Chapter 3 of "Expert Data Modeling with Power BI" by Soheli Bakhshi focuses on the process of importing data into Power BI and transforming it into a usable format for data modeling. The chapter provides an overview of the different types of data sources that can be used in Power BI and the steps involved in preparing data for analysis.

The chapter begins by discussing the different types of data sources that can be used in Power BI, including Excel files, CSV files, and databases such as SQL Server and Oracle. The author provides an overview of each data source type and explains how to connect to them in Power BI.

Next, the chapter covers the process of data transformation, which involves cleaning and shaping data to prepare it for analysis. The author explains the different data transformation tools available in Power BI, such as Power Query, and provides examples of how to use them to clean and shape data.

The chapter also covers the concept of data mashups, which involves combining data from multiple sources into a single dataset. The author provides an overview of how to perform data mashups in Power BI and provides examples of how this can be useful in data modeling.

The chapter concludes with a discussion of best practices for importing and transforming data in Power BI. These include starting with a small dataset to test the process, documenting the data transformation steps, and regularly updating the data model to ensure that it reflects the most current data.

Overall, Chapter 3 provides a detailed overview of the process of importing and transforming data in Power BI. The author provides practical examples and tips to help readers prepare their data for analysis and create effective data models that meet their business requirements.

Here are some real-world examples that illustrate the concepts covered in Chapter 3 of "Expert Data Modeling with Power BI" by Soheli Bakhshi:

1. Importing data: A retail company may import data from multiple sources, such as sales data from their point of sale system, inventory data from their warehouse management system, and customer data from their CRM system. By importing this data into Power BI, the company can gain a more comprehensive view of their operations and identify opportunities for improvement.

2. Data transformation: A financial services company may use data transformation tools in Power BI to clean and shape customer data. For example, the company may use Power Query to remove duplicate entries and correct spelling errors in customer names, addresses, and other personal information.

3. Data mashups: A manufacturing company may use data mashups to combine data from multiple sources, such as sales data and production data. By combining this data into a single dataset, the company can gain insights into the relationship between sales and production and identify areas where improvements can be made.

4. Best practices: A healthcare provider may follow best practices for importing and transforming data in Power BI. For example, the provider may start by importing a small dataset to test the process, document the data transformation steps to ensure reproducibility, and regularly update the data model to reflect the most current patient data.

Overall, the concepts covered in Chapter 3 are applicable to a wide range of industries and use cases. By using Power BI to import and transform data, organizations can gain a more comprehensive view of their operations and make more informed business decisions.

Chapter 3: Data Preparation in Power Query Editor

In the previous chapters, we discussed various layers in Power BI and went through some scenarios. By now, it should be pretty clear to you that Power BI is not only a reporting tool. Power BI is indeed a sophisticated all-round **business intelligence** (**BI**) technology, with the flexibility to be used as a self-service BI tool that supports many BI aspects such as **extract, transform, and load** (**ETL**) processes, data modeling, data analysis, and data visualization. Power BI, as a powerful BI tool, is improving every day, which is fantastic. The Power BI development team at Microsoft is constantly bringing new ideas to this technology to make it even more powerful. One of the BI areas that Power BI is great for is taking care of ETL activities in the data preparation layer, with the so-called **Power Query Editor** in Power BI. **Power Query Editor** is the dedicated tool in Power BI to write Power Query expressions, and is also available in Excel and in many other Microsoft data platform products. In this chapter, we look at **Power Query M** in more detail. You will learn about the following topics:

- Introduction to the Power Query M formula language in Power BI
- Introduction to Power Query Editor
- Introduction to Power Query features for data modelers
- Understanding query parameters
- Understanding custom functions

We will use some hands-on, real-world scenarios to see the concepts in action.

Introduction to the Power Query M formula language in Power BI

Power Query is a data preparation technology offering from Microsoft to connect to many different data sources from various technologies to enable businesses to integrate data, transform it, make it available for analysis, and get meaningful insights from it. Not only can Power Query currently connect over a lot of data sources, but it also provides a **custom connectors software development kit** (**SDK**) that third parties can use to create their data connectors. Power Query was initially introduced as an Excel add-in but quickly turned into a vital part of the Microsoft data platform for data preparation and data transformation.

Power Query is currently integrated with many Microsoft products such as **Dataverse** (also known as **Common Data Service** (**CDS**)), **SQL Server Analysis Services Tabular Models** (**SSAS Tabular**), and **Azure Analysis Services** (**AAS**), as well as Power BI and Excel. Therefore, learning about Power Query can help data professionals support data preparation in all technologies mentioned previously. With that, let's have a look at Power Query in Power BI.

Power Query M is a formula language connecting to many different data sources to mix and match the data between those data sources, to create a single dataset. In this section, we introduce Power Query M.

Power Query is CaSe-SeNsItIvE

While Power Query is a *case-sensitive* language, **Data Analysis Expressions** (**DAX**) is not, which may confuse some developers. Remember, Power Query and DAX are different worlds that came together in Power BI to take care of different aspects of working with data. Not only is Power Query case-sensitive in terms of syntax, but it is also case-sensitive when interacting with data. For instance, we get an error message if we type the following function:

```
datetime.localnow()
```

This is because the following is the correct syntax:

```
DateTime.LocalNow()
```

Ignoring Power Query's case sensitivity in data interactions can turn into an issue that is time-consuming to identify. A real-world example is when we get **globally unique identifier** (**GUID**) values from a data source containing lowercase characters. Then, you get some other GUID values from another data source with uppercase characters. When we match the values in Power Query by merging the two queries, we may not get any matching values. In reality, if we turn the GUID containing lowercase characters to uppercase, we get actual matching values. While we do not get any error messages when comparing the two GUIDs, the result is incorrect if we do not have the two GUIDs with matching character cases.

For example, the following two GUID values are not equal from a Power Query point of view, while they are equal from a DAX point of view:

`D5E99E0E-0737-45B2-B62A-4170B3FEFC0E`

`d5e99e0e-0737-45b2-b62a-4170b3fefc0e`

Queries

In Power Query, a query contains **expressions**, **variables**, and **values** encapsulated by **`let`** and **`in`** statements. A **`let`** and **`in`** statement block is structured as follows:

```
let
   Variablename = expression1,
   #"Variable name" = expression2
in
   #"Variable name"
```

As the preceding structure shows, we can have spaces in the variable names. However, we need to encapsulate the variable name using a number sign followed by quotation marks—for example, **`#"Variable Name"`**. By defining a variable in a query, we are creating a **query formula step** in Power Query. Query formula steps can reference any previous steps. Lastly, the output of the query is any variable that comes straight after the **`in`** statement. Each step must end with a comma, except the last step before the **`in`** statement.

Expressions

In Power Query, an expression is a formula that results in values. For instance, the following screenshot shows some expressions and their resulting values:

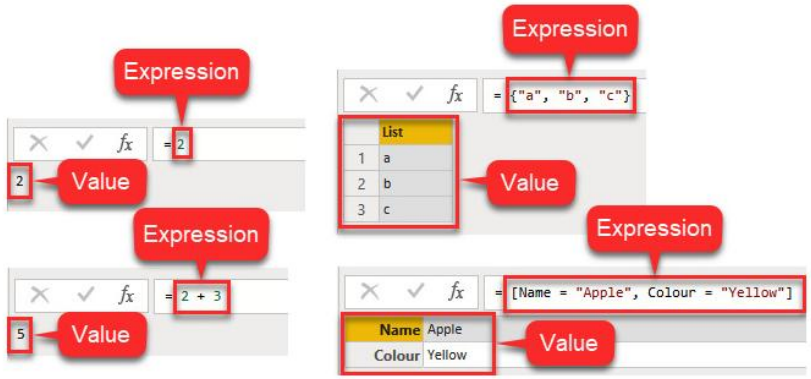

Figure 3.1 – Expressions and their values

Values

As mentioned earlier, values are the results of expressions. For instance, in the top left of *Figure 3.1*, the expression is **2**, resulting in **2** as a constant value.

In Power Query, values fall into two general categories: **primitive values** and **structured values**.

Primitive values

A primitive value is a constant value such as a number, a text, a null, and so on. For instance, **`123`** is a primitive **`number`** value, while **`"123"`** (including quotation marks) is a primitive **`text`** value.

