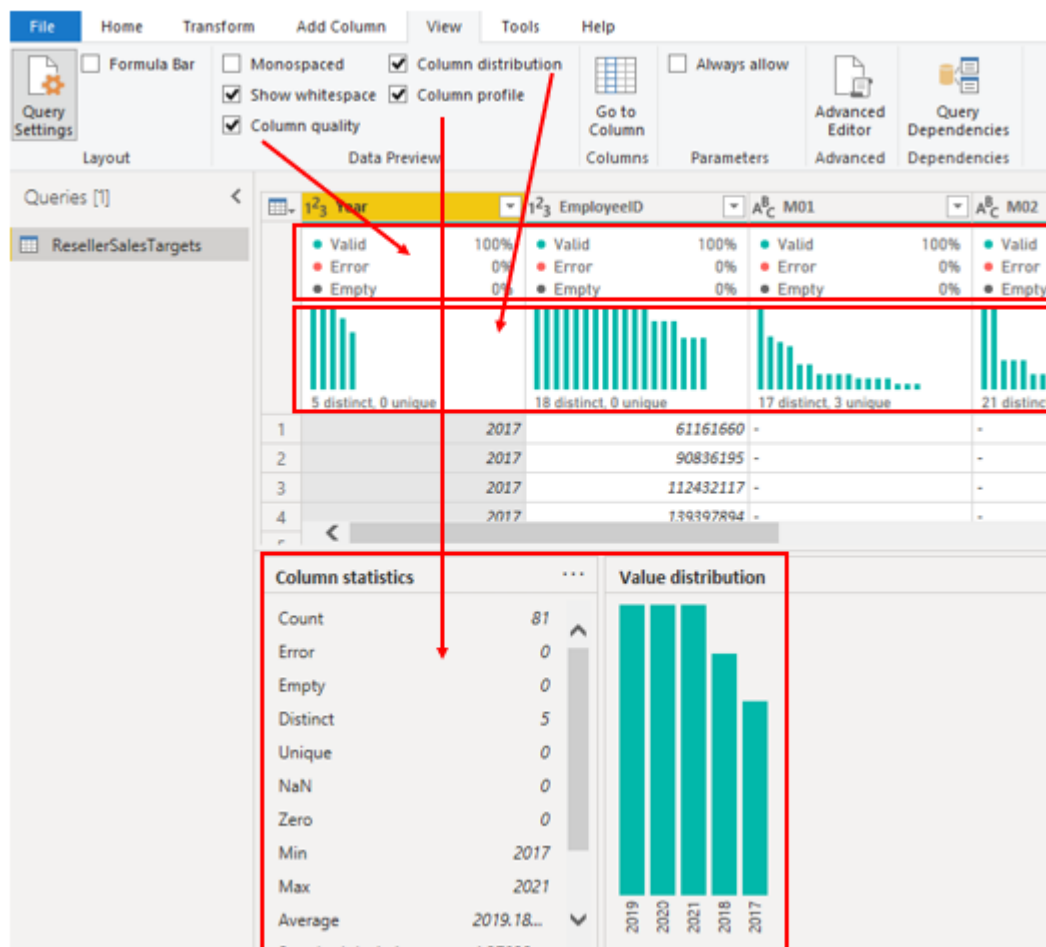
You can use Power Query in Power BI Desktop to examine the columns, rows, and values of your source data. You can then use the available tools, such as those highlighted in the following image, to make the necessary changes.



