Chapter 9 of "Expert Data Modeling with Power BI" by Soheli Bakhshi is titled "Star Schema and Data Modeling Common Best Practices" and provides guidance on how to implement best practices for data modeling in Power BI. Here are some of the key points from the chapter:

Start with business requirements: Before starting any data modeling project, it is important to understand the business requirements. This includes identifying the key metrics and dimensions that will be used to analyze the data.

Use a star schema: The star schema is a common data modeling technique that uses a central fact table to store the metrics and related dimension tables to provide context. This makes it easier to analyze and query the data.

Use surrogate keys: Surrogate keys are unique identifiers that are assigned to each record in a table. They are used to establish relationships between tables and make it easier to maintain data integrity.

Normalize data: Normalization is a process of organizing data in a database so that it is consistent and easy to maintain. This involves breaking down tables into smaller, more focused tables that have a single purpose.

Avoid using calculated columns: Calculated columns can be computationally expensive and slow down queries. It is better to use calculated measures instead, which are calculated on the fly during queries.

Use Power Query Editor to prepare data: Power Query Editor is a powerful tool for preparing data before it is loaded into Power BI. This includes cleaning up data, transforming it into the required format, and removing any duplicate or irrelevant data.

Overall, Chapter 9 provides a comprehensive overview of best practices for data modeling in Power BI. By following these guidelines, users can ensure that their data is well-organized, easy to maintain, and optimized for analysis.

Chapter 9 of "Expert Data Modeling with Power BI" by Soheli Bakhshi covers best practices for star schema and data modeling. Here are some examples of how these best practices can be applied in real-world scenarios:

1. Keep it simple: When designing a data model, it's important to keep it simple and easy to understand. This can be especially important in large organizations where multiple people may need to use the data model. By keeping the model simple, users can quickly understand how the data is organized and make better decisions based on the information.

2. Use descriptive and concise names: Naming conventions play a key role in creating a clear and understandable data model. Using descriptive and concise names for tables, columns, and relationships can make it easier for users to navigate and understand the data. For example, a sales table might be named "Sales" rather than "Table 1" to make it clear what data is being represented.

3. Avoid circular references: Circular references can occur when two or more tables reference each other. While this may seem like a convenient way to organize data, it can actually create problems when querying or analyzing the data. It's important to avoid circular references to ensure data integrity and efficient queries.

4. Optimize data types and formats: Choosing the right data types and formats can help ensure data accuracy and reduce the risk of errors. For example, using a numeric data type for a numeric column can help prevent incorrect data entry. It's also important to choose appropriate formats for date and time columns to ensure consistency and accuracy across the data model.

5. Use relationships effectively: Relationships are a key aspect of star schema data modeling. It's important to use relationships effectively to ensure that data is correctly related across tables. This can involve creating one-to-many, many-to-one, or many-to-many relationships depending on the nature of the data.

6. Test and validate the data model: Before deploying a data model, it's important to thoroughly test and validate it to ensure that it's accurate and efficient. This can involve running queries and testing various scenarios to ensure that the data is being correctly organized and displayed.

These best practices can be applied in a wide range of industries and use cases, from retail and finance to healthcare and government. By following these guidelines, organizations can create more effective and efficient data models that support better decision-making and drive business success.

Chapter 9: Star Schema and Data Modeling Common Best Practices

In the previous chapter, we learned a lot about data modeling components in Power BI Desktop, including table and field properties. We also learned about the feature tables and how they make a table from our data model accessible across the organization. We then learned how to build summary tables with DAX. Then we looked at the relationships in more detail; we learned about different relationship cardinalities, filter propagation, and bidirectional relationships. In this chapter, we look at some star schema and data modeling best practices, including the following:

- Dealing with many-to-many relationships
- Being cautious with bidirectional relationships
- Dealing with inactive relationships
- Using configuration tables
- Avoiding calculated columns when possible
- Organizing the model
- Reducing model size by disabling auto date/time

In this chapter, we use the **`Chapter 9, Star Schema and Data Modelling Common Best Practices.pbix`** sample file to go through the scenarios.

Dealing with many-to-many relationships

In the previous chapter, *Chapter 8*, *Data Modeling Components*, we discussed different relationship cardinalities. We went through some scenarios to understand the one-to-one, one-to-many, and many-to-many cardinalities. We showed an example of creating a many-to-many relationship between two tables using non-key columns. While creating a relationship with many-to-many cardinality may work for smaller and less complex data models, it can cause some severe issues if we do not precisely know what we are doing. In some cases, we may get incorrect results in totals; we might find some missing values or get poor performance in large models; while in other cases, we may find the many-to-many cardinality very useful. The message here is that, depending on the business case, we may or may not use many-to-many cardinality; it depends on what works the best for our model while satisfying the business requirements. For instance, the many-to-many cardinality works perfectly fine in the scenario we went through in *Chapter 8*, *Data Modeling Components*, in the *Many-to-many relationships* section. Just as a reminder, the scenario was that the business needed to create analytical sales reports for **Customers with Yearly Income Greater Than $100,000** from a summary table. We created a calculated table on the granularity of **`Customer`**, **`Product`**, and **`Year-Month`**. To enable the business to achieve the requirement, we needed to create a relationship between the **Sales for Customers with Yearly Income Greater Than $100,000** calculated table and the **`Date`** table using the **`Year-Month`** column on both sides. The following diagram shows the many-to-many cardinality relationship between the **Sales for Customers with Yearly Income Greater Than $100,000** table and the **`Date`** table via the **`Year-Month`** column:

Figure 9.1 – A many-to-many cardinality relationship

In the preceding model, the **`Year-Month`** column is not a key column in any of the tables participating in the relationship, which means the **`Year-Month`** column has duplicate values on both sides. When we create a relationship between two tables using non-key columns, we are creating a many-to-many **cardinality** relationship. We insist on using the term "cardinality" for this kind of relationship to avoid confusing it with the **classic** many-to-many relationship. In fact, in relational data modeling, the many-to-many relationship is just a conceptual relationship between two tables via a bridge table. Indeed, in classic relational data modeling, we cannot create a physical many-to-many relationship between two tables. Instead, we always require creating relationships between the primary key on the **one** side of the relationship to the corresponding foreign key column on the **many** side of the relationship. Therefore, the only legitimate kinds of relationships from a classic relational data modeling viewpoint are the one-to-one and the one-to-many relationships. Hence, there is no such relationship kind like many-to-many.

Nevertheless, many business scenarios require many-to-many relationships. Consider banking: a customer can have many accounts, and an account can link to many customers when it is a joint account; or in an education system, a student can have multiple teachers, and a teacher can have many students.

Let's use our **`Chapter 9, Star Schema and Data Modelling Common Best Practices.pbix`** sample file with a scenario.

The business needs to analyze their online customers' buying behavior for **`Quantity Sold`** over **`Sales Reasons`**. The following diagram shows the data model. Let's have a look at it:

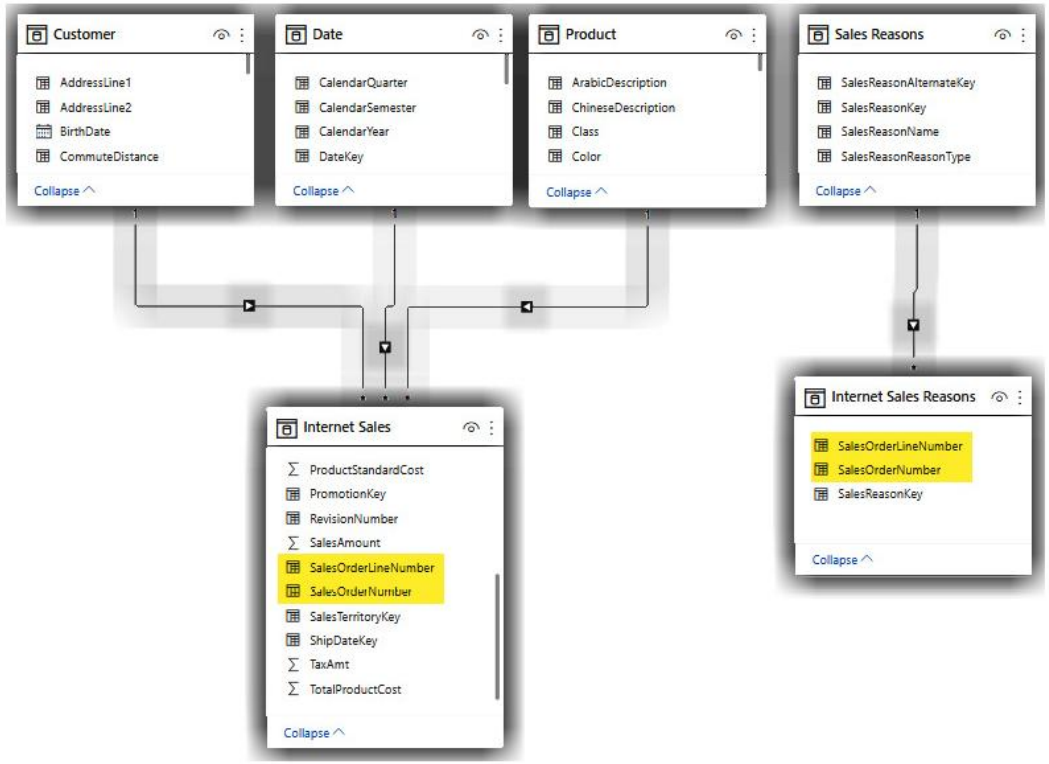


Figure 9.2 – Analyzing Internet Sales by Sales Reasons

As we see in the preceding diagram, we have a **`Sales Reasons`** table containing the descriptive data, and we also have an **`Internet Sales Reasons`** table having a one-to-many relationship with the **`Sales Reasons`** table. Looking closer at the **`Internet Sales Reasons`** table reveals that this table does not have any sales transactions in it. It only contains three columns, the **`SalesOrderLineNumber`**, **`SalesOrderNumber`**, and **`SalesReasonKey`** columns. On the other hand, we have the **`Customer`** table with a one-to-many relationship to the **`Internet Sales`** table. We keep all sales transactions in the **`Internet Sales`** table, where each row of data is unique for the combination of the **`SalesOrderLineNumber`** and the **`SalesOrderNumber`** columns. But there is currently no relationship between the **`Customer`** table and the **`Sales Reasons`** table. Each customer may have several reasons to buy products online, and each sales reason relates to many customers. Therefore, conceptually, there is a many-to-many relationship between the **`Customer`** and the **`Sales Reason`** tables. It is now time to refer back to the classic type of many-to-many relationship we always see in relational data modeling. As mentioned earlier, in relational data modeling, unlike in Power BI, we can only implement the many-to-many relationship using a bridge table regardless.

Many-to-many relationships using a bridge table

In classic relational data modeling, we put the primary keys of both tables participating in the relationship into an intermediary table referred to as a bridge table. The bridge tables usually are available in the transactional source systems. For instance, there is always a many-to-many relationship between a customer and a product in a sales system. A customer can buy many products, and a product can end up in many customers' shopping bags. What happens in the sales system is that when we go to the cashier to pay for the products we bought, the cashier scans each product's barcode. So the system now knows which customer bought which product.

Using the Star Schema approach, we spread the columns across **dimensions** and **facts** when we design a data warehouse. Therefore, in a sales data model (in a sales data warehouse), there is always a many-to-many relationship between the **dimensions** surrounding a fact table via the fact table itself. So, when in a sales data model designed in the Star Schema approach, we have a **`Customer`** table and **`Product`** table containing the *descriptive* values. The **`Customer`** and the **`Product`** tables are **dimensions** from a Star Schema perspective. We also have a **`Sales`** table that holds the foreign keys for both **`Customer`** and **`Product`** tables. The **`Sales`** table also keeps the **`numeric`** values related to sales transactions such as sales amount, tax amount, ordered quantity, and so on. The **`Sales`** table is a fact table in the Star Schema approach. Then we create the relationships between the **`Customer`** table, the **`Product`** table, and the **`Sales`** table. The following diagram shows how the **`Customer`**, the **`Product`**, and the **`Internet Sales`** tables are related:

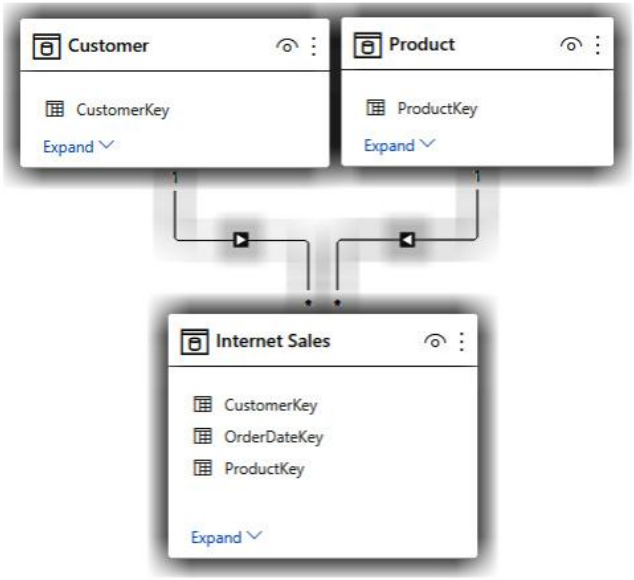


Figure 9.3 – Relationships between Customer, Product, and Internet Sales

In the preceding data model, we have the following relationships:

- A one-to-many relationship between **Customer** and **Sales**
- A one-to-many relationship between **Product** and **Sales**
- A many-to-many relationship between **Customer** and **Product** (via **Sales** table)

The first two relationships are trivial as we can visually see them in the data model. However, the latter is somewhat a conceptual relationship handled by the **Sales** table. From a Star Schema standpoint, we do not call the **Sales** table a bridge table, but the principles remain the same. In data modeling using the Star Schema approach, a bridge table is a table created specifically for managing many-to-many relationships. The many-to-many relationships usually happen between two or more dimensions. However, there are some cases when two fact tables are involved in a many-to-many relationship. In data modeling in the Star Schema, the fact tables containing the foreign keys of the dimensions without any other additive values are called **factless fact** tables.

Our scenario already has a proper bridge table to satisfy the many-to-many relationship between the **Customer** table and the **Sales Reasons** table. We only need to create a relationship between the **Internet Sales** table and the **Internet Sales Reasons** table (the bridge). But we know that the xVelocity engine does not support composite keys for creating physical relationships. Therefore, we have to add a new column in both the **Internet Sales** table and the **Internet Sales Reasons** table, concatenating the **SalesOrderLineNumber** and the **SalesOrderNumber** columns. We can take care of the new column either in Power Query or DAX. For simplicity, we create the calculated column using the following DAX expressions.