Structured values

Structured values contain either primitive values or other structured values. There are four kinds of structured values: **list**, **record**, **table**, and **function** values:

- **List value**: A list is a sequence of values shown in only one column. We can define a list value using curly brackets **`{}`**. For instance, we can create a list of small English letters as **`{"a".."z"}`**, as shown in the following screenshot:

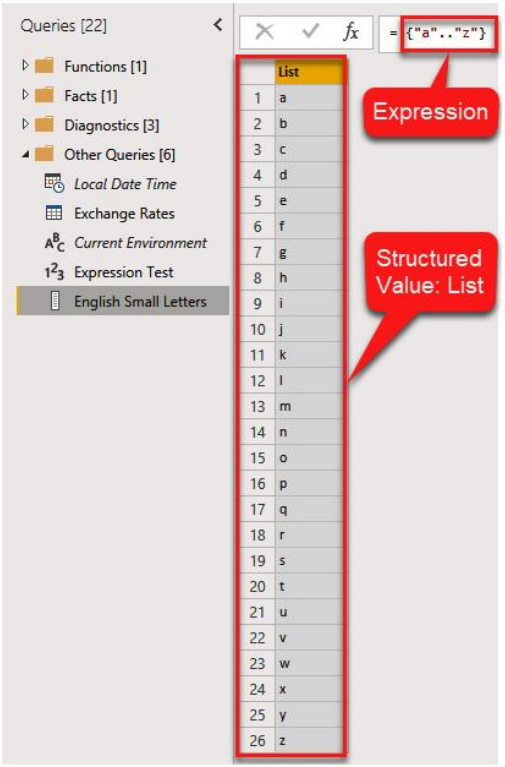

Figure 3.2 – Defining a list of small English letters

- **Record value**: A record is a set of fields that make a row of data. To create a record, we put the field name, an equals sign, and the field's value in brackets **[ ]**. We separate different fields and their values by using a comma, as follows:

```
[
    First Name = "Soheil"
    , Last Name = "Bakhshi"
    , Occupation = "Consultant"
    ]
```

The following screenshot shows the expression and the values:

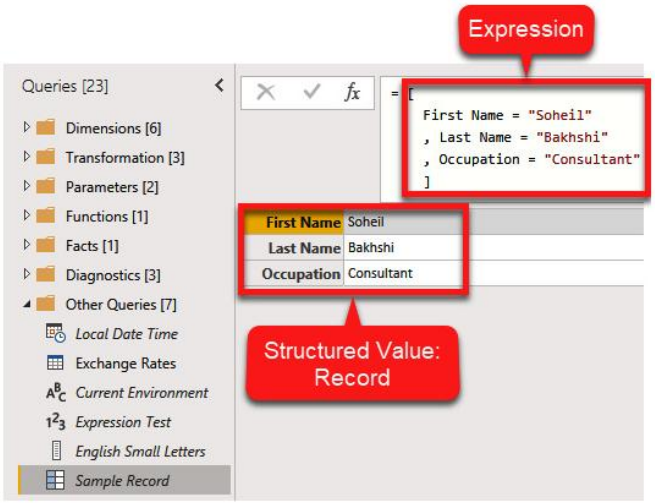

Figure 3.3 – Defining a record in Power Query

NOTE

When defining a record, we do not need to put the field names in quotation marks.

*As illustrated in Figure 3.3, in **Power Query Editor**, records are shown vertically.*

As stated before, a structured value can contain other structures' values. The following expression produces a record value containing a list value that holds primitive values:

```
[
    Name = {"Soheil", "John"}
    ]
```

The following screenshot shows the result (a record value containing list values):

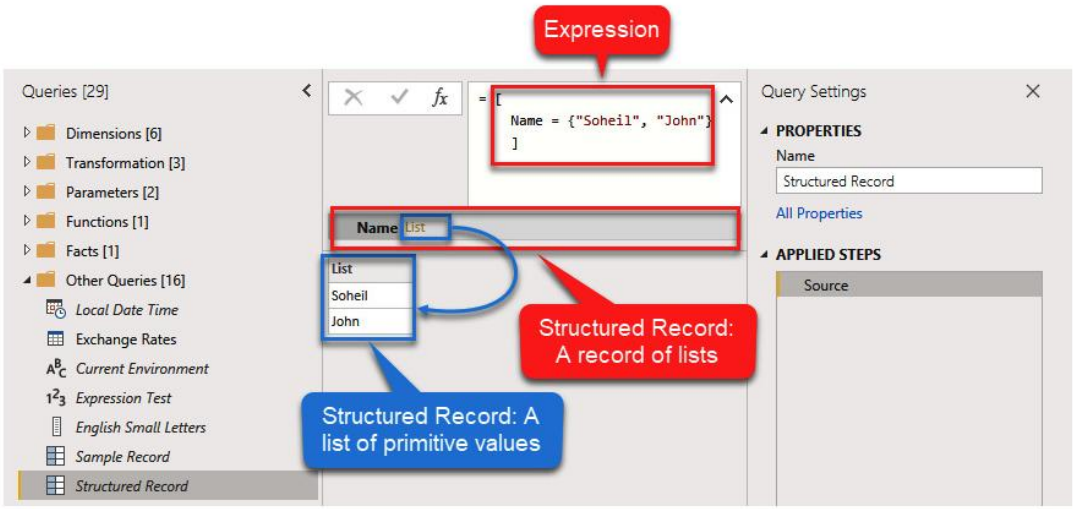

Figure 3.4 – A structured value containing other structures' values

- **Table value**: A table is a set of values organized into columns and rows. Each column must have a name. There are several ways to create a table using several standard Power Query functions. Nevertheless, we can construct a table from lists or records. *Figure 3.5* shows two ways to construct a table, using the **`#table`** keyword shown in the next code snippet.

1. Here is the first way to construct a table:

```
#table( {"ID", "Fruit Name"}                , {{1, "Apple"}, {2, "Orange"}, {3, "Banana"}})
```

2. Here is the second way to construct a table:

```
#table( type table [ID = number, Fruit Name = text], {{1, "Apple"}, {2, "Orange"}, {3, "Banana"}} )
```

The following screenshot shows the results:

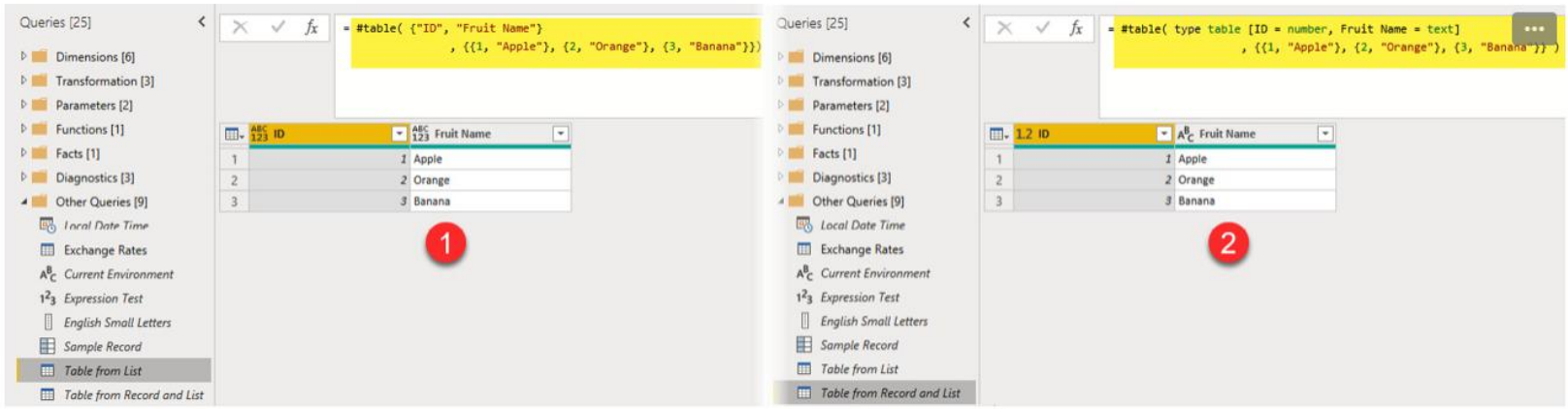

Figure 3.5 – Constructing a table in Power Query

As you see in the preceding screenshot, we defined the column data types in the second construct, while in the first one, the column types are **any**.