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

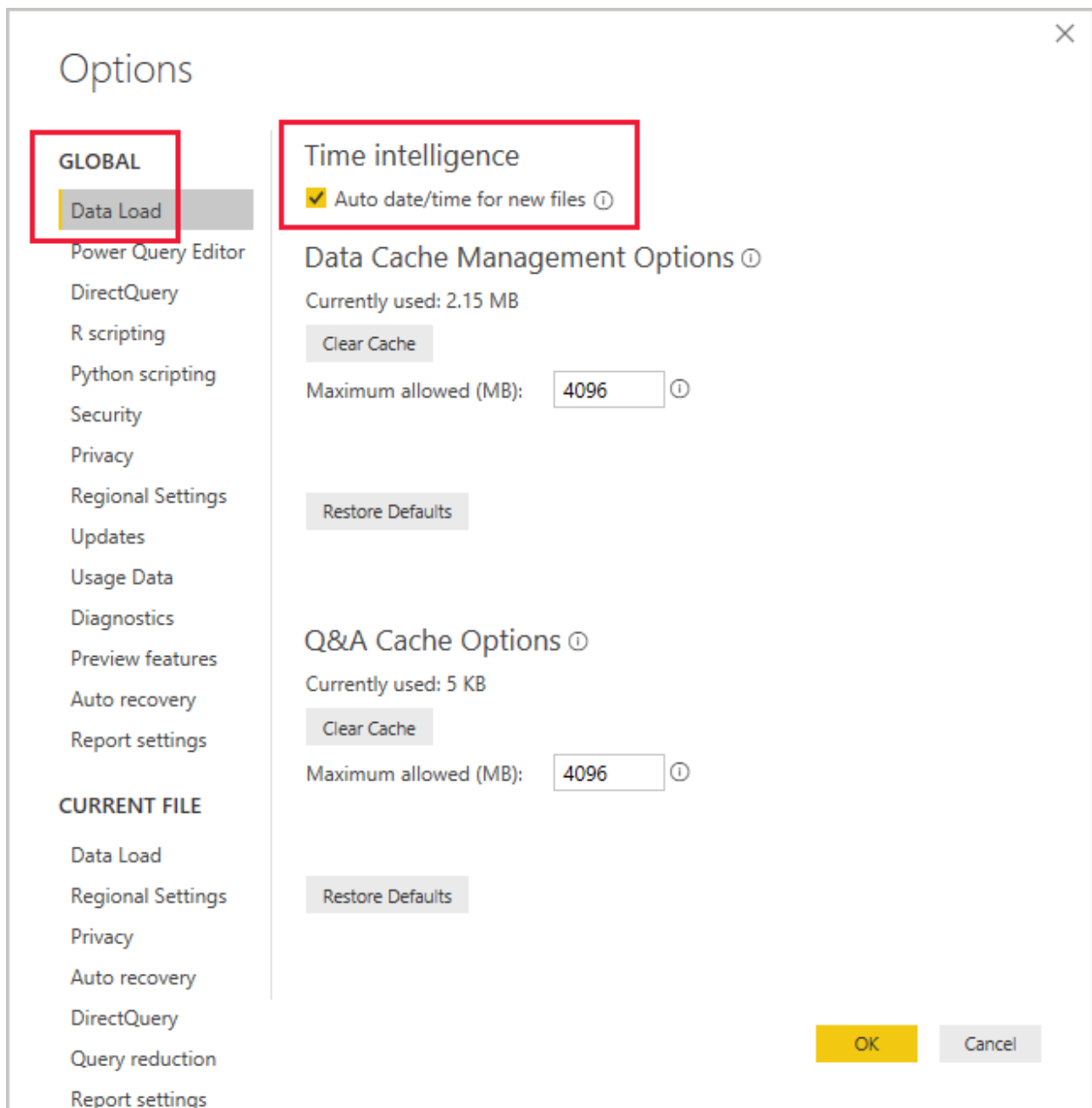
Another item to consider when optimizing performance is the **Auto date/time** option in Power BI Desktop. By default, this feature is enabled globally, which means that Power BI Desktop automatically creates a hidden calculated table for each date column, provided that certain conditions are met. The 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))  
RETURN  
    SalesPriorYear
```