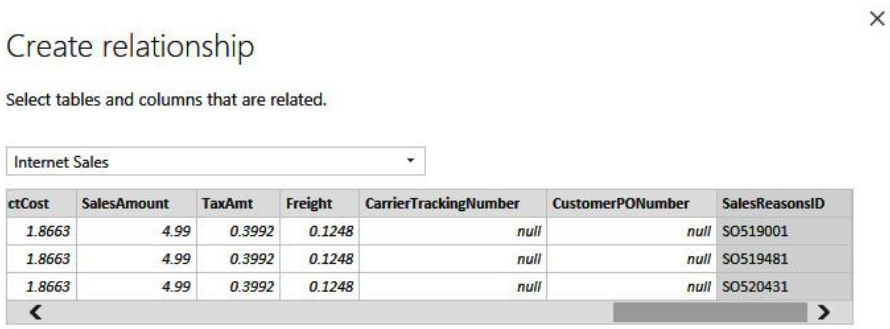
In the **Internet Sales** table, we use the following DAX expression to create a new calculated column:

```
SalesReasonsID = 'Internet Sales'[SalesOrderNumber] & 'Internet Sales'[SalesOrderLineNumber]
```

In the **Internet Sales Reason** table, we use the following DAX expression:

```
SalesReasonsID = 'Internet Sales Reasons'[SalesOrderNumber] & 'Internet Sales Reasons'[SalesOrderLineNumber]
```

Now that we have created the **SalesReasonsID** column in both tables, we create a relationship between the two tables. The following screenshot shows the **Create relationship** window to create a one-to-many relationship between the **Internet Sales** and the **Internet Sales Reasons** tables:



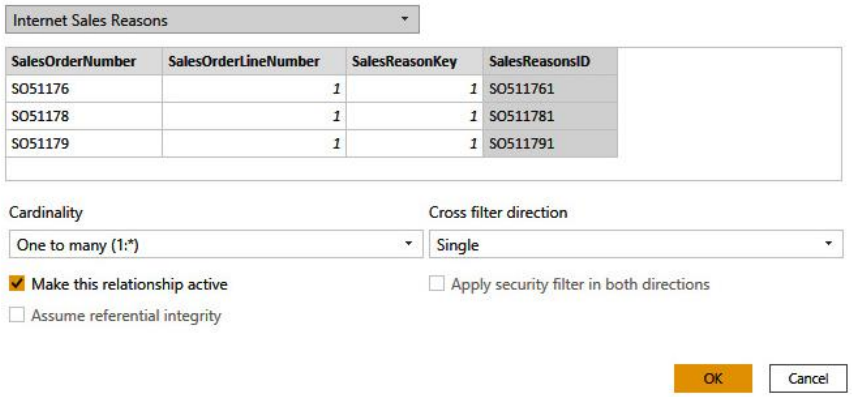


Figure 9.4 – Creating a relationship between the Internet Sales and the Internet Sales Reasons tables

The following diagram shows our data model after creating the preceding relationship:

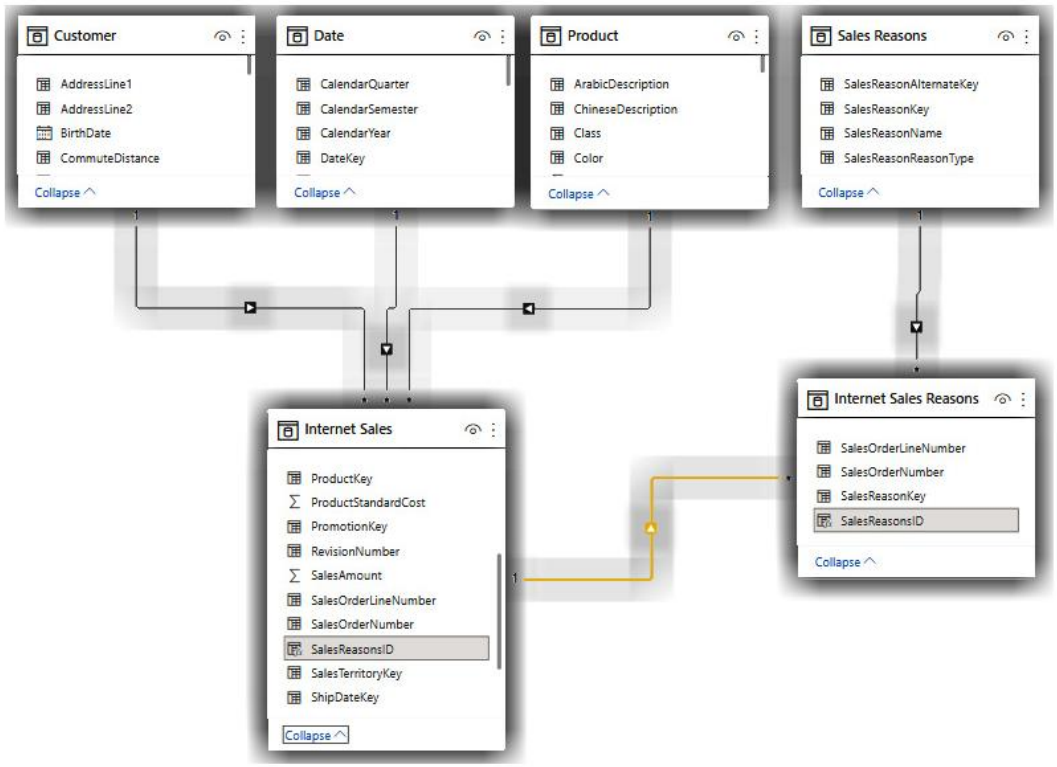


Figure 9.5 – The data model after creating a new relationship between the two tables

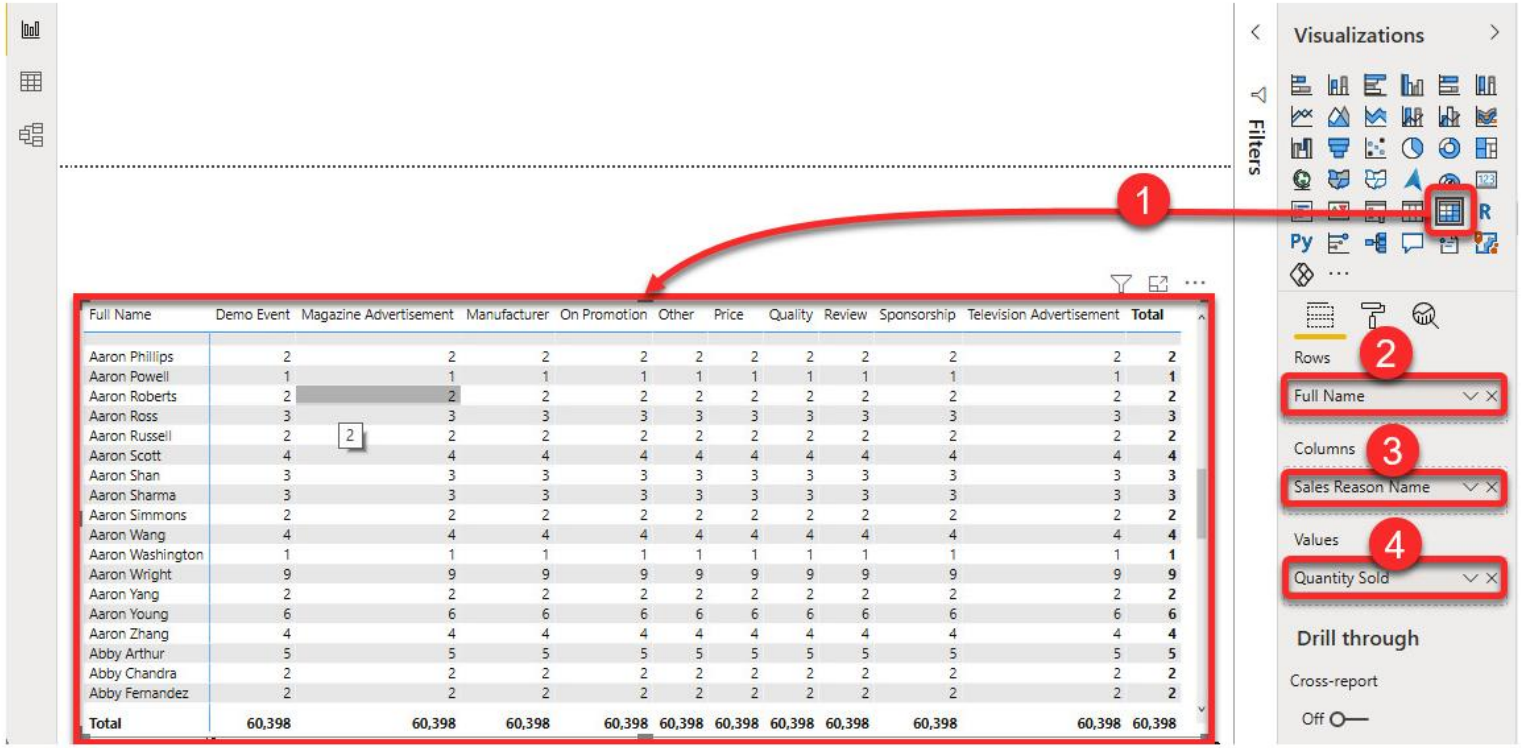
From a data modeling perspective, there is now a many-to-many relationship between the **`Internet Sales`** table and the **`Sales Reasons`** table via the bridge (the **`Internet Sales Reasons`**) table. Consequently, there is also a many-to-many relationship between the **`Customer`** table and the **`Sales Reason`** table. We can now visualize the data and see whether we can satisfy the business requirements to analyze customers' buying behavior for **`Quantity Sold`** over **`Sales Reasons`**. To visualize the data, we need to create a new measure to show **`Quantity Sold`** with the following DAX expression:

```
Quantity Sold = SUM('Internet Sales'[OrderQuantity])
```

Let's visualize the data and see how it works:

1. Put a **Matrix** visual on the reporting canvas.
2. Choose the **`Full Name`** column from the **`Customer`** table from the **Rows** dropdown.
3. Choose the **`Sales Reason Name`** column from the **`Sales Reasons`** table from the **Columns** dropdown.
4. Choose the **`Quantity Sold`** measure from the **Values** dropdown.

The following screenshot shows the preceding steps:



Full Name	Demo Event	Magazine Advertisement	Manufacturer	On Promotion	Other	Price	Quality	Review	Sponsorship	Television Advertisement	Total
Aaron Phillips	2	2	2	2	2	2	2	2	2	2	2
Aaron Powell	1	1	1	1	1	1	1	1	1	1	1
Aaron Roberts	2	2	2	2	2	2	2	2	2	2	2
Aaron Ross	3	3	3	3	3	3	3	3	3	3	3
Aaron Russell	2	2	2	2	2	2	2	2	2	2	2
Aaron Scott	4	4	4	4	4	4	4	4	4	4	4
Aaron Shan	3	3	3	3	3	3	3	3	3	3	3
Aaron Sharma	3	3	3	3	3	3	3	3	3	3	3
Aaron Simmons	2	2	2	2	2	2	2	2	2	2	2
Aaron Wang	4	4	4	4	4	4	4	4	4	4	4
Aaron Washington	1	1	1	1	1	1	1	1	1	1	1
Aaron Wright	9	9	9	9	9	9	9	9	9	9	9
Aaron Yang	2	2	2	2	2	2	2	2	2	2	2
Aaron Young	6	6	6	6	6	6	6	6	6	6	6
Aaron Zhang	4	4	4	4	4	4	4	4	4	4	4
Abby Arthur	5	5	5	5	5	5	5	5	5	5	5
Abby Chandra	2	2	2	2	2	2	2	2	2	2	2
Abby Fernandez	2	2	2	2	2	2	2	2	2	2	2
Total	**60,398**	**60,398**	**60,398**	**60,398**	**60,398**	**60,398**	**60,398**	**60,398**	**60,398**	**60,398**	**60,398**

Figure 9.6 – Visualizing customers' full name, sales reason, and quantity sold in a matrix

Looking at the results reveals an issue. The **Quantity Sold** values are repeated for **Sales Reason Name**. The reason is filter propagation: the **Full Name** filter flows from the **Customer** table to the **Internet Sales** table; therefore, the **Quantity Sold** values are calculated correctly for the customers.

However, the **Sales Reason Name** cannot get to the **Inter Sales** table as we currently set the **Cross filter direction** of the relationship between the **Internet Sales** table and the **Internet Sales Reasons** table to **Single**. The relationship between the latter tables is one-to-many. The **one** side table is the **Internet Sales** table, and the **many** side table is **Internet Sales Reasons**. Hence, the filters on the **Internet Sales** flow to the **Internet Sales Reasons** table but not the other way round. So we need to set the **Cross filter direction** of the relationship between the **Internet Sales** and the **Internet Sales Reasons** tables to **Both**. The following screenshot shows the preceding setting:

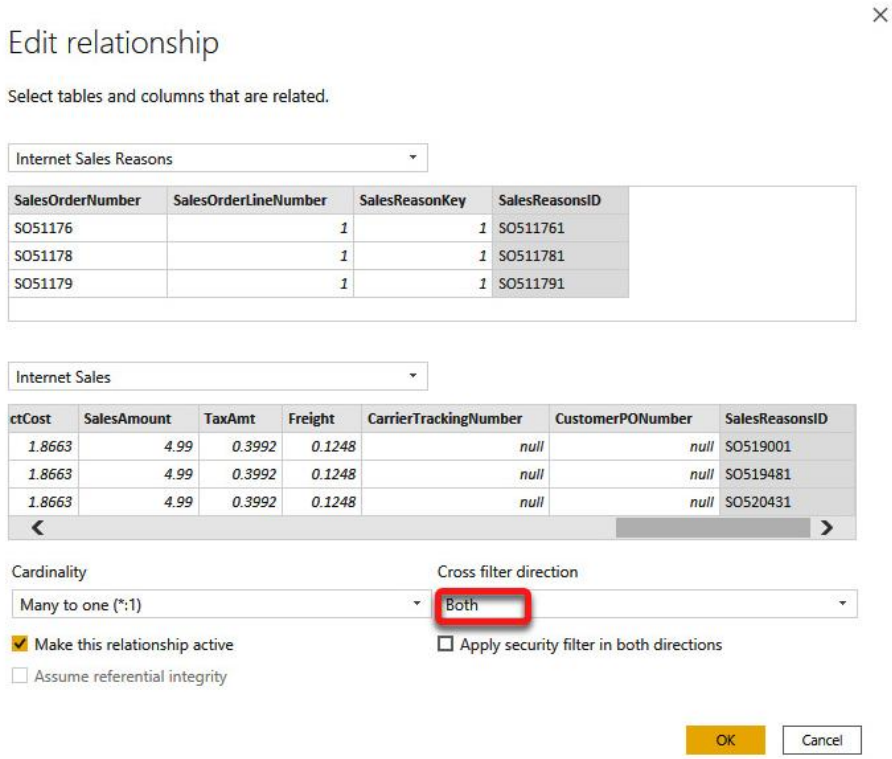


Figure 9.7 – Setting up a bidirectional relationship

When we switch back to the **Report** view, we see that the matrix shows correct results. The following screenshot shows the corrected results after setting the relationship to bidirectional:

Full Name	Manufacturer	On Promotion	Other	Price	Quality	Review	Television Advertisement	Total
Aaron Phillips				2				2
Aaron Powell				1				1
Aaron Roberts				2				2
Aaron Ross				3				3
Aaron Russell								2
Aaron Scott				4				4
Aaron Shan				3				3
Aaron Sharma				3				3
Aaron Simmons				2				2
Aaron Wang	1			3	1			4
Aaron Washington				1				1
Aaron Wright	1	1		5	1			9
Aaron Yang				2				2
Aaron Young	1			5	1			6
Aaron Zhang		4		4				4
Abby Arthur	1			4	1			5
Abby Chandra				2				2
Abby Fernandez				1			1	2
Total	**1,818**	**7,390**	**3,653**	**47,733**	**1,551**	**1,640**	**730**	**60,398**

Figure 9.8 – The correct results shown after setting the relationship to bidirectional

Looking at the **Total** row shows that the price is the most motivator for customers to buy the products with the largest **`Quantity Sold`** total value. But there is still something a bit confusing about the data: the **`Total`** value for each row doesn't make too much sense. Look at the highlighted row: the **`Total`** value is 4 while the data shows that the **`Total`** value should be 5. Here is the thing: the relationship between **`Customer`** and **`Sales Reasons`** is many-to-many, so there can be more than one reason for a customer to buy a product. In the highlighted row, Aaron Wang had at least more than one reason to buy a product. Let's analyze the situation in more detail. In the next few steps, we put another Matrix on the report, this time showing **`Quantity Sold`** by **`Product`** and **`Sales Reasons`**. Then we click a customer from the first matrix visual, which filters the second matrix. This way, we can identify for which product the customer had more than one reason to buy:

1. Put another **Matrix** on the report canvas.
2. Choose the **`Product Name`** column from the **`Product`** table in the **Rows** dropdown.
3. Choose the **`Sales Reason Name`** column from the **`Sales Reasons`** table in the **Columns** dropdown.
4. Choose **`Quantity Sold`** measure in the **Values** dropdown.
5. Click on the **Aaron Wang** row from the previous Matrix visual to cross-filter the new Matrix.