The following expression produces a table value holding two lists. Each list contains primitive values:

```
#table( type table
            [Name = list]
                , {{{"Soheil", "John"}}}
                            )
```

We can expand a structured column to get its primitive values, as illustrated in the following screenshot:

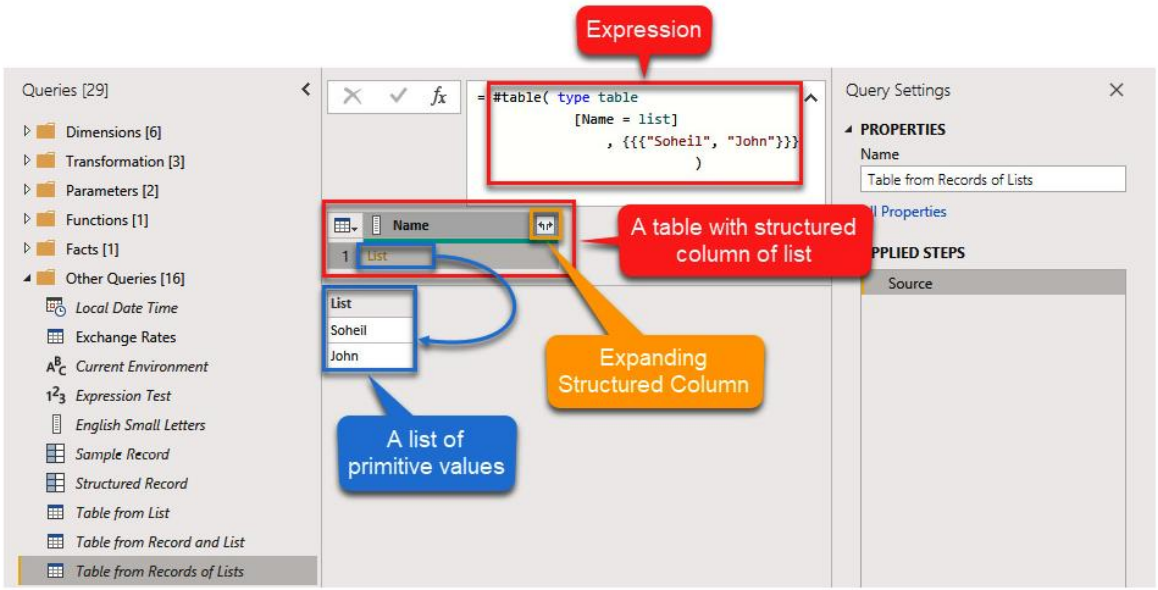

Figure 3.6 – Table with a structured column

The following screenshot shows the result after expanding the structured column to new rows:

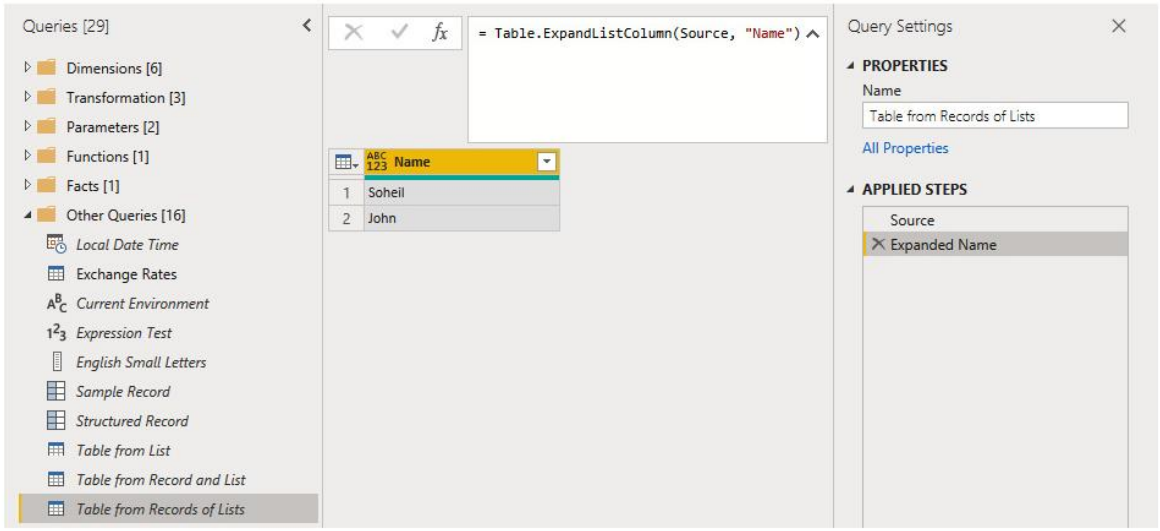

Figure 3.7 – Table with an expanded structured column

- **Function value**: A function is a value that accepts input parameters and produces a result. To create a function, we put the list of parameters (if any) in parentheses, followed by the output data type. We use the goes-to symbol (**=>**), followed by a definition of the function.

  For instance, the following function calculates the end-of-month date for the current date:

  ```
  ()as date => Date.EndOfMonth(Date.From(DateTime.LocalNow()))
  ```

  The preceding function does not have any input parameters but produces an output.

  The following screenshot shows a function invocation without parameters that returns the end-of-month date for the current date (today's date):

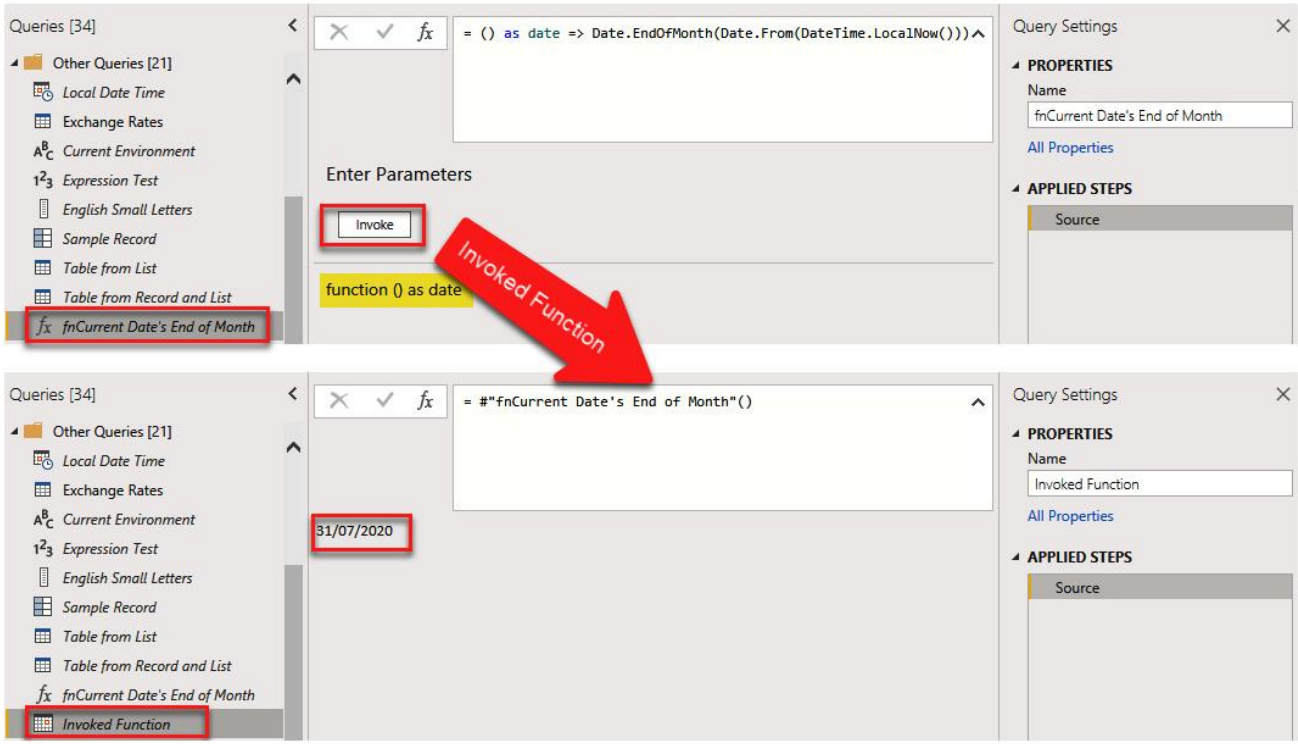

Figure 3.8 – Invoking a custom function

Types

In Power Query, values have types. There are two general categories for types: **primitive types** and **custom types**.