```
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

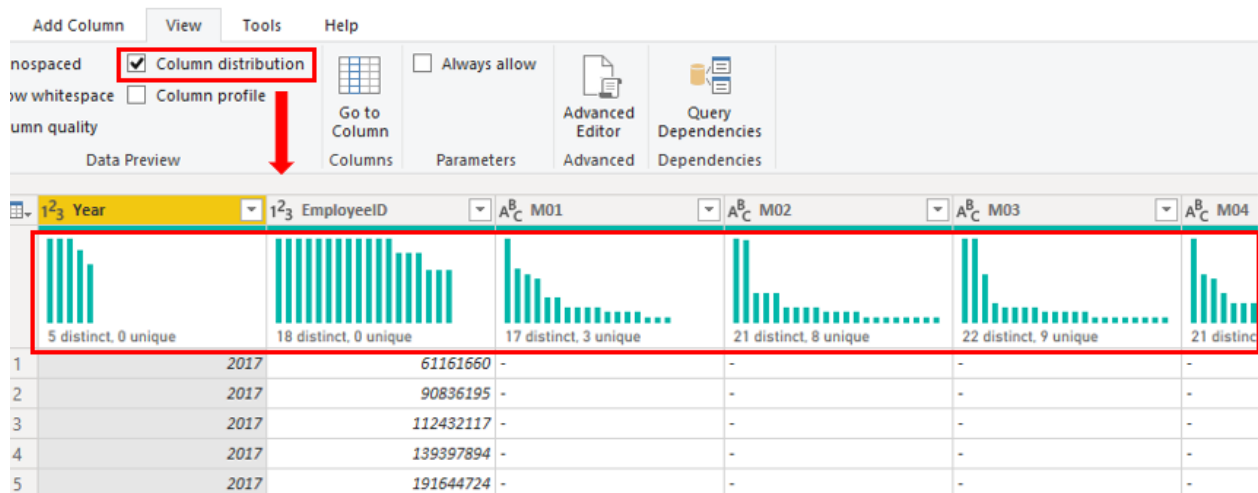
Completed

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.

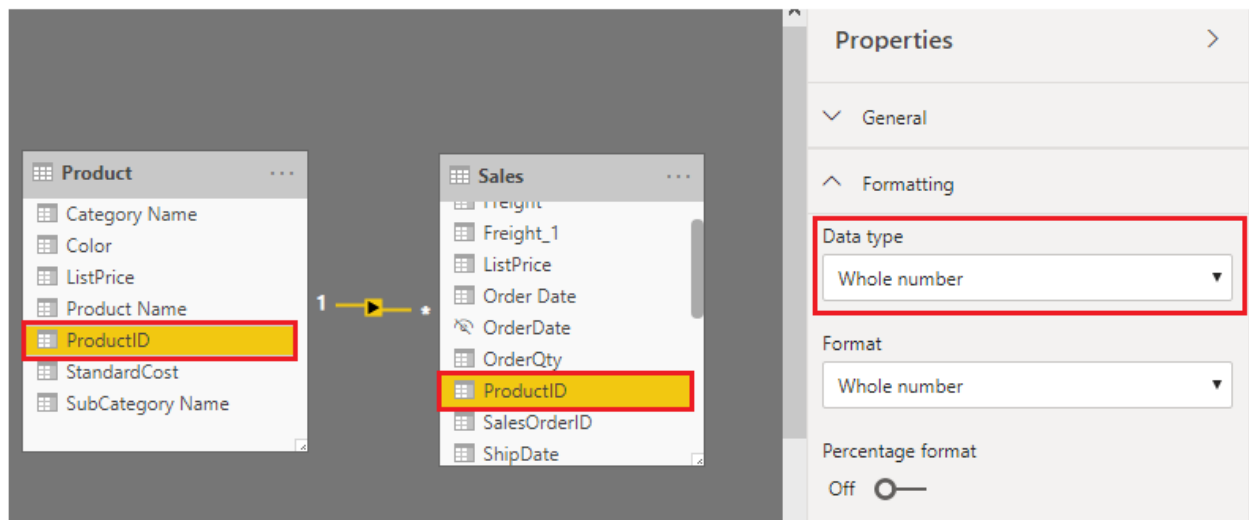
When you create or edit a relationship, you can configure other options. By default, Power BI Desktop automatically configures other options based on its assessment of the model data, which can be different for each relationship based on the data in the columns.

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](#).