The following screenshot shows the results after clicking on the **Aaron Wang** row:

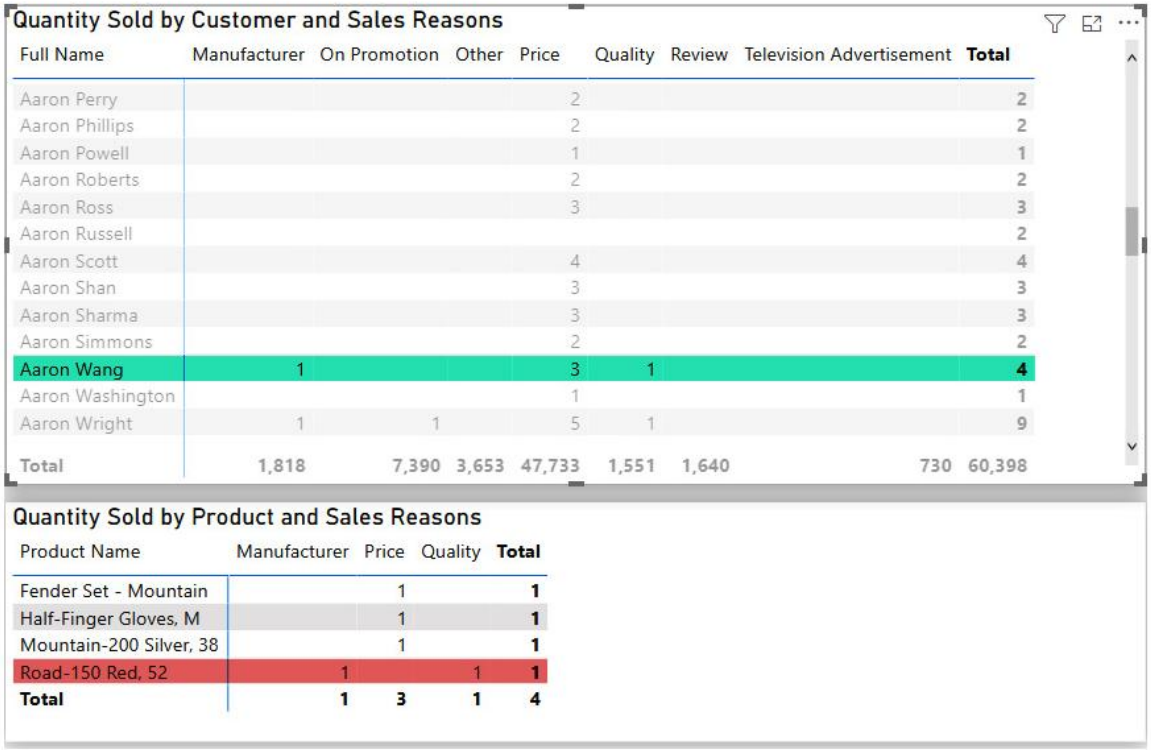


Figure 9.9 – Filtering Quantity Sold by Product and Sales Reasons with Customer

As the preceding screenshot shows, while Aaron has two reasons (**`Manufacturer`** and **`Quality`**) to buy the **Road-150 Red, 52**, he still bought only one item; therefore, the **`Total`** row shows **1** instead of **2**, which is correct. But this may confuse the end users if they do not have a complete understanding of the business. In that case, we have two options: to disable column subtotals, or to modify the **`Quantity Sold`** calculation to omit the **`Total`** values for **`Sales Reasons`**. The first option is working on the visual level, so we have to do it for each visual having a column from the **`Sales Reasons`** table, and the **`Quantity Sold`** measure, while the latter doesn't have a **`Total`** value. We can modify the **`Quantity Sold`** measure as follows:

```
Quantity Sold =
    IF(
        HASONEVALUE('Sales Reasons'[SalesReasonKey])
        , SUM('Internet Sales'[OrderQuantity])
        , BLANK()
        )
```

Hiding the bridge table

After we have implemented the many-to-many in our data model, it is good to hide the bridge table from the data model. We only have the bridge table in our data model as it carries the key columns of both tables participating in the many-to-many relationship. Hiding the bridge table also avoids confusion for other report creators who connect to our dataset to build the reports. To hide a table, we only need to switch to the **Model** view and click the hide/unhide ( ) button at the top right of the table. The following diagram shows the data model after hiding the **`Internet Sales Reasons`** table:

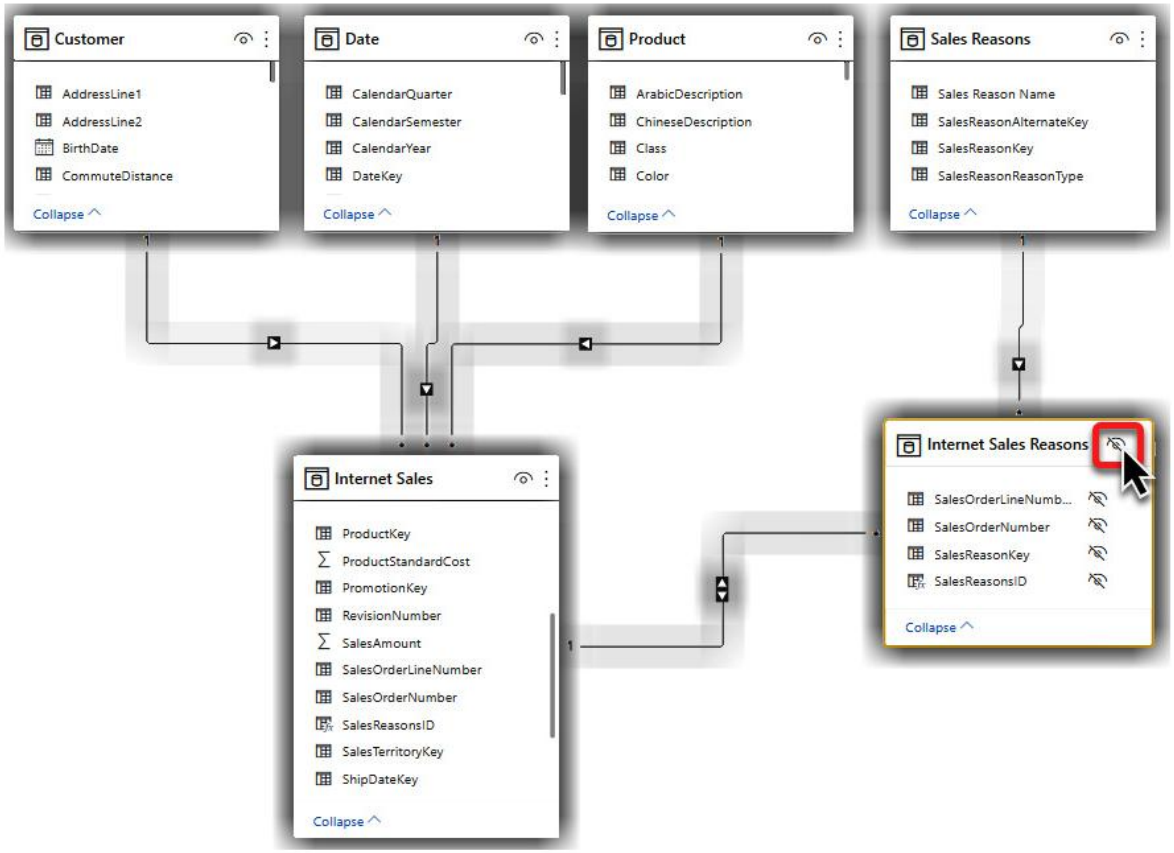


Figure 9.10 – Hiding bridge tables from the Model view

Being cautious with bidirectional relationships

One of the most misunderstood and somehow misused Power BI features in data modeling is setting the **Cross filter direction** to **Both**. This is widely known as a **bidirectional** relationship. There is nothing wrong with setting a relationship to bidirectional if we know what we are doing and are conscious of its effects on our data model. We have seen Power BI developers who have many bidirectional relationships in their model and consequently end up with many issues, such as getting unexpected results in their DAX calculations or being unable to create a new relationship due to ambiguity. The reason that overusing bidirectional relationships increases the model ambiguity lies in filter propagation. In *Chapter 8*, *Data Modeling Components*, we covered the concept of filter propagation as well as bidirectional relationships. *Chapter 8* looked at a scenario where the developer needed to have two slicers on a report page, one for the product category and another for filtering the customers. It is indeed a common scenario that the developers decide to set the relationships to bidirectional, which is no good. On many occasions, if not all, we can avoid creating a bidirectional relationship. Depending on the scenario, we may use different techniques. Let's look at the scenario we used in *Chapter 8*, *Data Modeling Components*, in the *Bidirectional relationships* section again. We solve the scenario where we have two slicers on the report page without making the relationship between the table bidirectional. The following diagram shows the data model:

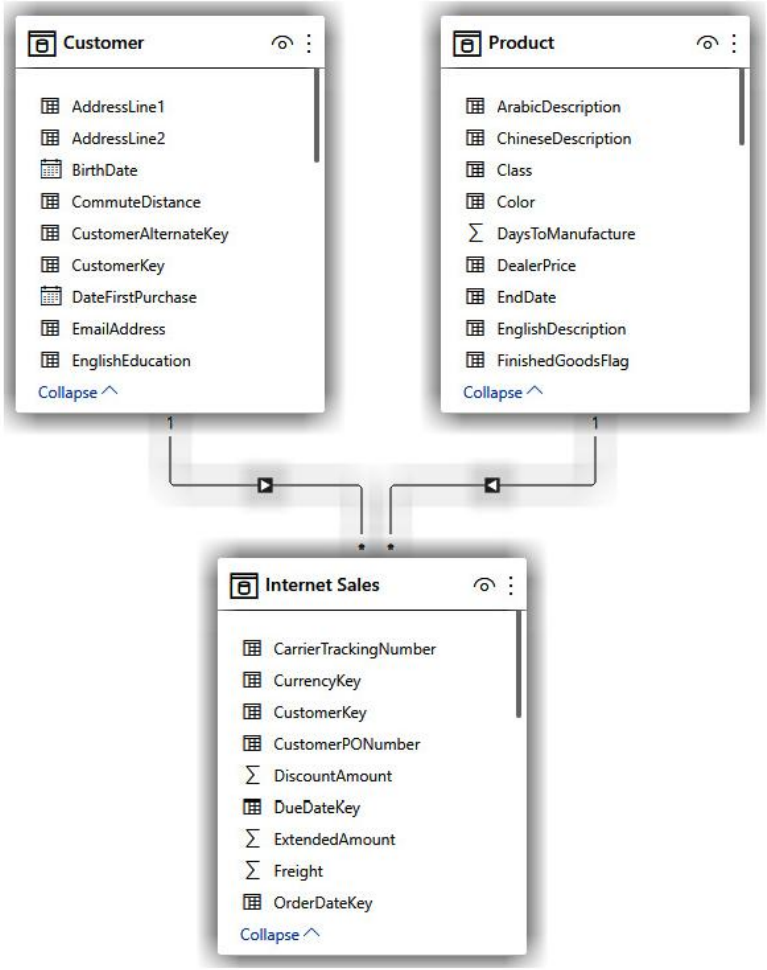


Figure 9.11 – Internet Sales data model

The following diagram shows the reporting requirements:

Figure 9.12 – The Customer Name slicer filters the Sales data but not the Product Category data

As the preceding diagram shows, the **`Customer Name`** slicer filters **`Internet Sales`**. Still, the filter does not propagate to the **`Product Category`** table as the **Cross filter direction** of the one-to-many relationship between the **`Product Category`** table and the **`Internet Sales`** table is set to single; therefore, the filters flow from the **`Product Category`** table to the **`Internet Sales`** table but not the way round. So, let's solve the problem without physically changing the **Cross filter direction** of the relationships. The way to solve the scenario is to set the relationships to bidirectional programmatically using the **`CROSSFILTER()`** function in DAX. The following measure is a modified version of the **`Internet Sales`** measure, where we programmatically made the relationships between the **`Product`** table, the **`Internet Sales`** table, and the **`Customer`** table bidirectional:

```
Internet Sales Bidirectional =
    CALCULATE(
        SUM('Internet Sales'[SalesAmount])
        , CROSSFILTER(Customer[CustomerKey], 'Internet Sales'[CustomerKey], Both)
        , CROSSFILTER('Product'[ProductKey], 'Internet Sales'[ProductKey], Both)
        )
```

Now we use the new measure instead of the **`Internet Sales`** measure in the **Matrix** visual. We also need to add the new measure in the **visual filters** on both slicers and set the filter's **value** to **is not blank**. The following diagram shows the preceding process:

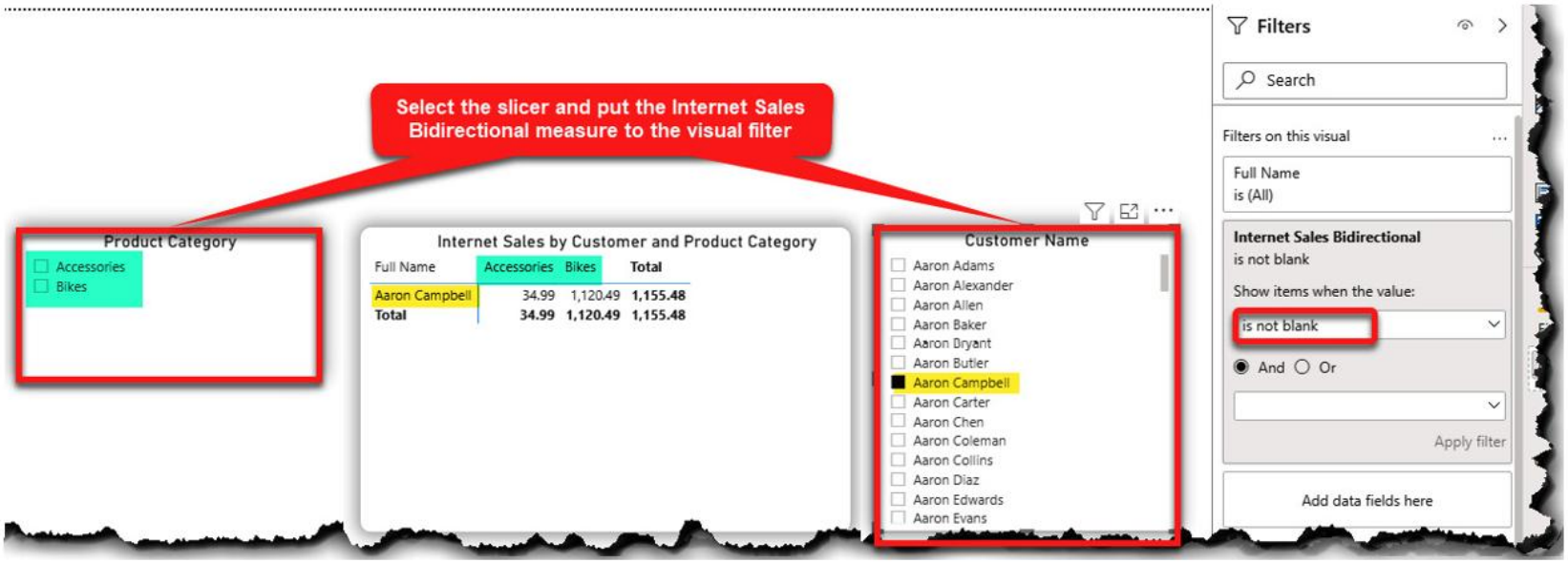


Figure 9.13 – Using a measure in the visual filter for slicers

As the preceding diagram shows, the **`Customer Name`** slicer is successfully filtering the **`Product Category`** slicer and vice versa. The following steps show how it works on the **`Customer Name`** slicer; the same happens on the **`Product Category`** slicer:

1. The slicer gets the list of all customers from the **`Full Name`** column.
2. The visual filter kicks in and applies the filters. The **Internet Sales Bidirectional** measure used in the filter forces the slicer visual to run the measure and omit the blank values.
3. The **Internet Sales Bidirectional** measure forces the relationships between the **`Product`** table, the **`Internet Sales`** table, and the **`Customer`** table to be bidirectional for the duration that the measure runs.