Primitive types

A value can have a primitive type, as follows:

- `binary`
- `date`
- `datetime`
- `datetimezone`
- `duration`
- `list`
- `logical`
- `null`
- `number`
- `record`
- `text`
- `time`
- `type`
- `function`
- `table`
- `any`
- `none`

Out of the values in the preceding list, the **any** type is an interesting one. All other Power Query types are compatible with the **any** type. However, we cannot say a value is of type **any**.

Custom types

Custom types are types we can create. For instance, the following expression defines a custom type of a list of numbers:

```
type { number }
```

Introduction to Power Query Editor

In Power BI Desktop, Power Query is available within **Power Query Editor**. There are several ways to access **Power Query Editor**, outlined as follows:

- Click the **Transform data** button from the **Home** tab, as illustrated in the following screenshot:

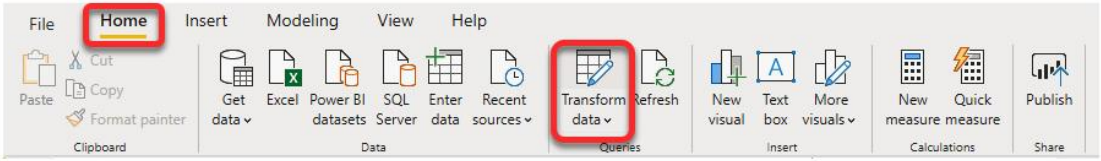

Figure 3.9 – Opening Power Query Editor from the ribbon in Power BI

- We can navigate directly to a specific table query in **Power Query Editor** by right-clicking the desired table from the **Fields** pane then clicking **Edit query**, as shown in the following screenshot:

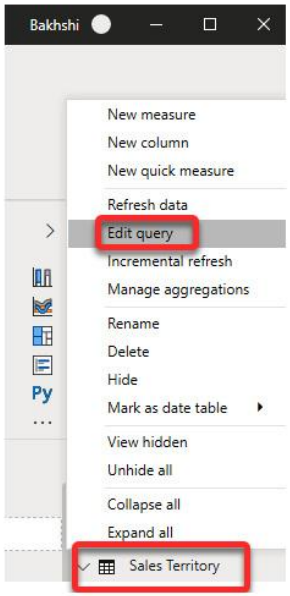

Figure 3.10 – Navigating directly to a specific underlying query in Power Query Editor

Power Query Editor has the following sections:

3. The Ribbon bar
4. The **Queries** pane
5. The **Query Settings** pane
6. The **Data View** pane
7. The Status bar

The following screenshot shows the preceding sections:

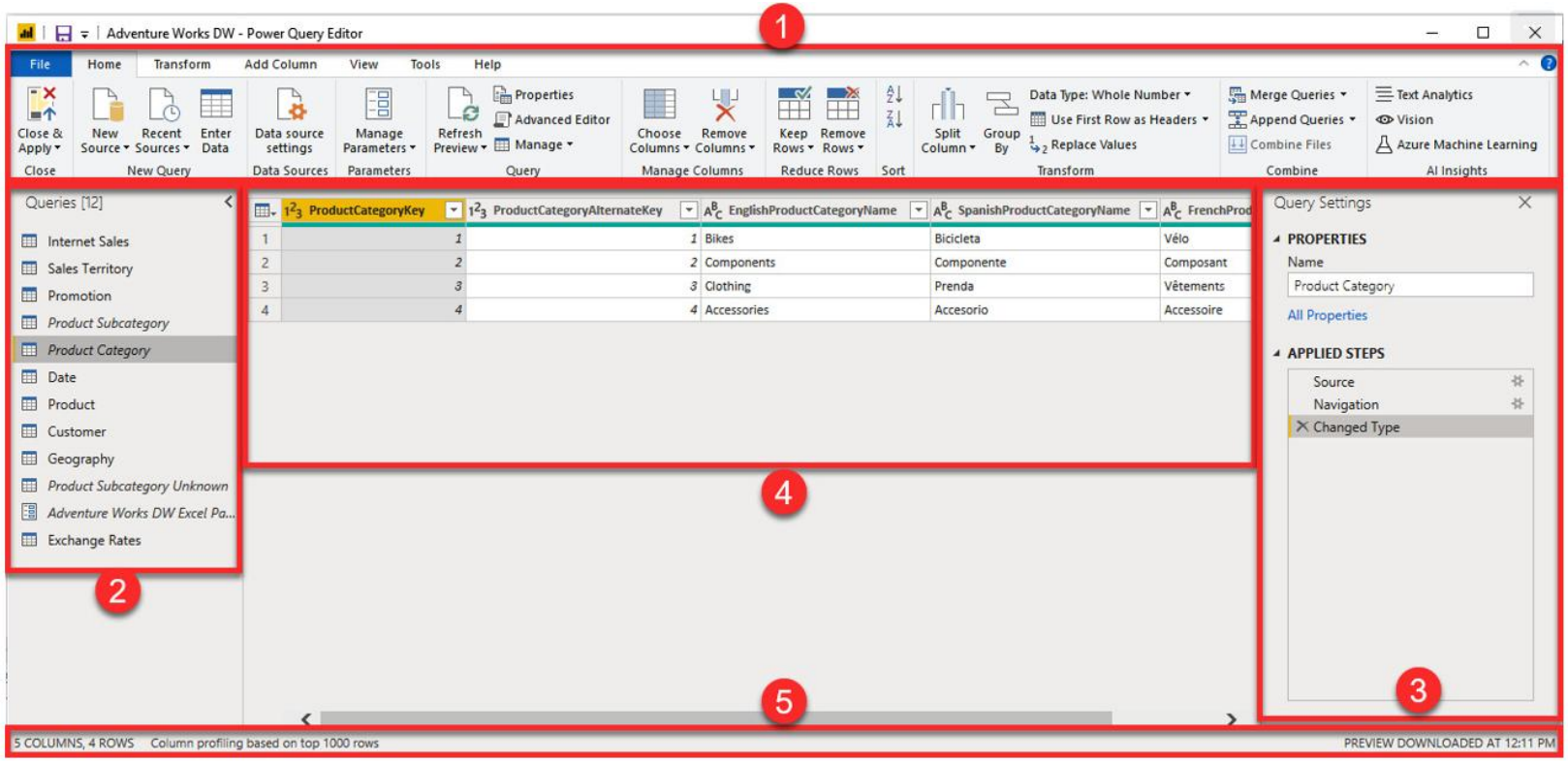

Figure 3.11 – Sections of Power Query Editor

In the next few sections, we will go through some features and tools available in **Power Query Editor** related to data modeling.

Queries pane

This section shows all active and inactive queries, including tables, custom functions, query parameters, constant values, and groups. In the next few sections, we discuss those.

Tables

This includes tables loaded from the data sources, tables created within Power BI using **Enter Data**, and tables that reference other queries. The icon for tables is .

Custom Functions

These are functions we create within **Power Query Editor**. We can invoke and reuse custom functions in other queries. The icon for custom functions is .

Query parameters

We can parameterize various parts of our queries using query parameters that must be hardcoded otherwise. We can find the query parameters in the **Queries** pane under this icon: .

Constant values

In some cases, we may have a query with a constant result that can be a string, datetime, date, time zone, and so on. We can quickly recognize queries with a constant value output from their icon, depending on their resulting data type. For instance, if the query output is **`DateTime`**, then the query icon would be , or if the output is a string, then the iconography would be .

Groups

We can organize the **Queries** pane by grouping queries as follows:

8. Select relevant tables to group by pressing and keeping down the *Ctrl* key from your keyboard and clicking the desired tables from the **Queries** pane.
9. Right-click on the mouse and select **Move To Group**.
10. Click **New Group…** from the context menu.