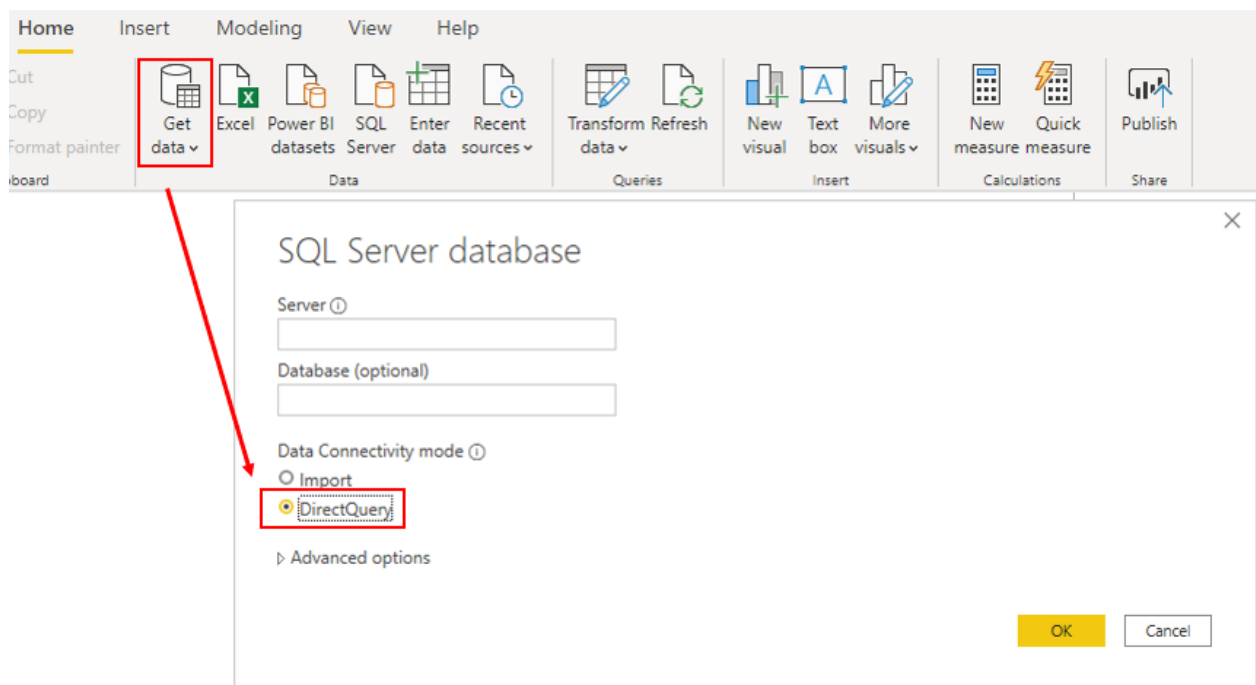
6. Optimize DirectQuery models with table level storage

<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.