If we do not select anything on the slicers, then both slicers show the values having at least one row within the **`Internet Sales`** table.

The key message here is that you do not use bidirectional relationships unless you know what you are doing. In some cases, omitting the bidirectional relationship makes the DAX expressions too complex and hence not performant. You, as the data modeler, should decide which method works the best for your model.

Dealing with inactive relationships

In real-world scenarios, the data models can get very busy, especially when we are creating a data model to support enterprise BI; there are many instances that we have an inactive relationship in our data model. In the majority of cases, there are two reasons that a relationship is inactive, as follows:

- The table with an inactive relationship is reachable via multiple filter paths.
- There are multiple direct relationships between two tables.

In both preceding cases, the engine does not allow to activate an inactive relationship to avoid **ambiguity** across the model.

Reachability via multiple filter paths

A filter path between two tables is when the two tables are related via multiple tables. Therefore, the filter propagates from one table to the other via multiple hops (relationships). The following diagram shows a data model with an inactive relationship:

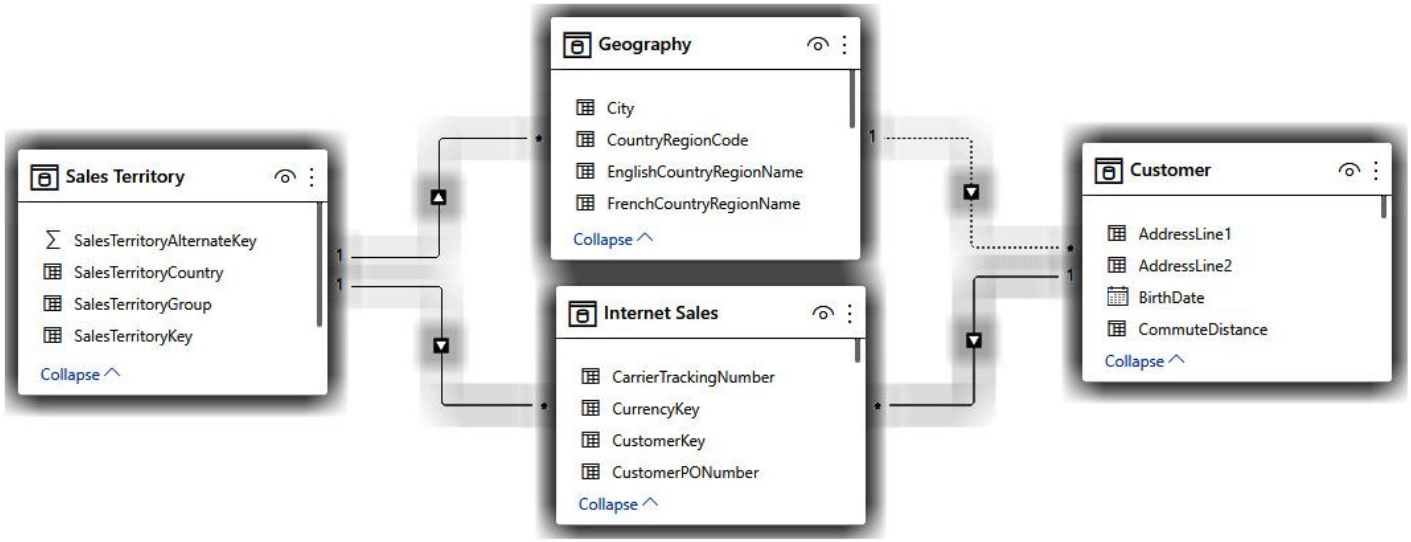


Figure 9.14 – Inactive relationship due to multiple paths detected

As the preceding diagram shows, there is already a relationship between the **Sales Territory** table and the **Internet Sales** table. Power BI raises an error message indicating that activating the relationship between the **Geography** table and **Customer** table will cause ambiguity in between **Sales Territory** and **Internet Sales** if we attempt to activate the inactive relationship. The following diagram illustrates how the **Internet Sales** table would be reachable through multiple filter paths:

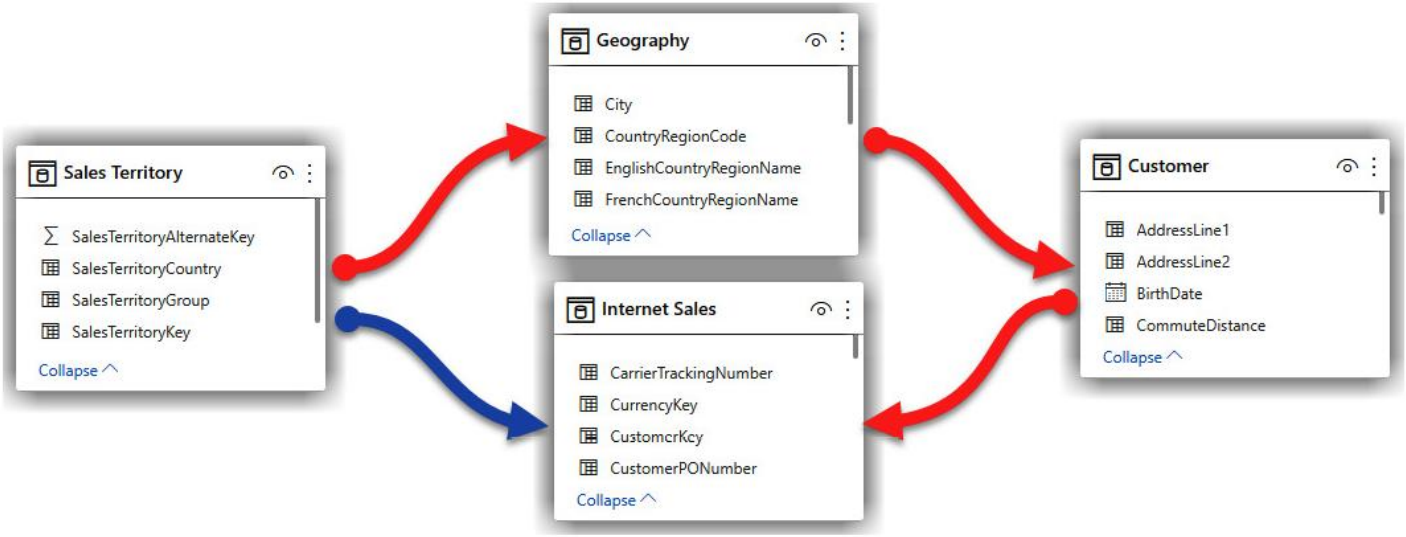


Figure 9.15 – Internet Sales reachability through multiple paths

Looking at the preceding diagram shows how the **Internet Sales** table is reachable via multiple paths. The following steps show this happens when we put a filter on the **Sales Territory** table:

1. The filter propagates to the **Geography** table via the relationship between **Sales Territory** and **Geography**.
2. The filter then propagates again to the **Customer** table through the relationship between **Geography** and **Customer**.
3. The filter propagates one more time, reaching **Internet Sales** via the relationship between **Customer** and **Internet Sales**.

The **Sales Territory** table and the **Internet Sales** table are related through two hops in a filter path. The red arrows in *Figure 9.15* show a filter path between **Sales Territory** and **Internet Sales**. But there is another filter path between the two tables, which is shown by a purple arrow in *Figure 9.15*. It is now clear why the relationship between **Geography** and **Customer** is inactive.

Multiple direct relationships between two tables

The other common cause of having an inactive relationship is when there is more than one direct relationship between two tables. Having multiple relationships means that we can use each relationship for a different analytical calculation. The following diagram shows that the **Date** table is related to the **Internet Sales** table via several relationships:

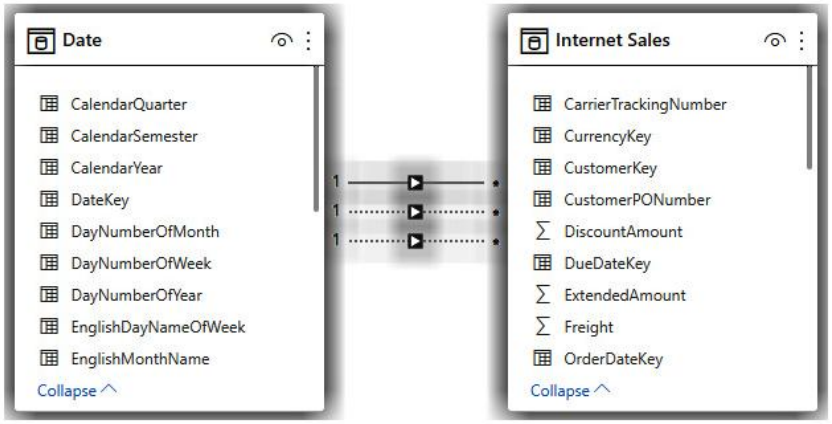


Figure 9.16 – Two tables with multiple direct relationships

We can look at the **Manage relationships** window to see what those relationships are. The following screenshot shows the **Manage relationships** window:

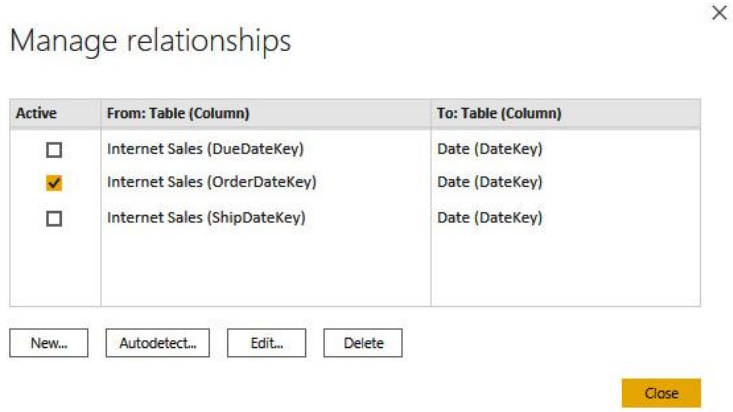


Figure 9.17 – The Date table and the Internet Sales table are related via multiple relationships

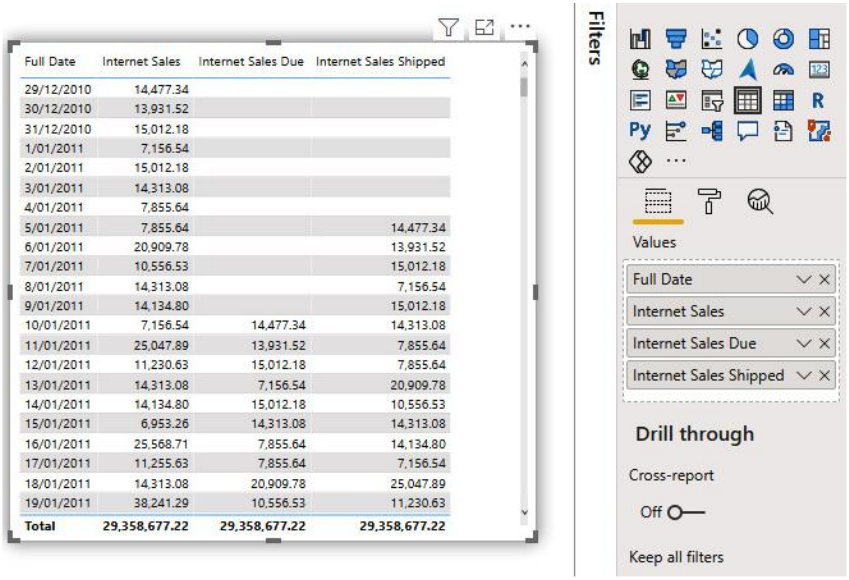
As the preceding screenshot shows, there are three columns in the **Internet Sales** table participating in the relationships, and all of them are legitimate. Each relationship filters the **Inter Sales** table differently, but we can have only one active relationship between two tables at a time. Currently, the relationship via the **OrderDateKey** column from the **Internet Sales** table and the **DateKey** column from the **Date** table is the active relationship that propagates the filter from the **Date** table to the **Internet Sales** table. This behavior means that when we use the **Year** column from the **Date** table and the **Internet Sales** measure from the **Internet Sales** table, we are slicing the **Internet Sales** by **order date year**. But what if the business needs to analyze the **Internet Sales** by **Due Date**? What if the business also needs to analyze the **Internet Sales** by **Ship Date**? We obviously cannot physically make a relationship active and inactive to solve this issue. We have to solve the problem programmatically using the **USERELATIONSHIP()** function in DAX. The **USERELATIONSHIP()** function activates an inactive relationship for the duration that the measure is calculating. So to meet the preceding business requirements, we can create two new measures. The following DAX expression activates the **DueDateKey** -> **DateKey** relationship:

```
Internet Sales Due =
    CALCULATE([Internet Sales]
        , USERELATIONSHIP('Internet Sales'[DueDateKey], 'Date'[DateKey])
    )
```

The following DAX expression is to activate the **ShipDateKey** -> **DateKey** relationship:

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

Let's use the new measure side by side with the **Internet Sales** measure and the **Full Date** column from the **Date** table in a table to see the differences between values:



Full Date	Internet Sales	Internet Sales Due	Internet Sales Shipped
29/12/2010	14,477.34		
30/12/2010	13,931.52		
31/12/2010	15,012.18		
1/01/2011	7,156.54		
2/01/2011	15,012.18		
3/01/2011	14,313.08		
4/01/2011	7,855.64		
5/01/2011	7,855.64		14,477.34
6/01/2011	20,909.78		13,931.52
7/01/2011	10,556.53		15,012.18
8/01/2011	14,313.08		7,156.54
9/01/2011	14,134.80		15,012.18
10/01/2011	7,156.54	14,477.34	14,313.08
11/01/2011	25,047.89	13,931.52	7,855.64
12/01/2011	11,230.63	15,012.18	7,855.64
13/01/2011	14,313.08	7,156.54	20,909.78
14/01/2011	14,134.80	15,012.18	10,556.53
15/01/2011	6,953.26	14,313.08	14,313.08
16/01/2011	25,568.71	7,855.64	14,134.80
17/01/2011	11,255.63	7,855.64	7,156.54
18/01/2011	14,313.08	20,909.78	25,047.89
19/01/2011	38,241.29	10,556.53	11,230.63
Total	**29,358,677.22**	**29,358,677.22**	**29,358,677.22**

Figure 9.18 – Activating inactive relationships programmatically

Using configuration tables

There are many cases when the business needs to analyze some of the business metrics in clusters. Some good examples are analyzing sales by unit price range, analyzing sales by product cost range, analyzing customers by their age range, or analyzing customers by commute distance. In all of the preceding examples, the business does not need to analyze constant values; instead, it is more about analyzing a metric (sales, in the preceding examples) by a range of values.

Some other cases are related to data visualization, such as dynamically changing the color of values when they are in a specific range. An example can be to change the color of values in all visuals analyzing sales to red if the sales value for the data points is less than the average sale over time. This is a relatively advanced analysis that can be reused in our reports that keeps the consistency of our data visualization.

For all of the preceding examples, we need to define configuration tables. In the latter example, we will see how data modeling can positively affect our data visualization.