The following screenshot shows the steps outlined previously to group selected tables:

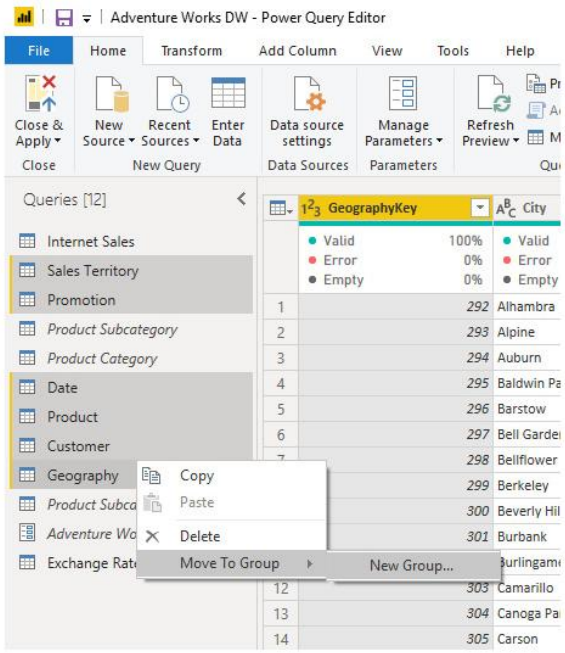

Figure 3.12 – Organizing queries in Power Query Editor

Organizing queries is recommended, especially in larger models that may have many queries referencing other queries.

The following screenshot illustrates what organized queries may look like:

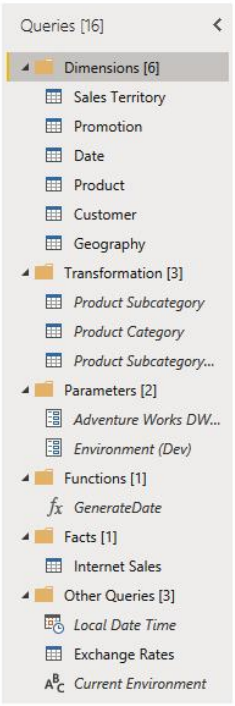

Figure 3.13 – Organized queries in Power Query Editor

Query Settings pane

This pane, located on the right side of the **Power Query Editor** window, contains all query properties and all transformation steps applied to the selected query (from the **Queries** pane). The **Query Settings** pane will not show up if the selected query is a query parameter.

The **Query Settings** pane has two parts: **PROPERTIES** and **APPLIED STEPS**, as illustrated in the following screenshot:

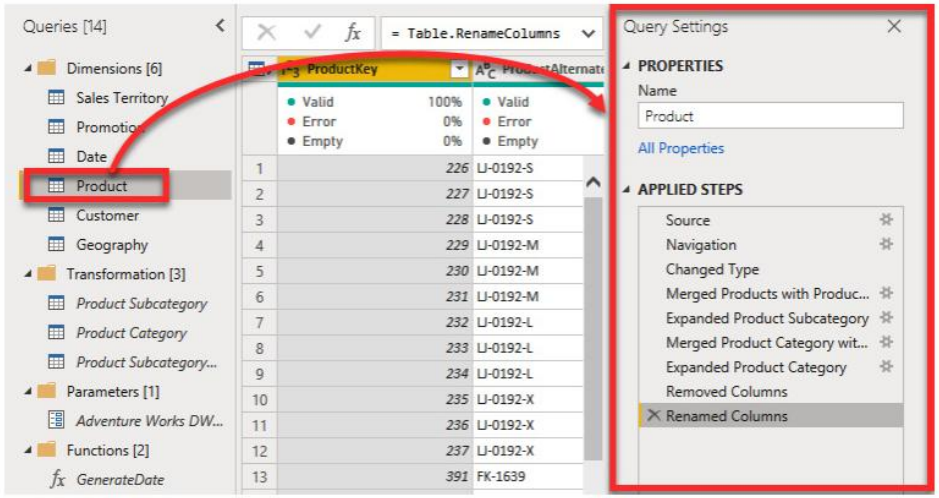

Figure 3.14 – Query Settings pane in Power Query Editor

Query Properties

We can rename a selected query by typing in a new name in the **Name** textbox. We can also set some other properties by clicking **All Properties**, as shown in the following screenshot:

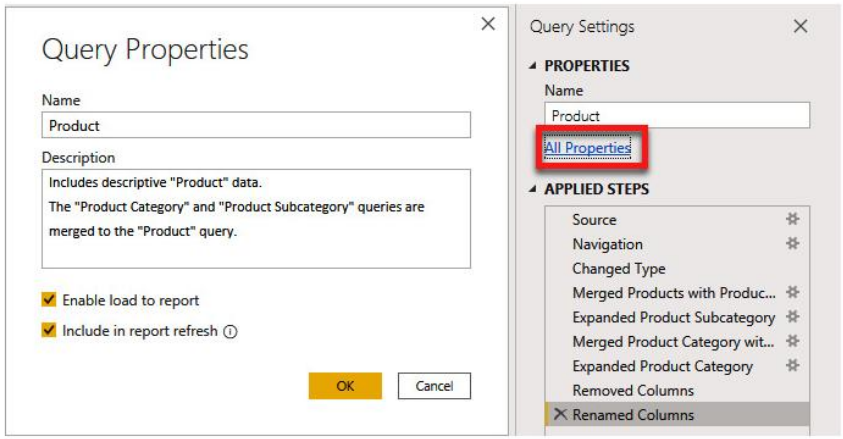

Figure 3.15 – Query Properties from the Query Settings pane

Here is what we can see in the preceding screenshot:

- **Name**: This is, again, the query name.
- **Description**: We can type in some description for the query. This is useful as it can help us with documentation.
- **Enable load to report**: When enabled, data will be loaded into the data model from the source system(s). As you see in *Figure 3.15*, we merged the **`Product`** query with two other queries. Each query may come from a different data source. When this option is disabled, data will not be loaded into the data model. However, if other queries reference this query, data will flow through all the transformation steps applied to this query.
- **Include in report refresh**: In some cases, we need data to be loaded into the model just once, so we do not need to include the query in the report refresh. When this option is enabled, the query gets refreshed whenever the data model is refreshed. We can either refresh the data model from Power BI Desktop when we click the **Refresh** button or we can publish the report to the Power BI service and refresh data from the service. Either way, if this option is disabled for a query, that query is no longer included in future data refreshes.

  IMPORTANT NOTE

  *The **Include in report refresh** option is dependent upon the **Enable load to report** option. Therefore, if **Enable load to report** is disabled, then **Include in report refresh** will also be disabled.*

  *It is a common technique in more complex scenarios to disable **Enable load to report** for some queries that are created as transformation queries. The other queries then reference these queries.*

As the following screenshot shows, we can also access the query properties as well as the **Enable load** and **Include in report refresh** settings from the **Queries** pane by right-clicking on a query:

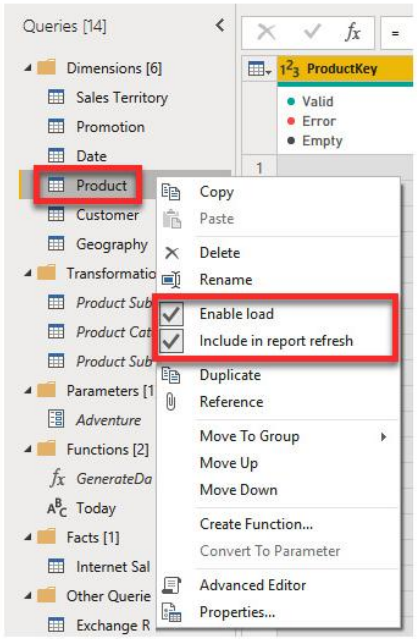

Figure 3.16 – Accessing Enable load and Include in report refresh settings from the Queries pane

Data View pane

The **Data View** pane is placed in the center of **Power Query Editor**. When selecting a query from the **Queries** pane, depending on the type of the selected query we see one of the following:

- A table with its underlying data when the selected query is a table, as shown in the following screenshot:

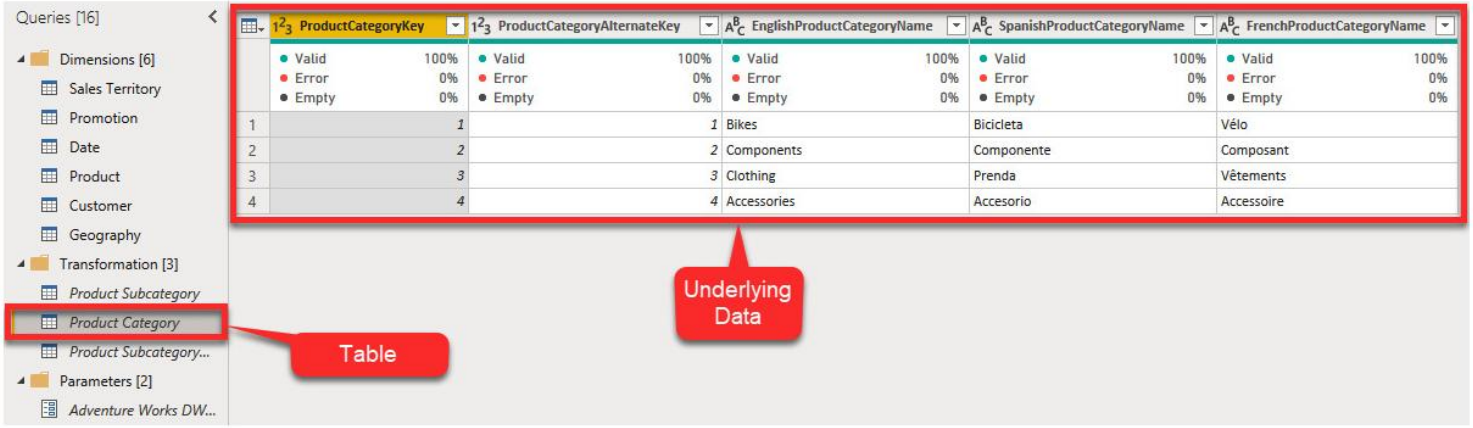

Figure 3.17 – The Data View pane when the selected query from the Queries pane is a table

- **Enter Parameters**, to invoke a function when the selected query is a custom function, as shown in the following screenshot:

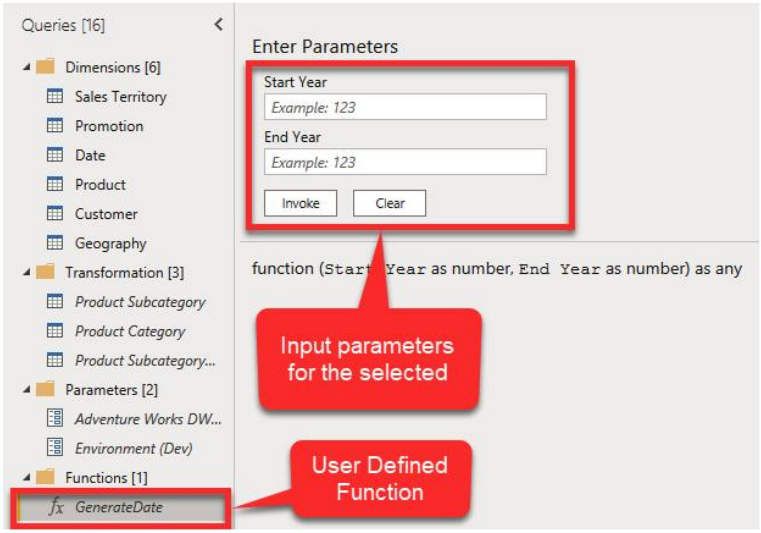

Figure 3.18 – Data View pane when selecting a custom function from the Queries pane

- The results of the selected query. The following screenshot shows the **Data view** pane when the selected query retrieves the local date and time:

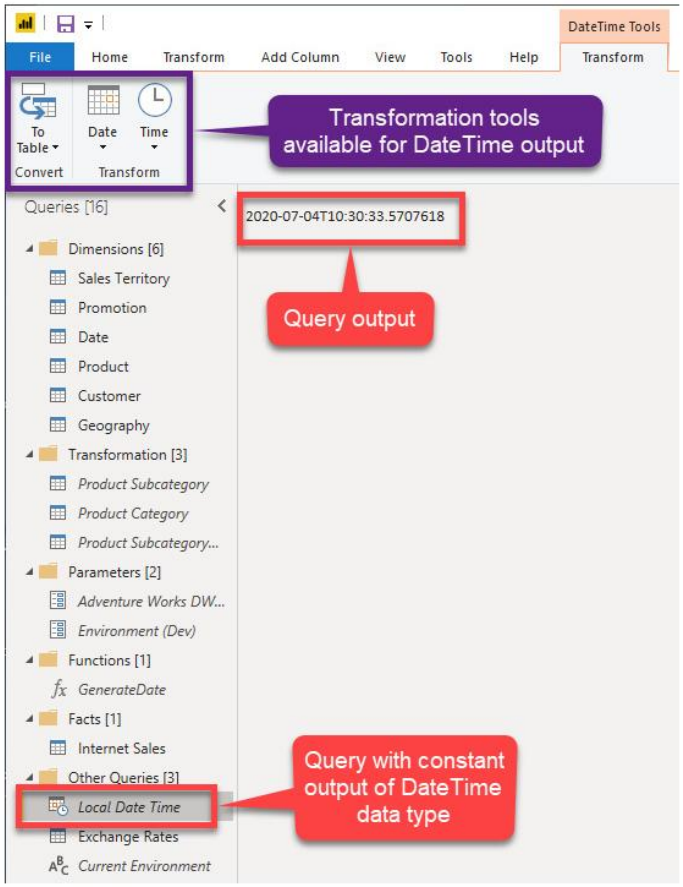

Figure 3.19 – Data View pane when the query output is a constant value

NOTE

Different transformation tools are available in the ribbon bar, depending on the data type of the results of the selected query. Figure 3.20 shows the results of a query that references the ***`Environment`*** *parameter, which is a query parameter. So, the result of the* ***`Current Environment`*** *query varies depending on the values selected in the* ***`Environment`*** *query parameter.*

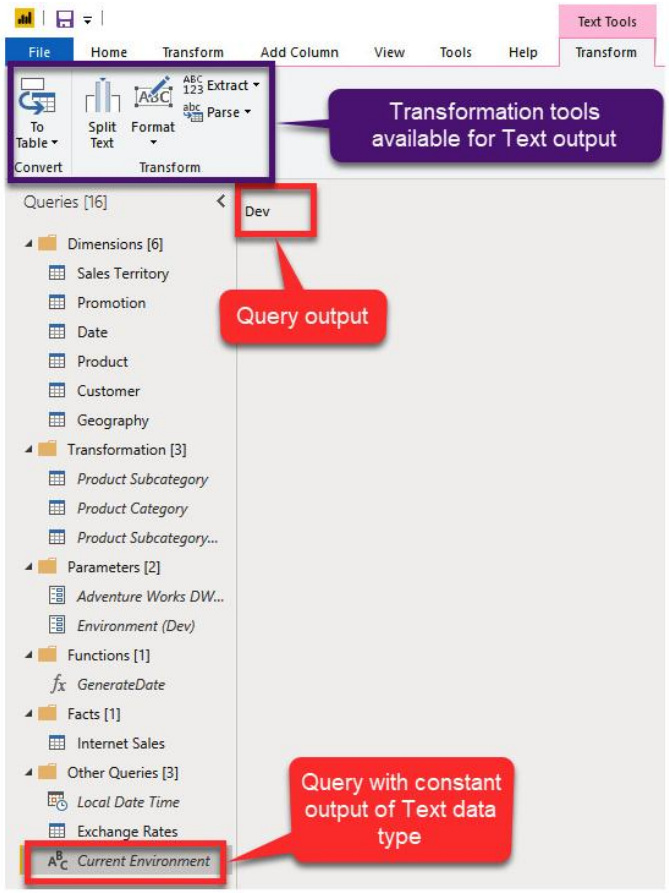

Figure 3.20 – Transformation tools available for a query resulting in a Text value

As we can see, the transformation tools available in *Figure 3.19* and *Figure 3.20* are different.