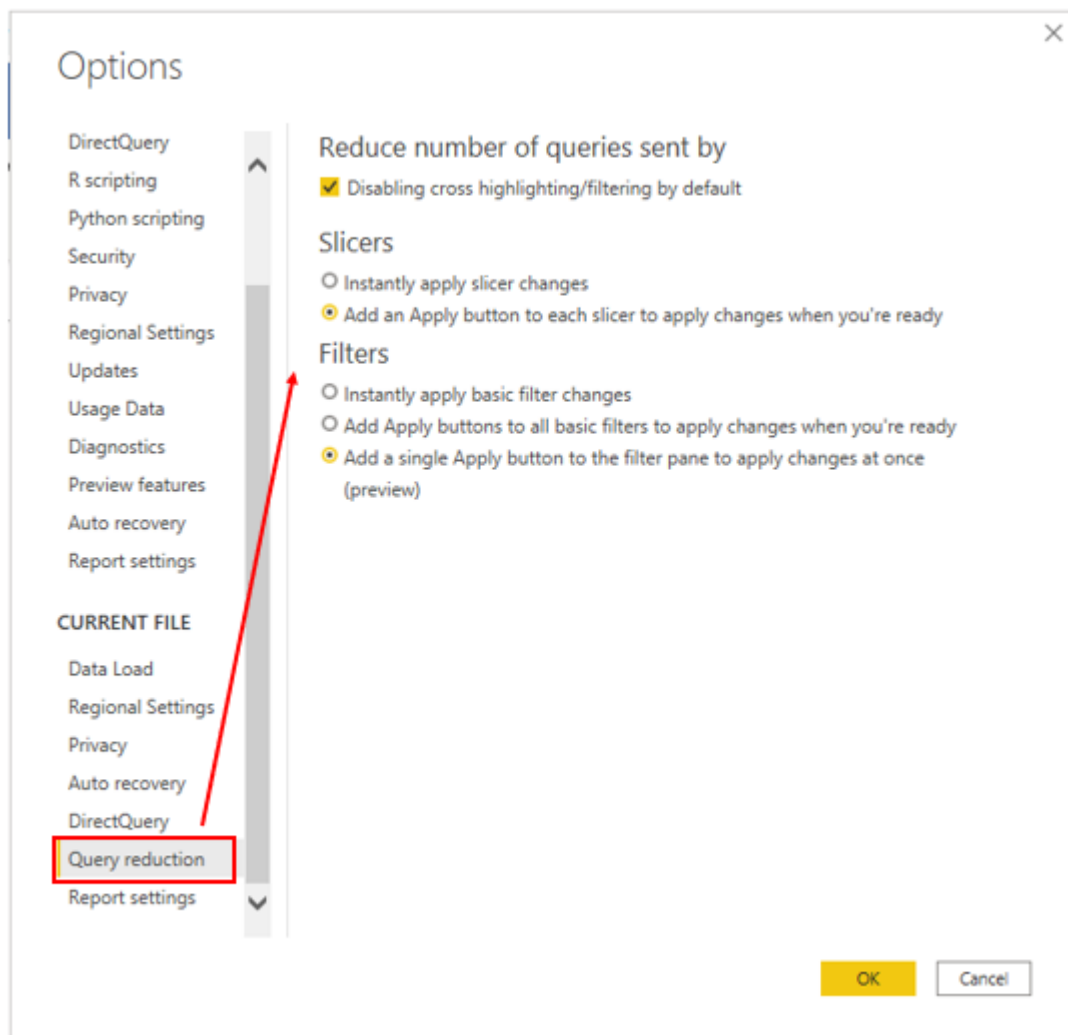
Customize the Query reduction options

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)**



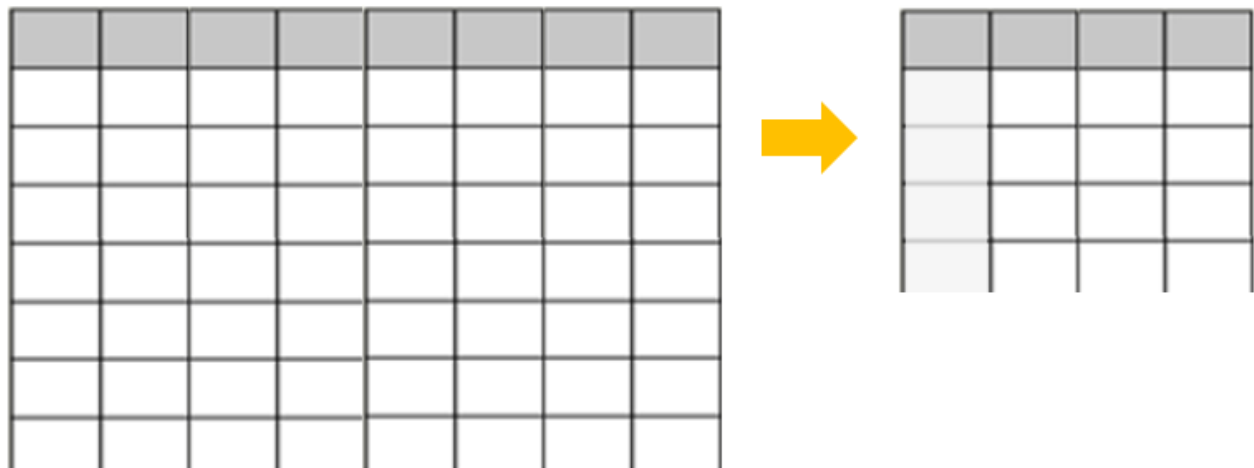
7. Create and manage aggregations

<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.