Segmentation

As stated earlier, there are cases when the business needs to analyze their business metrics by clusters of data. This type of analysis is commonly known as **segmentation**, as we are analyzing the business values of different segments. Let's continue with an example. The business needs to analyze **`Internet Sales`** by **`UnitPrice`** ranges. To be able to do the analysis, we need to have the definition of unit price ranges. The following list shows the definition of unit price ranges:

- **Low**: When the **`UnitPrice`** is between $**`0`** and $**`50`**
- **Medium**: When the **`UnitPrice`** is between $**`51`** and $**`450`**
- **High**: When the **`UnitPrice`** is between $**`451`** and $**`1,500`**
- **Very high**: When the **`UnitPrice`** is greater than $**`1,500`**

At this point, you may think of adding a calculated column to the **`Internet Sales`** table to take care of the business requirement. That is right, but what if the business needs to modify the definition of unit price ranges several times in the future? We need to frequently modify the calculated column, which does not sound like a viable option. A better option is to have the definition of unit price ranges in a table. We can store the definition in an Excel file accessible in a shared **`OneDrive for Business`** folder. It can be a SharePoint List that is accessible to the business to make any necessary changes. For simplicity, we manually enter the preceding definition as a table using the **Enter data** feature in Power BI.

NOTE

*We do not recommend entering the definition values manually in Power BI using the **Enter data** feature in your real-world scenarios. Suppose the business needs to modify the values. In that case, we have to modify the report in Power BI Desktop and republish the report to the Power BI service.*

The following screenshot shows a **`Unit Price Ranges`** table created in Power BI:

Sort	Price Range	From	To
4	Low	0	50
3	Medium	51	450
2	High	451	1500
1	Very hight	1501	15000

Figure 9.19 – Unit Price Ranges table

Now that we have the definitions data available in Power BI, we need to add a calculated column in the **Internet Sales** table. The new calculated column looks up the **Price Range** value for each **UnitPrice** value within the **Internet Sales** table. To do so, we have to compare the **UnitPrice** value of each row from the **Internet Sales** table with the values of the **From** and **To** columns from the **Unit Price Ranges** table. The following DAX expressions cater for that:

```
Price Range =
    CALCULATE(
        VALUES('Unit Price Ranges'[Price Range])
        , FILTER('Unit Price Ranges'
            , 'Unit Price Ranges'[From] < 'Internet Sales'[UnitPrice]
                && 'Unit Price Ranges'[To] >= 'Internet Sales'[UnitPrice]
            )
    )
```

The following screenshots show how we can now quickly analyze the **Internet Sales** measure by **Price Range**:

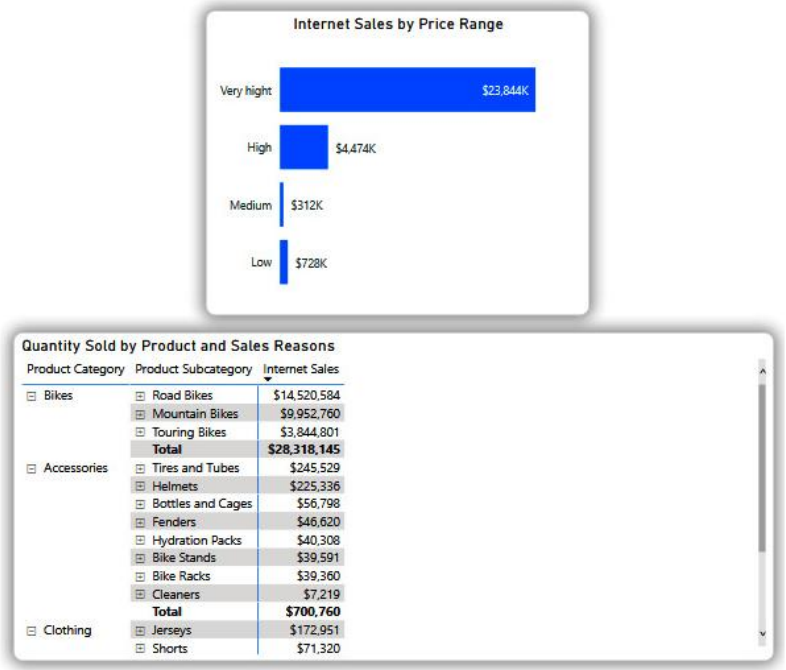


Figure 9.20 – Analyzing Internet Sales by price range

As a minor note, look at how the bars are sorted in the bar chart. They are not alphabetically sorted by **Price Range** name nor by the **Internet Sales** values. You already learned how to manage column sorting in the previous chapter, so I leave this to you to find out how that is possible.

We can now click on a bar within the bar chart and see what products are falling into a selected price range.

Dynamic conditional formatting with measures

So far, we have discussed many aspects of data modeling, as the primary goal of this book is to learn how to deal with various day-to-day challenges. This section discusses an essential aspect of data visualization, **color coding**, and how data modeling can ease many advanced data visualization challenges. Color coding is one of the most compelling and efficient ways to provide pertinent information about the data. In this section, we make a bridge between data modeling and data visualization.

We could color code the visuals from the day that Power BI was first born. However, conditional formatting was not available on many visuals for a long time. Luckily, we can now set conditional formatting on almost all default visuals (and many custom visuals) in Power BI Desktop. Let's continue with a scenario.

The business decided to use predefined color codes and use them dynamically in various visuals so that the visuals' colors are picked depending on the value of the **`Sales MoM%`** measure. The **`Sales MoM%`** measure calculates the percentage of sales changes based on the sales values for each year in comparison with the sales values for the previous year. The goal is to visualize **`Sales MoM%`** in a **`Clustered column chart`**. The color for each data point should be calculated based on a config table. The following diagram shows the structure of the config table:

Index	ColourHex	Range%	Description
1	#264653	100%	Devine
2	#287271	90%	Excellent
3	#2A9D8F	80%	Very good
4	#8AB17D	70%	Good
5	#E9C46A	60%	Fine
6	#DA941B	50%	Normal
7	#CB7C15	40%	Keep going
8	#EF7A1A	30%	Low
9	#CB4F15	20%	Very low
10	#7E2711	10%	Critical

Figure 9.21 – Config table defining color codes

1. First of all, we need to enter the data in the preceding diagram into a table in Power BI. We name the new table **ConfigColour**.

   The following screenshot shows the **ConfigColour** table in Power BI Desktop:

ColourHex	Index	Range%	Status
#261653	1	100%	Devine
#2872A1	2	90%	Excellent
#2A9D8F	3	80%	Very good
#8A317D	4	70%	Good
#E9C46A	5	60%	Fine
#DA941B	6	50%	Normal
#CB7C15	7	40%	Keep going
#FF7A1A	8	30%	Low
#CB4F15	9	20%	Very low
#7E2711	10	10%	Critical

Figure 9.22 – The ConfigColour table

2. Now we need to create a **Sales MoM%** measure. The following DAX expression calculates sales for last month (**Sales LM**):

```
Sales LM =
    CALCULATE([Internet Sales]
        , DATEADD('Date'[Full Date], -1, MONTH)
        )
```

3. After we have calculated **Sales LM**, we just need to calculate the percentage of differences between the **Internet Sales** measures and the **Sales LM** measure. The following expression caters for that:

```
Sales MoM% = DIVIDE([Internet Sales] - [Sales LM], [Sales LM])
```

4. The next step is to create two textual measures. The first measure picks a relevant value from the **ColourHex** column, and the other one picks the relevant value from the **Status** column from the **ConfigColour** table. Both textual measures pick their values from the **ConfigColour** table based on the value of the **Sales MoM%** measure.

Before we implement the measures, let's understand how the data within the **ConfigColour** table supposed to work. The following points are essential to understand how to work with the **ConfigColour** table:

- The **ConfigColour** contains 10 rows of data.
- The **ColourHex** contains hex codes for colors.
- The **Range%** contains a decimal number between 0.1 and 1.
- The **Status** column contains a textual description for each color.
- The **Index** column contains the table index.

The preceding points look easy to understand, but the **Range%** column is a bit tricky. When we format the **Range%** column with percentage, then values are between 10% and 100%. Each value, however, represents a range of values, not a constant value. For instance, 10% means all values from 0% up to 10%. In the same way, 20% means all values between 11% and 20%. The other point to note is when we format the **Range%** values with percentage, each value is divisible by 10 (such as 10, 20, 30,…).

The new textual measures pick the relevant values either from the **ColourHex** column or from the **Status** column based on the **Range%** column and the **Sales MoM%** measure. So we need to identify the ranges the **Sales MoM%** values fall in, then compare them with the values within the **ColourHex** column. The following formula guarantees that the **Sales MoM%** values are divisible by **10**, so we can later find the matching values within the **ColourHex** column:

```
CONVERT([Sales MoM%] * 10, INTEGER)/10
```

Here is how the preceding formula works:

1. We multiply the value of **Sales MoM%** by **10**, which returns a decimal value between **0** and **10** (we will deal with the situations when the value is smaller than **0** or bigger than **10**).
2. We convert the decimal value to an integer to drop the digits after the decimal point.
3. Finally, we divide the value by **10**.

When we format the results in percentage, the value is divisible by **10**. We then check whether the value is smaller than 10%, we return 10%, and if it is bigger than 100%, we return 100%.

It is now time to create textual measures. The following DAX expression results in a hex color. We will then use the retrieved hex color in the visual's conditional formatting:

```
Sales MoM% Colour =
var percentRound = CONVERT([Sales MoM%] * 10, INTEGER)/10
var checkMinValue = IF(percentRound < 0.1, 0.1, percentRound)
var checkMaxValue = IF(checkMinValue > 1, 1, checkMinValue)
return
CALCULATE(
    VALUES(ConfigColour[ColourHex])
    , FILTER( ConfigColour
        , 'ConfigColour'[Range%] = checkMaxValue
        )
    )
```

As you can see in the preceding expression, the **checkMinValue** and **checkMaxValue** variables are adjusting the out-of-range values. The following DAX expression results in a description calculated in a similar way to the previous measure:

```
Sales MoM% Description =
var percentRound = CONVERT([Sales MoM%] * 10, INTEGER)/10
var checkMinValue = IF(percentRound < 0.1, 0.1, percentRound)
var checkMaxValue = IF(checkMinValue > 1, 1, checkMinValue)
return
CALCULATE(
    VALUES(ConfigColour[Status])
    , FILTER( ConfigColour
        , 'ConfigColour'[Range%] = checkMaxValue
        )
    )
```

Now that we have created the textual measures, we can use them to format the visuals conditionally (if supported). The following visuals currently support conditional formatting:

Stacked Column Chart	Clustered Bar Chart	Clustered Column Chart	100% Stacked Bar Chart
100% Stacked Column Chart	Line and Stacked Column Chart	Line and Clustered Column Chart	Ribbon Chart
Funnel Chart	Scatter Chart	Treemap Chart	Gauge
Card	KPI	Table	Matrix

Figure 9.23 – Power BI default visuals supporting conditional formatting

The following steps show how to format a clustered column chart conditionally:

1. Put a **`Clustered column chart`** on a new report page.
2. Choose the **`Year-Month`** column from the **`Date`** table from the visual's **Axis** dropdown.
3. Choose the **`Sales MoM%`** measure from the visual's **Values** dropdown.
4. Choose the **`Sales MoM% Description`** measure from the **Tooltips** dropdown.
5. Switch to the **Format** tab from the **Visualizations** pane.
6. Expand the **Data colors**.
7. Click the **fx** button.
8. Select **Field value** from the **Format by** drop-down menu.
9. Select the **`Sales MoM% Colour`** measure from the **Based on field** menu.
10. Click **OK**.

The following screenshot shows the preceding steps:

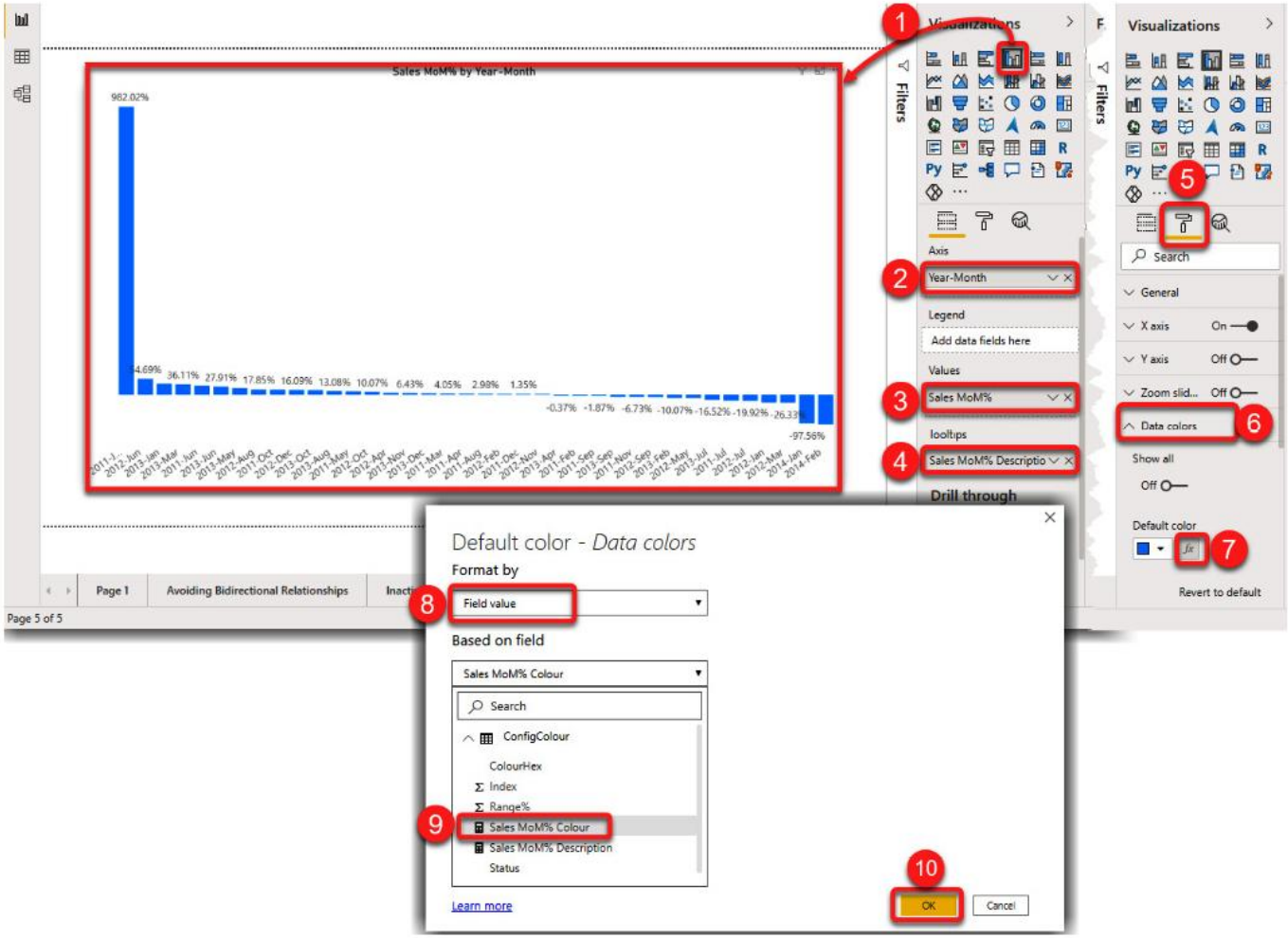


Figure 9.24 – Applying dynamic conditional formatting on a clustered column chart

The following screenshot shows the clustered column chart after the preceding settings:

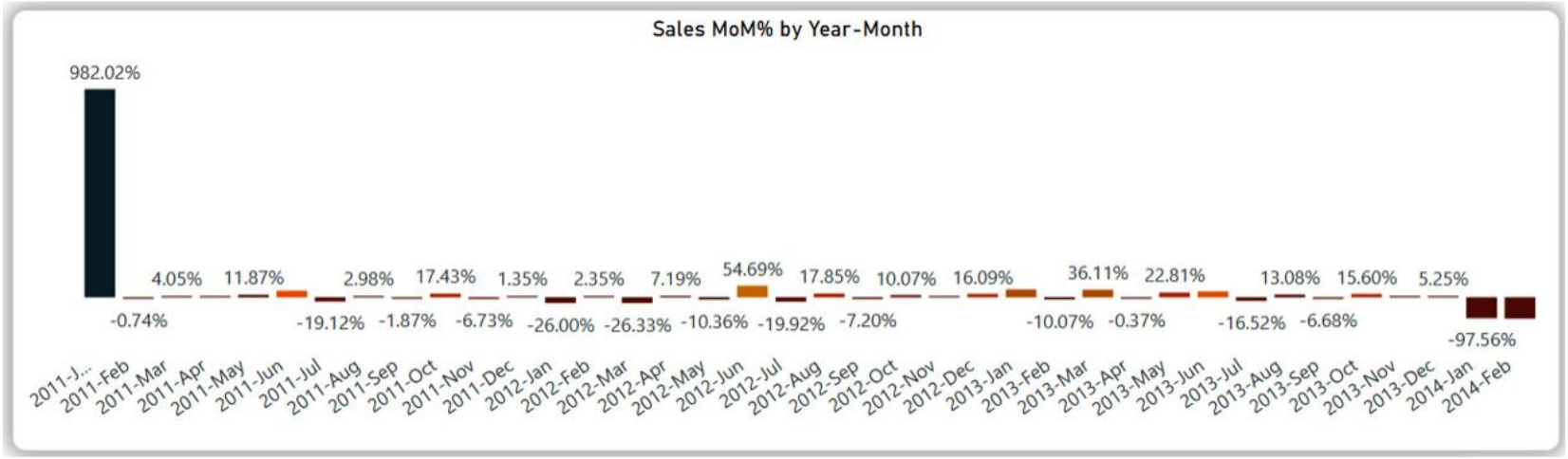


Figure 9.25 – The clustered column chart after it has been conditionally formatted

With this technique, we can create compelling data visualizations that can quickly provide many insights about the data. For instance, the following screenshot shows a report page analyzing sales by date. I added a matrix showing **`Internet Sales`** by **`Month`** and **`Day`**:

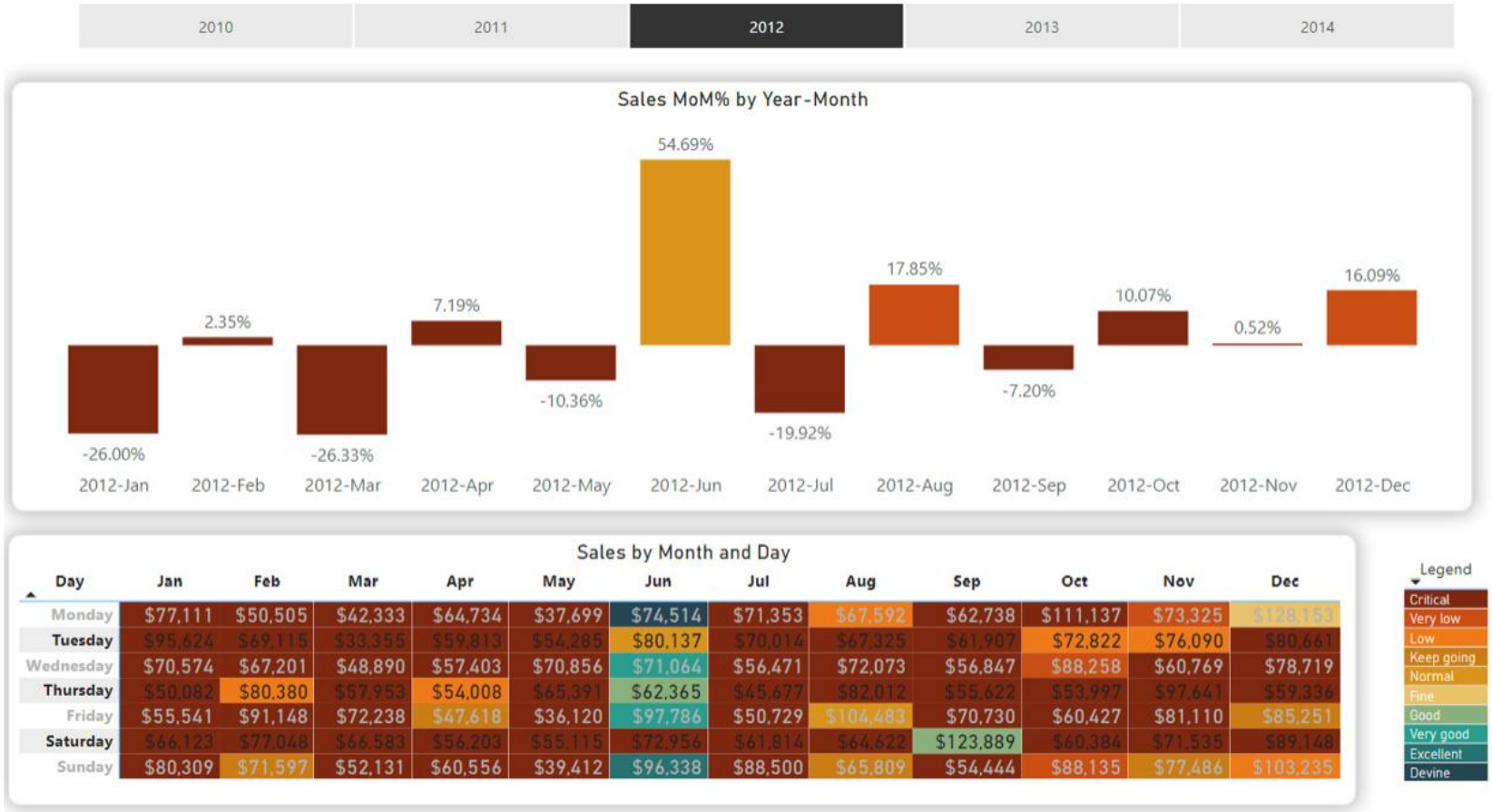


Figure 9.26 – Color-coded Sales report

I used the **Sales MoM% Colour** measure to color code the Matrix. In the preceding report, every cell of the Matrix shows the sales amount by weekday for the entire month. However, the color of the cell shows the comparison of current sales against the same weekday last month. For instance, if we look at the **2012-Jun** sales, we quickly see that the **Monday** sales were excellent compared to **2011-Jun** (colored in dark blue).

The preceding report page might not be a perfect example of a high-standard data visualization. However, without a doubt, it provides many more insights than a similar report page without color coding. In real-world scenarios, we might have some more colors to show the severity of the metric we are analyzing for negative numbers. I leave this to you as an exercise to use this technique to create very professional-looking reports.

Avoiding calculated columns when possible

The ability to create calculated columns is one of the most essential and powerful features in DAX. Calculated columns, as the name suggests, are computed based on a formula; therefore, the calculated column values are not available either in the source systems or in the Power Query layer. The values of the calculated columns are computed during the data refresh and then stored in memory. It is important to note that the calculated columns reside in memory unless we unload the whole data model from memory, which in Power BI means when we close the file in Power BI Desktop or switch to other contents in the Power BI service. Calculated columns, after creation, are just like any other columns, so we can use them in other calculated columns, measures, calculate tables, or for filtering the visualization layer. A common approach between developers is to use calculated columns to divide complex equations into smaller chunks. That is precisely the point when we suggest stopping excessive use of calculated columns. The general rules of thumb for using calculated columns are as follows:

- Create a calculated column if you are going to use it in filters.
- Even though you need to use the calculated column in filters, consider creating the new column in the Power Query layer when possible.
- Do not create calculated columns if you can create a measure with the same results.
- Always think about the data cardinality when creating calculated columns. The higher the cardinality, the lower the compression and the higher memory consumption.
- Always have a firm justification for creating a calculated column, especially when you are dealing with large models.
- Use the **View Metrics** tool in **DAX Studio** to monitor the size of the calculated column, which directly translates to memory consumption.

Let's look at an example in this chapter's sample file. The business needs to calculate **Gross Profit**. To calculate **Gross Profit**, we have to deduct total costs from total sales. We can create a calculated column with the following DAX expression that gives us **Gross Profit** for each row of the **Internet Sales** table:

```
Gross Profit = 'Internet Sales'[SalesAmount] - 'Internet Sales'[TotalProductCost]
```

We can then create the following measure to calculate **Total Gross Profit**:

```
Total Gross Profit with Calc Column = SUM('Internet Sales'[Gross Profit])
```

Let's have a look at the preceding calculated column in **DAX Studio** to get a better understanding of how it performs. Perform the following steps using **View Metrics** in DAX Studio:

1. Click the **External Tools** tab from the ribbon.
2. Click **DAX Studio**.
3. In DAX Studio, click the **Advanced** tab from the ribbon.
4. Click **View Metrics**.
5. Expand the **Internet Sales** table.
6. Find the **Gross Profit** column.

The following screenshot shows the preceding steps:

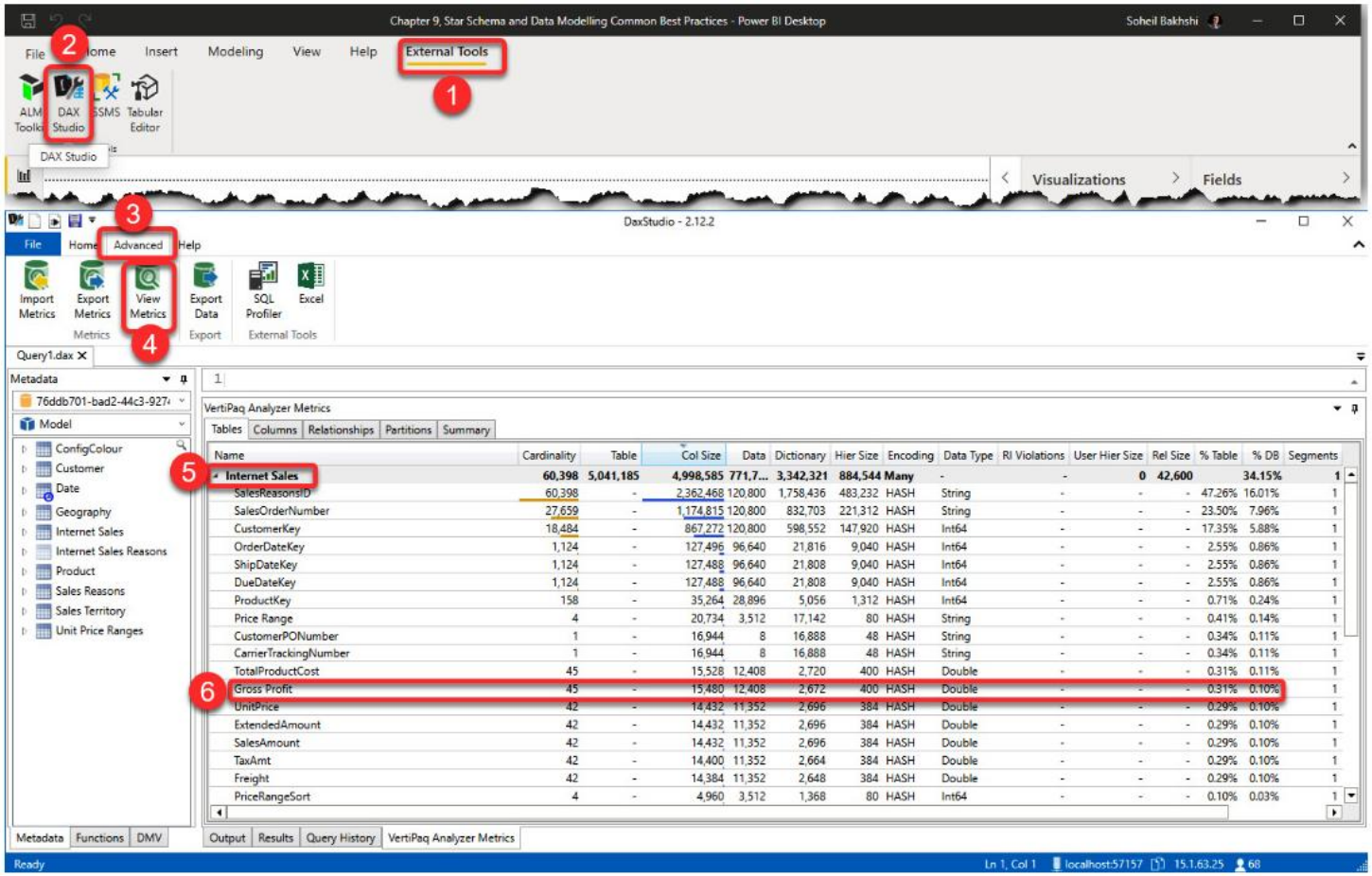


Figure 9.27 – View Metrics in DAX Studio

As you see in the preceding screenshot, the **Gross Profit** column size is **15,480** bytes (approximately 15 KB) with the cardinality of 45 consuming 0.31% of the table size. The **Internet Sales** table is a small table with 60,398 rows. So we can imagine how the column size can grow in larger tables.

While the process of creating a calculated column then getting the summation of the calculated column is legitimate, it is not the preferred method. We can compute **Total Gross Profit** in a measure with the following DAX expression:

```
Total Gross Profit Measure = SUMX('Internet Sales', 'Internet Sales'[SalesAmount] - 'Internet Sales'[TotalProductCost])
```

The difference between the two approaches is that the values of the **Gross Profit** calculated column computed at the table refresh time. It resides in memory, while its measure counterpart aggregates the gross profit when we use it in a visual. So when we use the **Product Category** column from the **Product** table in a **clustered column chart** and the **Total Gross Profit Measure**, the values of the **Total Gross Profit Measure** are aggregated for the number of product categories in memory. The following screenshot shows the **Total Gross Profit Measure** by **Product Category** in a **clustered column chart**:

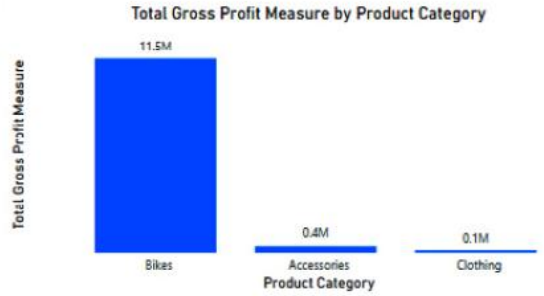


Figure 9.28 – Total Gross Profit Measure by Product Category in a clustered column chart

As the preceding screenshot shows, the **Total Gross Profit Measure** is aggregated in the **Product Category** level with only three values, so the calculation is superfast with minimal memory consumption.

Organizing the model

There are usually several roles involved in a Power BI project in real-world enterprise BI scenarios. From a Power BI development perspective, we might have data modelers, report writers, quality assurance specialists, support specialists, and so on. The data modelers are the ones who make the data model available for all other content creators, such as report writers. So, it is essential to make a model that is as organized as possible. In this section, we look at several ways to organize our data models.

Hiding insignificant model objects

One of the essential ways to keep our model tidier is to hide all insignificant objects from the data model. In many cases, we have some objects in the data model that are not used anywhere else. However, we cannot remove them from the data model as we may require them in the future. So, the best practice is to hide all those objects unless they are going to serve a business requirement. In the following few sections, we discuss the best candidate objects for hiding in our data model.