Status bar

At the bottom of **Power Query Editor**, we have a status bar that includes some information about the selected query from the **Queries** pane, as shown in the following screenshot:

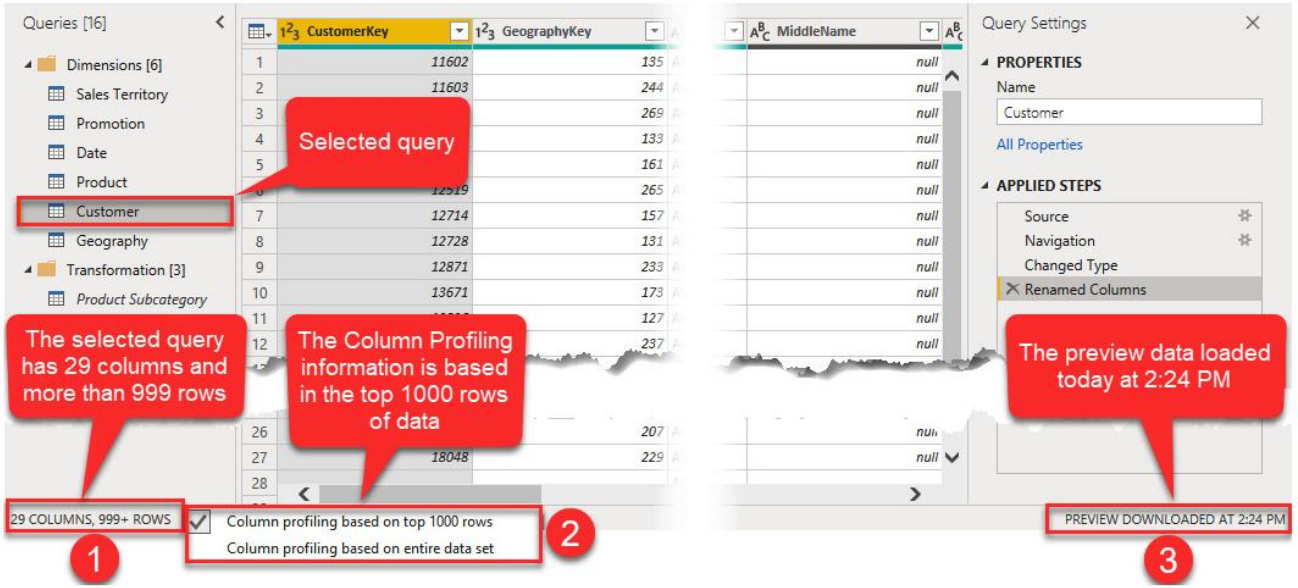

Figure 3.21 – Status bar in Power Query Editor

In the preceding screenshot, we can see the following features:

1. Number of columns: We can quickly get a sense of how wide the table is.
2. Number of rows contributing to **Column profiling**: This enables us to indicate whether the profiling information provided is trustworthy. In some cases, the **Column profiling** setting shows incorrect information when calculated based on **`1000`** rows (which is the default setting).
3. When the data preview refreshed.

Advanced Editor

To create a new query or modify an existing query, we might use the **Advanced Editor**. The **Advanced Editor** is accessible from various places in **Power Query Editor**, as shown in *Figure 3.22*.

To use the **Advanced Editor**, proceed as follows:

1. Select a query from the **Queries** pane.
2. Either click on **Advanced Editor** from the **Home** tab on the ribbon or right-click the query and select **Advanced Editor** from the context menu. Both options are illustrated in the following screenshot:

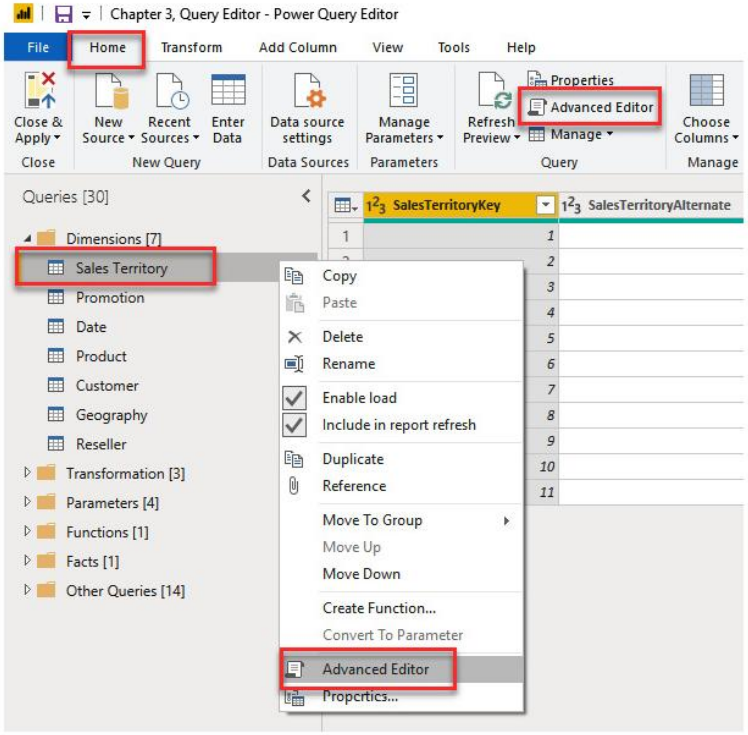

Figure 3.22 – Opening the Advanced Editor

Introduction to Power Query features for data modelers

This section looks at some features currently available within **Power Query Editor** that help data modelers identify and fix errors quicker. Data modelers can get a sense of data quality, statistics, and data distribution within a column (not the overall dataset). For instance, a data modeler can quickly see a column's cardinality, how many empty values a column has, and so on and so forth.

NOTE

*As previously mentioned, the information provided by the **Column quality**, **Column distribution**, and **Column profile** features is calculated based on the top **1000** rows of data (by default), which in some cases leads to false information. It is good practice to set **Column profile** to get calculated based on the entire dataset for smaller amounts of data. However, this approach may take a while to load the column profiling information for larger amounts of data, so be careful while changing this setting if you are dealing with large tables.*

To change the preceding setting from the status bar, proceed as follows:

1. Click the **Column profiling based on top 1000 rows** drop-down.
2. Select **Column profiling based on entire data set**.

The following screenshot illustrates how to do this:

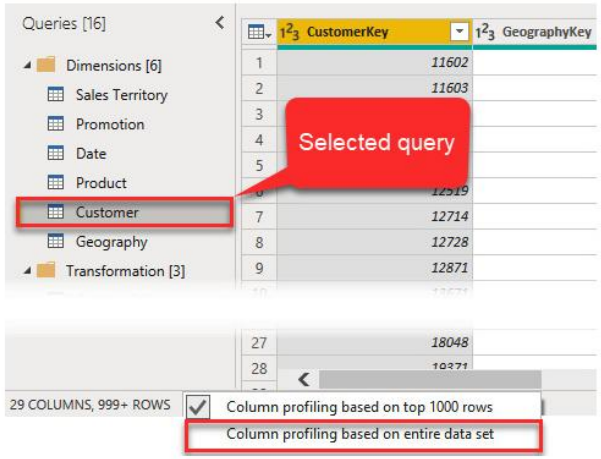

Figure 3.23 – Setting column profiling to be calculated based on the entire dataset

Column quality

In **Power Query Editor**, we see a green bar under each column title that briefly shows the column's data quality. This green bar is called the **Data Quality Bar**. When we hover over it, a flyout menu shows up more data quality-related information. The following screenshot shows the data quality of the **`Size`** column from the **`Product`** table:

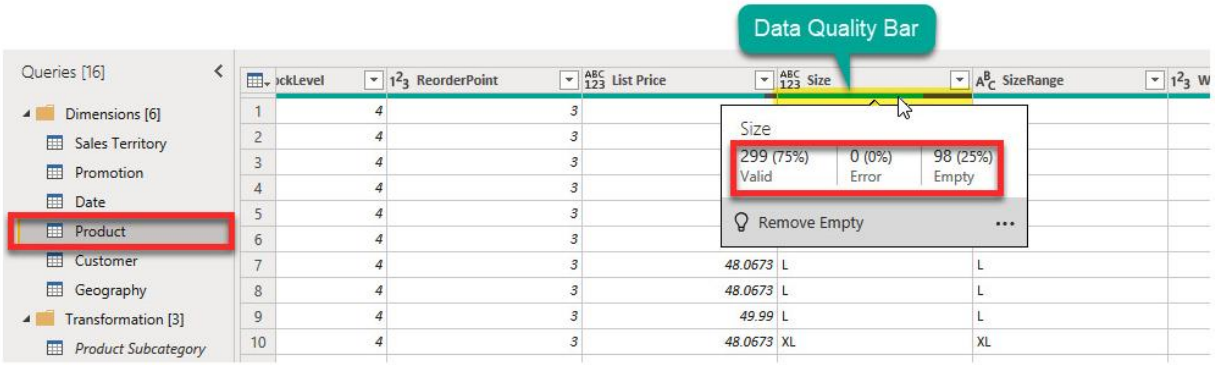

Figure 3.24 – Data Quality Bar in Power Query Editor

While this is an excellent feature, it is still hard to efficiently get a sense of the data quality. There is another feature available in **Power Query Editor** called **Column quality**. The following steps show how to enable the **Column quality** feature:

1. In **Power Query Editor**, navigate to the **View** tab.
2. Tick **Column quality**.
3. More details will be shown in a flyout menu by hovering over the **Column quality** box for each column.