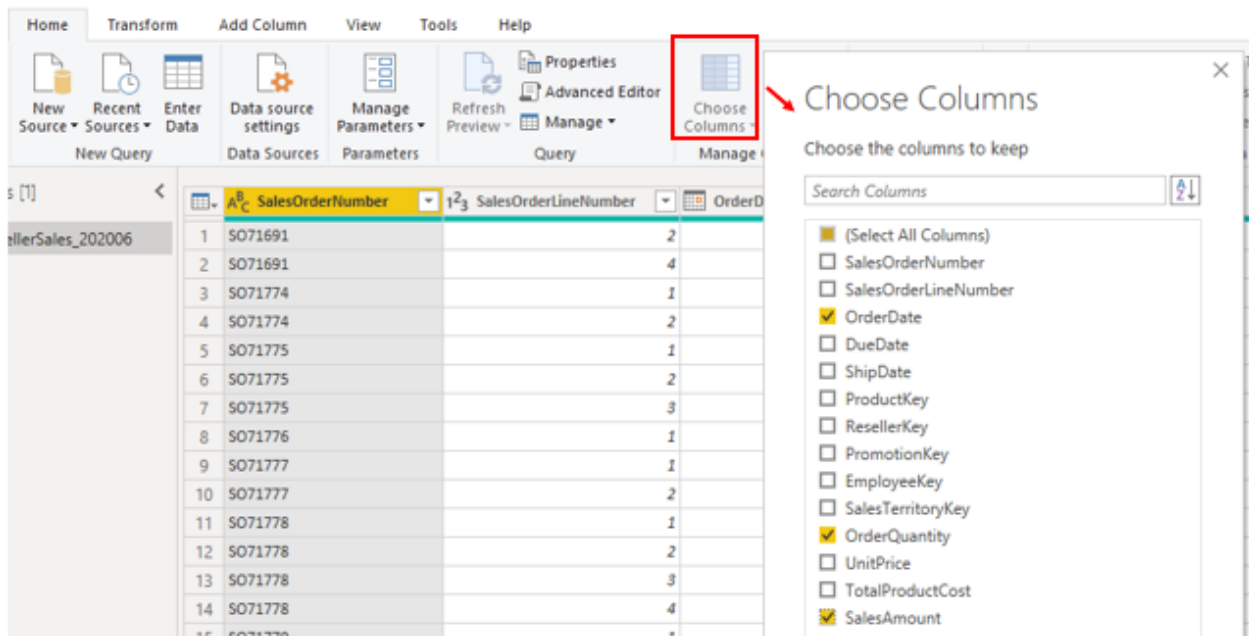
In this example, you open a query in Power Query and notice that the data has not been aggregated; it has over 999 rows, as illustrated the following screenshot.