Hiding unused fields and tables

There are many cases when we have some fields (columns or measures) or tables in the data model that are not used anywhere else. Unused fields are the measures or columns that fulfil the following criteria:

- Are not used in any visuals in any report pages
- Are not used within the **`Filters`** pane
- No measures, calculated columns, calculated tables, or calculation groups referencing those fields
- No roles within the row level security reference those fields

If we have some fields falling in all of the preceding categories, then it is highly recommended to hide them in the data model. The idea is to keep the data model as tidy as possible so you may hide some fields that fall into some of the preceding categories based on your use cases.

Unused tables, on the other hand, are the tables with all their fields unused.

While this best practice suggests hiding the unused fields and unused tables, applying it can be a time-consuming process. In particular, finding out the fields that are not referenced anywhere within the model or are not being used in any visuals can be a laborious job if done manually. Luckily, some third-party tools can make our lives easier. For instance, we can use **Power BI Documenter**, which can not only find unused tables and fields but also hide all unused tables and fields in one click.

The following screenshot shows how our **Fields** pane looks before and after hiding unused tables with Power BI Documenter:

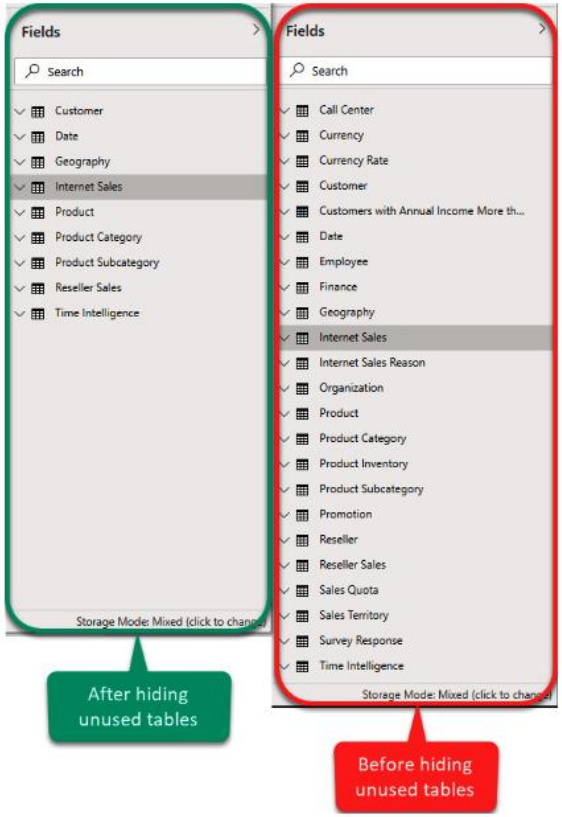


Figure 9.29 – Before and after hiding unused tables using Power BI Documenter

The following screenshot shows what the **`Internet Sales`** table looks like before and after using Power BI Documenter to hide the unused fields:

Figure 9.30 – Before and after hiding unused tables using Power BI Documenter

To learn more about Power BI Documenter, visit www.datavizioner.com.

Hiding key columns

The other best practice that helps us keep our data model tidy is to hide all key columns. The key columns are **`Primary Keys`** and their corresponding **`Foreign Keys`**. While keeping the key columns in the data model is crucial, we do not need them to be visible.

Hiding implicit measures

The other items we can hide in our data model are the implicit measures. We discussed the implicit and explicit measures in *Chapter 8*, *Data Modeling Components*, in the *Measures* section. Best practices suggest creating explicit measures for all implicit measures required by the business and hiding all the implicit measures in the data model. Hiding implicit measures reduces the confusion of which measure to use in the data visualizations for other content creators who are not necessarily familiar with the data model.

Hiding columns used in hierarchies when possible

When we create a hierarchy, it is better to hide the base columns from the report view. Having base columns in a table when the same column appears in a hierarchy is somewhat confusing. So avoid any confusion for the other content creators who are connecting to our data model (dataset).

Creating measure tables

Creating a measure table is a controversial topic in Power BI. Some experts suggest considering using this technique to keep the model even more organized, while others discourage using it. I think this technique is a powerful way to organize the data model; however, there are some side effects to be mindful of before deciding whether to use this technique or not. We will look at some considerations in the next section. For now, let's see what a measure table is. A measure table is not a data table in our data model. We only create them and use them as the home table for our measures. For instance, in our sample report, we can move all the measures from the **`Internet Sales`** table to a separate table. When a table holds the measures only (without any visible columns), Power BI detects the table as a measure table with a specific iconography (). The following steps explain how to create a measure table in Power BI:

1. Click the **Enter Data** button.
2. Leave **`Column1`** as is with an empty value.
3. Name the table **`Internet Sales Metrics.`**
4. Click **Load**:

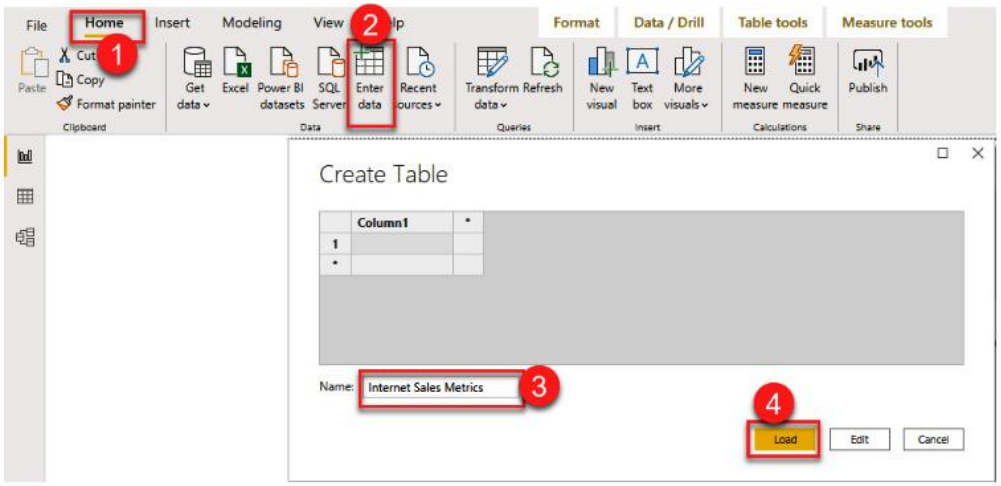


Figure 9.31 – Entering data in Power BI Desktop

5. Right-click **Column1** in the **Internet Sales Metrics** table.
6. Click **Hide**:

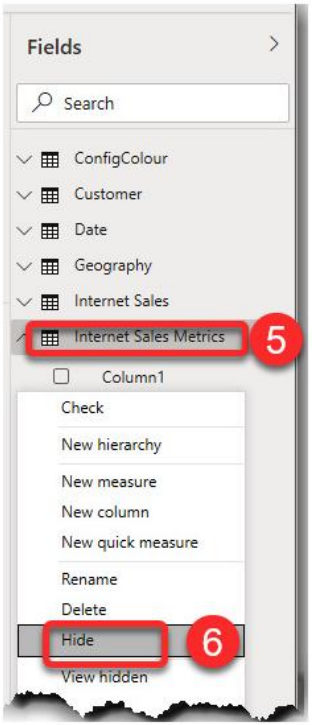


Figure 9.32 – Hiding a column

Now we have to move the measures from the **Internet Sales** table to the **Internet Sales Metrics** table. The following steps show how we can do that:

7. Click the **Model** view.
8. Right-click the **`Internet Sales`** table.
9. Click **Select measures** to select all measures within the **`Internet Sales`** table.

The following screenshot shows the preceding three steps:

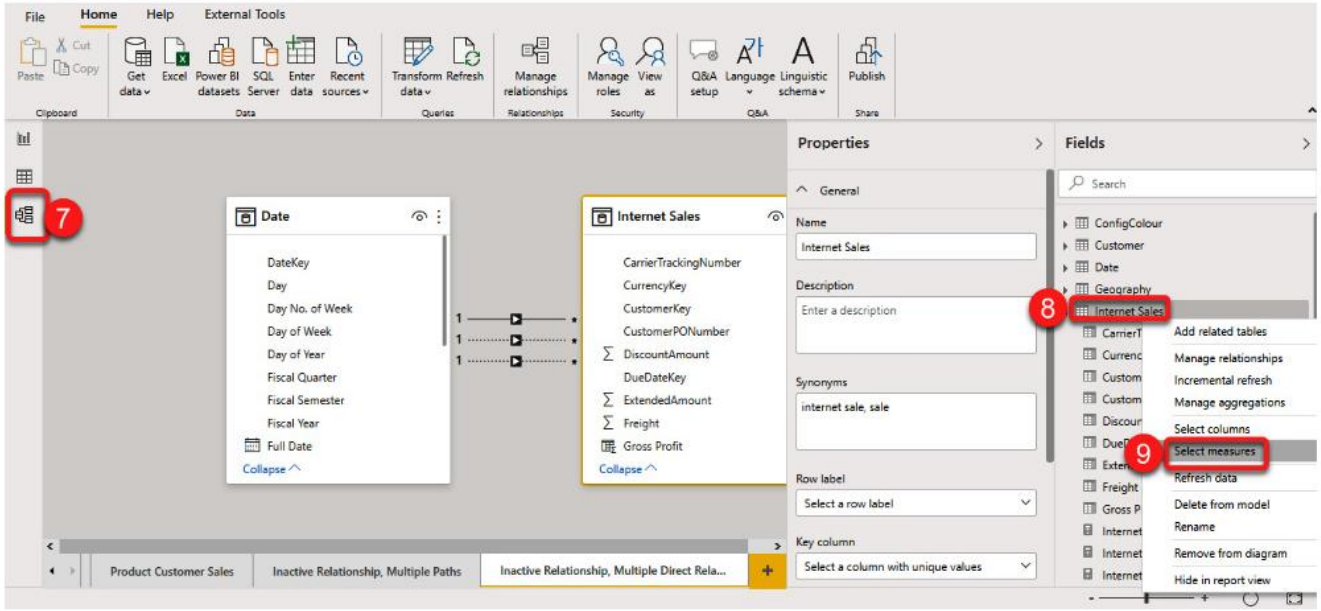


Figure 9.33 – Selecting all measures from a table

10. Drag the selected measures and drop them on the **`Internet Sales Metrics`** table, as shown in the following screenshot:

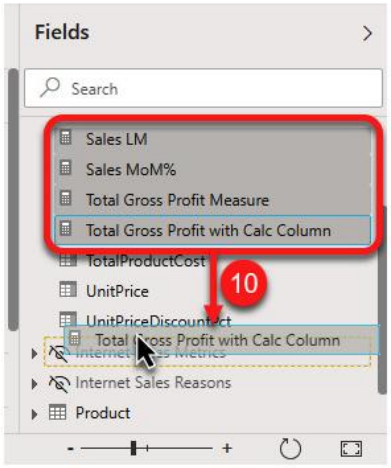


Figure 9.34 – Moving multiple measures from a table to another

11. Click the **Report** view.
12. Hide the **Fields** pane, then unhide it.

The following screenshot shows the preceding two steps:

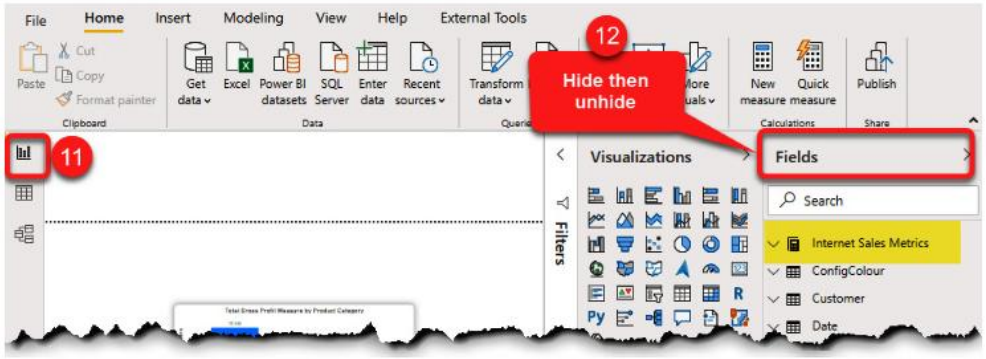


Figure 9.35 – The measure table created in Power BI Desktop

We now have a measure table keeping all the **`Internet Sales`** measures. When Power BI Desktop detects a measure table, it puts the measure table on top of the other tables within the **Fields** pane. We can create separate measure tables for different business domains so that we can have all relevant measures in the same place. This approach helps to make our model tidier and makes it easier for content creators to understand the model.

Considerations

While creating measure tables can help keep our data model tidy, some downfalls are associated with this approach. For instance, it does not make too much sense to have a table with an empty column in the model from a data modeling point of view. So if you do not see any issues with having a table in your model that is only used for holding your measures, this might not sound like a real issue. But there is one more real issue associated with the measure tables, which relates to the **featured tables**. As we discussed the concept of featured tables in *Chapter 8*, *Data Modeling Components*, in the *Featured tables* section, we can configure a table within our Power BI data model as a featured table. After we configure a featured table, the table's columns and measures are available for the Excel users across the organization. Therefore, when we move all the measures from a featured table to a measure table, then the measures will not be available to the Excel users anymore. So the key message here is to think about your use cases then decide whether the measure tables are suitable for your scenarios or not.

Using folders

Another method to keep our data model tidy is to create folders and put all relevant columns and measures into separate folders. Unlike creating measure tables, creating folders does not have any known side effects on the model. Therefore, we can create as many folders as required. We can create new folders or manage existing folders via the **Model** view within the Power BI Desktop. In this section, we discuss some tips and tricks for using folders more efficiently.

Creating a folder in multiple tables in one go

There is a handy way to create folders more efficiently by creating a folder in multiple tables in a single attempt. This method can is handy be many cases, such as creating a folder in the home tables to keep all the measures, or selecting multiple columns and measures from multiple tables and placing them in a folder in multiple tables. The following steps create a folder to keep all measures in multiple home tables:

1. Switch to the **Model** view.
2. Select a table either from the **Model** view or from the **Fields** pane, then press the *Crtl + A* combination from your keyboard to select all tables.
3. Right-click a selected table.
4. Click **Select measures** from the context menu.