	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

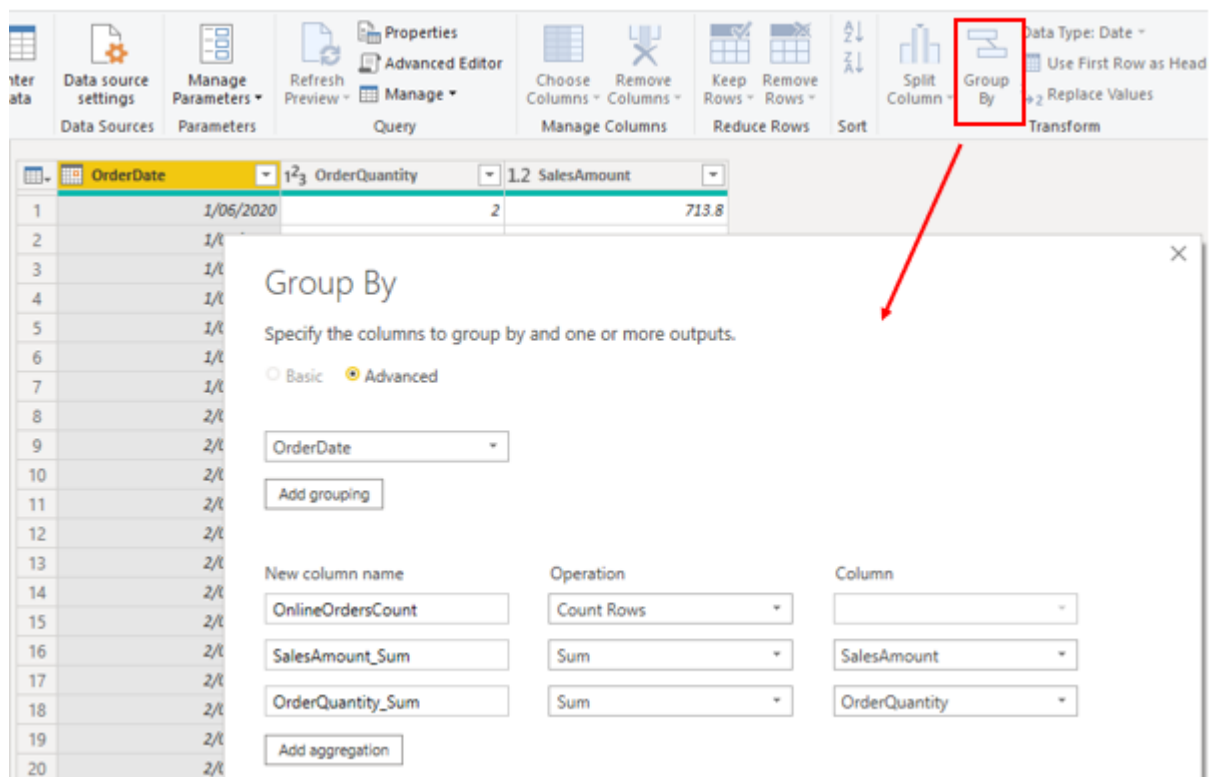
You want to group by the `OrderDate` column and summarize the `OrderQuantity` and `SalesAmount` columns. Start by selecting **Choose Columns** on the **Home** ribbon tab. In the window

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

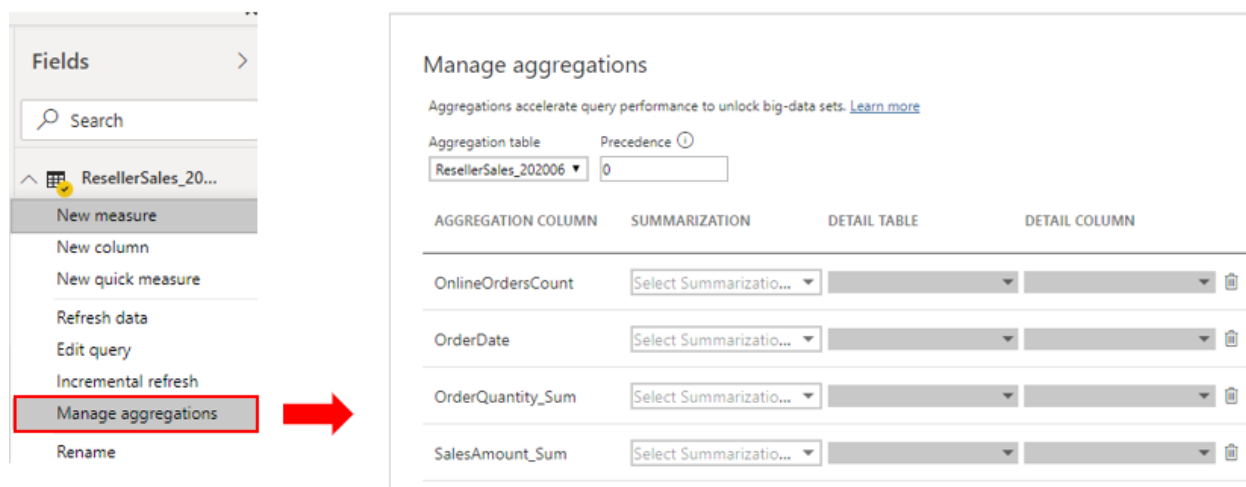
4 COLUMNS, 30 ROWS Column profiling based on top 1000 rows

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.