   The following screenshot illustrates the preceding steps:

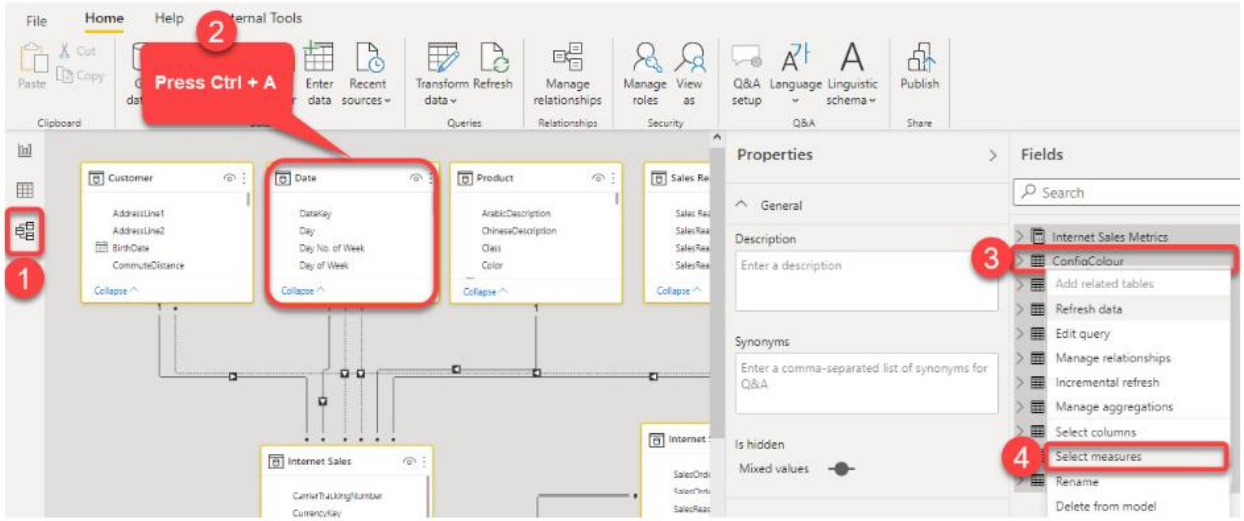


Figure 9.36 – Selecting all measures in the model

5. Type in a name in the **Display folder** (I entered **`Measures`**) and press *Enter* from the keyboard.

The following screenshot shows the **`Measures`** folder created in multiple tables containing all measures used by those tables:

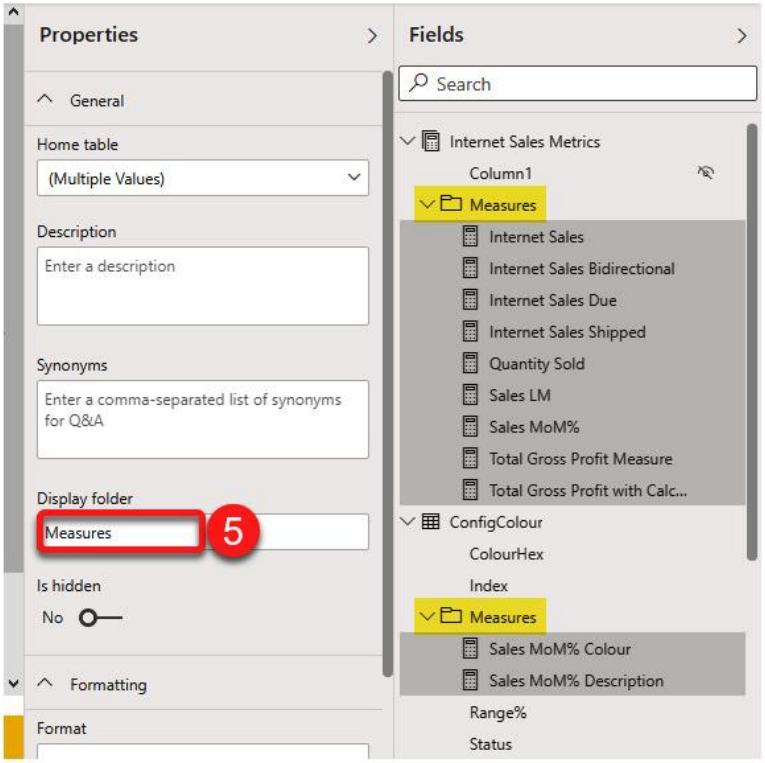


Figure 9.37 – Placing all selected measures in a folder within multiple tables

Placing a measure in multiple folders

In some cases, you might want to place a measure in multiple folders. A use case for this method is to make a measure more accessible for the contributors or support specialists. In our sample file, for instance, we want to have to show **`Sales LM`** and **`Sales MoM%`** measures in both the **`Measures`** folder and in a new **`Time Intelligence`** folder. The following steps show how to do so:

1. Select the **`Sales LM`** and **`Sales MoM%`** measures.
2. In the **Display folder**, add a semicolon after **`Measures`**, then type in the new folder name and press *Enter* from the keyboard.

The following screenshot shows the preceding steps:

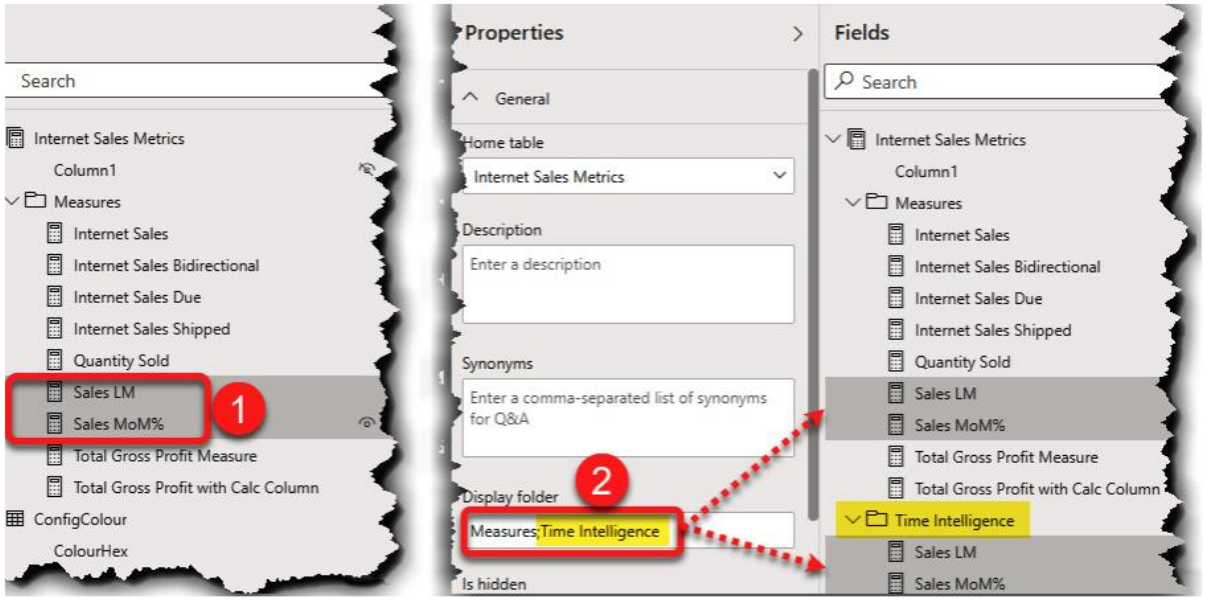


Figure 9.38 – Placing measures in multiple folders

Just to make it clear, the **Display folder** field in the preceding screenshot contains the folder names separated by a semicolon as follows:

```
Measures;Time Intelligence
```

Creating subfolders

In some cases, we want to create subfolders to make the folders even tidier. For instance, in our sample, we want to have a subfolder to keep our base measures. The following steps show how to create a subfolder nested in the root folder:

1. Select the desired measure(s).
2. Use a backslash (\) character to create a subfolder, then press *Enter* on the keyboard.

The following screenshot shows the preceding steps:

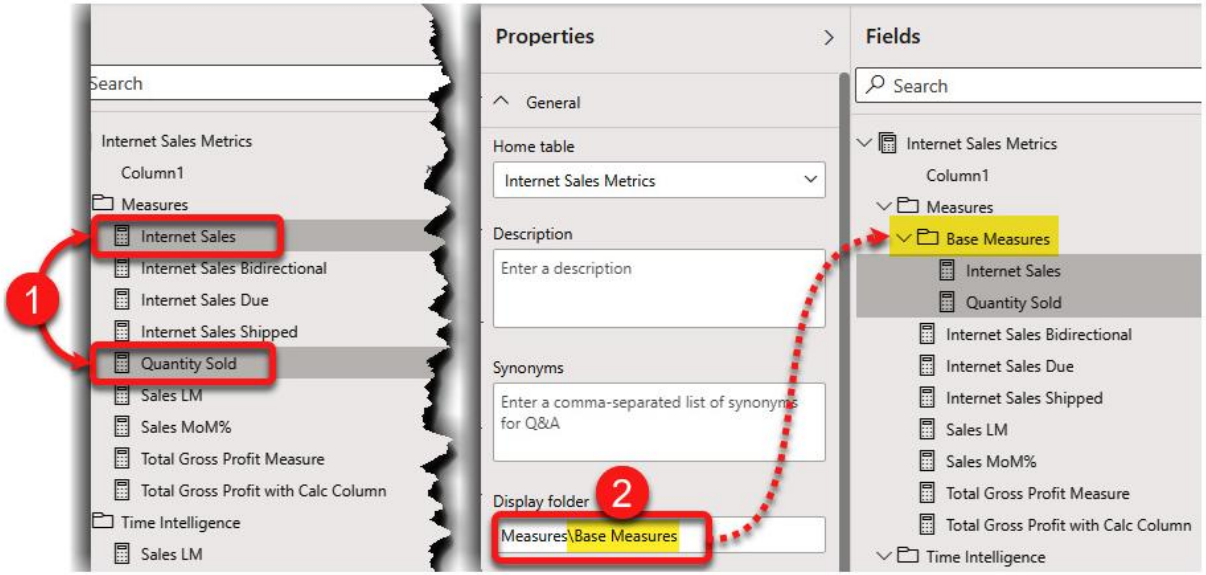


Figure 9.39 – Creating subfolders

Reducing model size by disabling auto date/time

When the data is loaded into the data model, Power BI automatically creates some **`Date`** tables to support calendar hierarchies for all columns in **`DateTime`** datatype. This feature is convenient, especially for beginners who do not know how to create a **`Date`** table or create and manage hierarchies. However, it can consume too much storage, which can potentially lead to severe performance issues. As mentioned earlier, the auto date/time feature forces Power BI Desktop to create **`Date`** tables for every single **`DateTime`** column within the model. The **`Date`** tables have the following columns:

- **`Date`**
- **`Year`**
- **`Quarter`**
- **`Month`**
- **`Day`**

The last four columns are used to create date hierarchies for each **`DateTime`** column. The Date column in the created **`Date`** table starts from January 1 of the minimum year of the related column in our tables. It ends on December 31 of the maximum year of that column. It is a common practice in data warehousing to use 10/01/1900 for unknown dates in the past and 31/12/9999 for unknown dates in the future. So imagine what happens if we have only one column having only one of preceding unknown date values. So it is a best practice to disable this feature in Power BI Desktop. The following steps show how to disable auto date/time:

1. In Power BI Desktop, click the **File** menu.
2. Click **Options and settings**.
3. Click **Options**.

   The following screenshot shows the preceding steps:

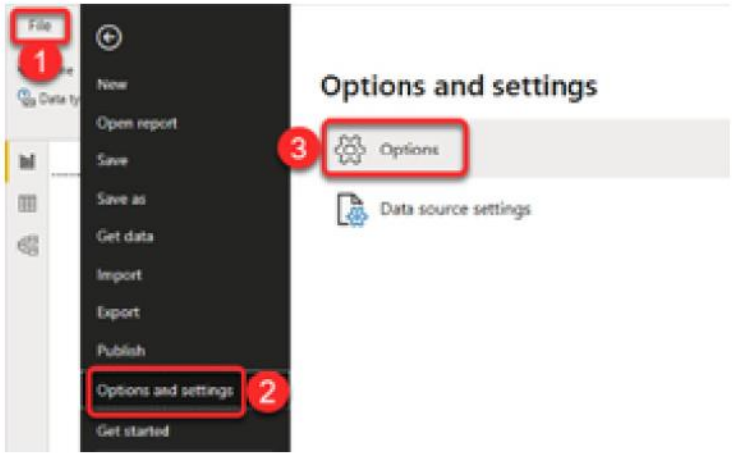


Figure 9.40 – Changing Power BI Desktop Options

4. Click **Data Load** from the **GLOBAL** section.
5. Untick the **Auto date/time for new files**. This will disable this feature globally; therefore, when you start creating a new file, the **Auto date/time** feature is already disabled.

   The following screenshot shows how to disable the **Auto date/time** feature globally:

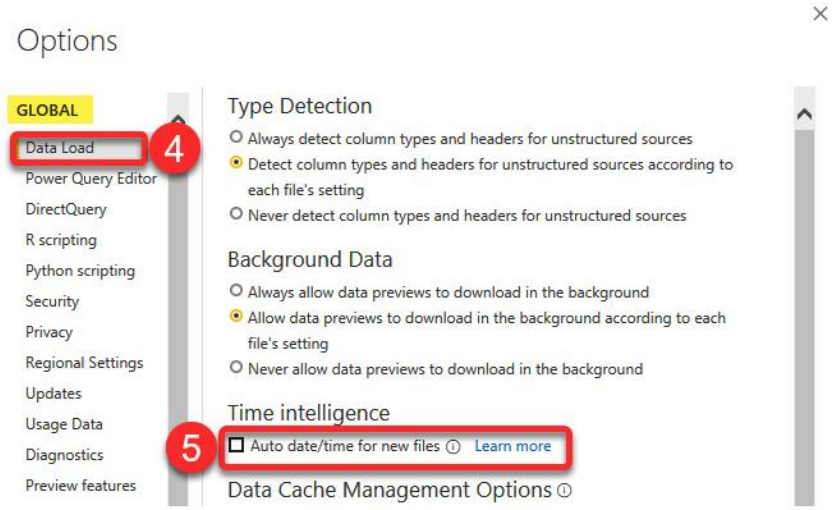


Figure 9.41 – Disabling the Auto date/time feature globally

6. Click **Data Load** from the **CURRENT FILE** section.
7. Untick the **Auto date/time**. This will disable the **Auto date/time** feature only for the current file. So if we did not do the previous step, the **Auto date/time** feature is not disabled for new files.
8. Click **OK**:

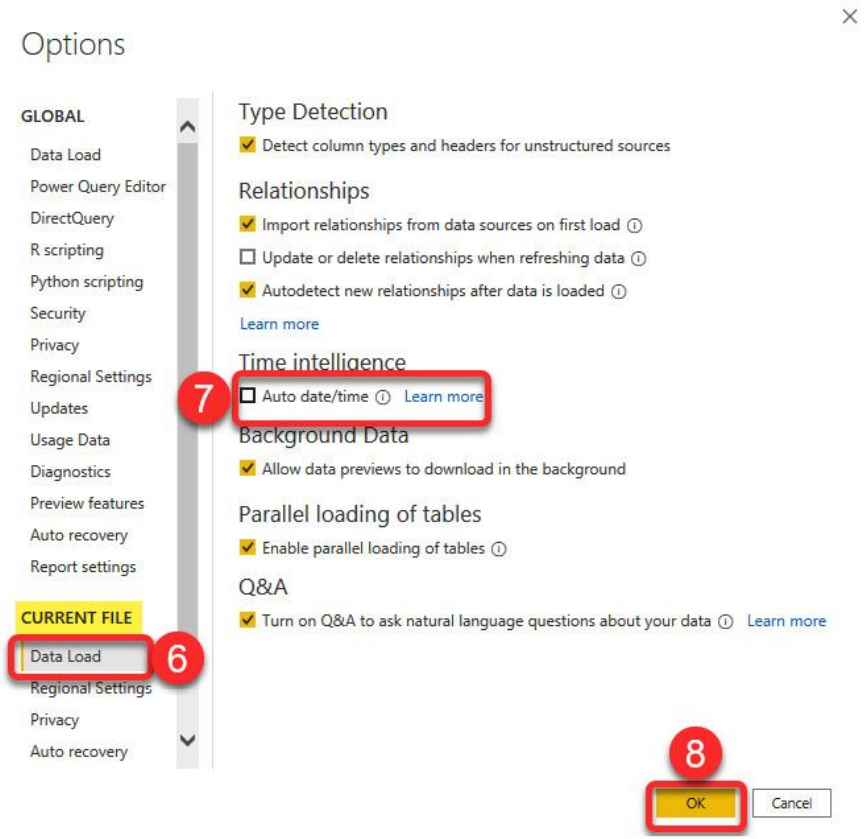


Figure 9.42 – Disabling the Auto date/time feature for the current file

Disabling the **Auto date/time** feature removes all the automatically created **`Date`** tables, which reduces the file size.

Summary

In this chapter, we learned some common best practices for working with Star Schema and data modeling. We learned how to implement many-to-many relationships. We also learned how and when to use bidirectional relationships. Then we looked at disabled relationships and how we can programmatically enable them. We also learned about the config tables and how they can help us with our data visualization. We then discussed why and when we should avoid using calculated columns. Next, we looked at some techniques to organize the data model. Last but not least, we learned how we could reduce the model size by disabling the Auto date/time feature in Power BI Desktop.

In the next chapter, *Advanced Modeling Techniques*, we will discuss some exciting data modeling techniques that can boost our Power BI model performance while creating complex models. See you there.