As illustrated in the following screenshot, with the **Column quality** feature, we can quickly validate columns' values and identify errors (if any), valid values, and empty values by percentage. This is very useful for identifying errors. We can also use this feature to identify columns with many empty values so that we can potentially remove them later:

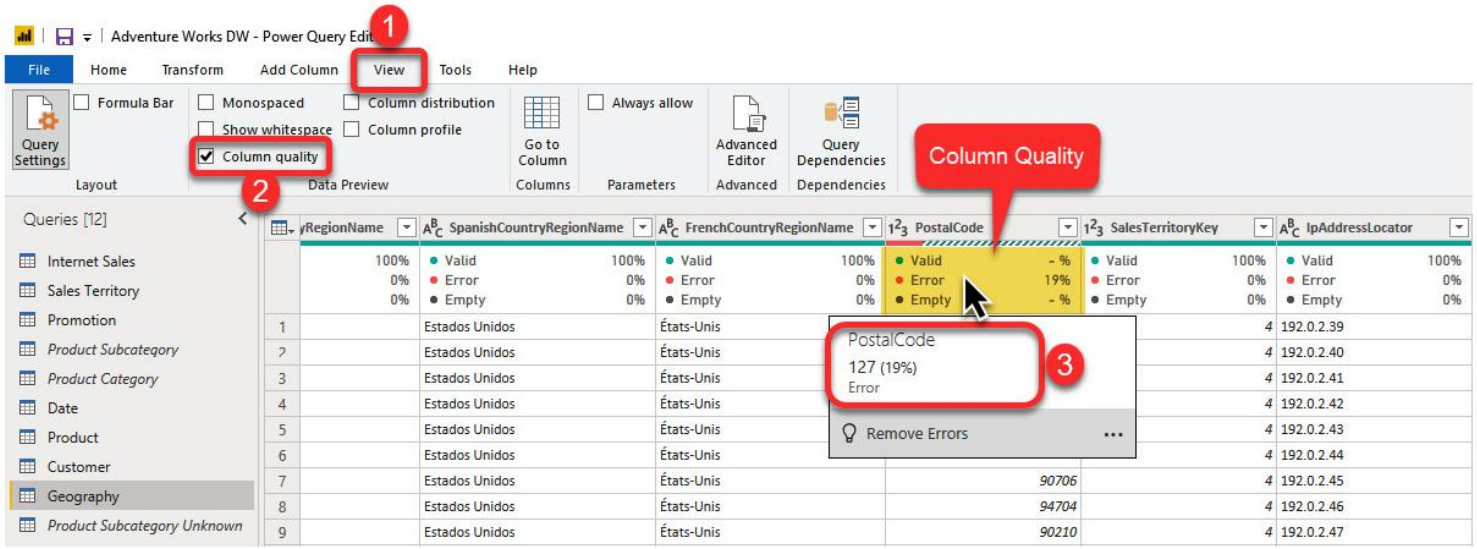

Figure 3.25 – Enabling Column quality in Power Query Editor

We can also take some actions from the flyout menu by clicking the ellipsis button on the bottom right of the flyout menu, as shown in the following screenshot:

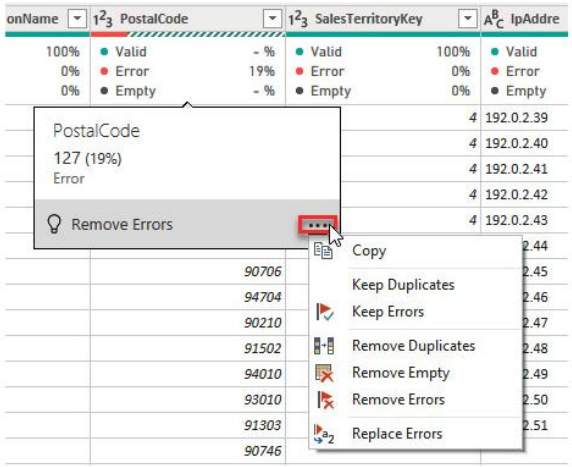

Figure 3.26 – Available options from the Column quality box flyout menu

The preceding screenshot shows that we can copy the data quality, which can be helpful for documentation. From the flyout menu, we can also take action on errors, such as removing any errors. It is a good practice to review and fix errors wherever possible, but we only tend to remove errors where necessary.

Let's look at other use cases for the **Column quality** feature to see how it can help us in real-world scenarios. For this scenario, we will use the **`Chapter 3, Query Editor.pbix`** file.

We want to remove all columns from the **`Customer`** table with less than 10% valid data. The following steps show how to do this:

1. Open the **`Chapter 3, Query Editor.pbix`** file.
2. Open **Power Query Editor**.
3. We can quickly look at the Quality box of all columns, and we can see that the following columns can be removed:

1. **Title**
2. **Suffix**
3. **AddressLine2**

The following screenshot shows that the preceding columns contain a lot of empty values:

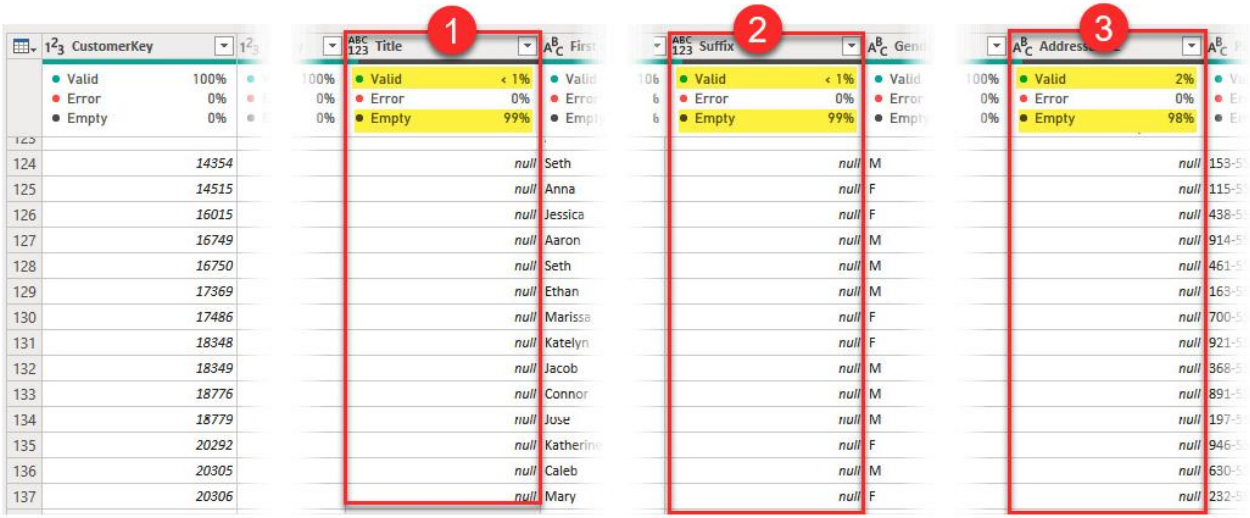

Figure 3.27 – Using Column quality to identify columns with less than 10% valid data

We can remove those columns by doing the following:

1. Click the **Home** tab.
2. Click the **Choose Columns** button.
3. Uncheck the preceding columns.
4. Click **OK**.

The following screenshot shows the preceding steps:

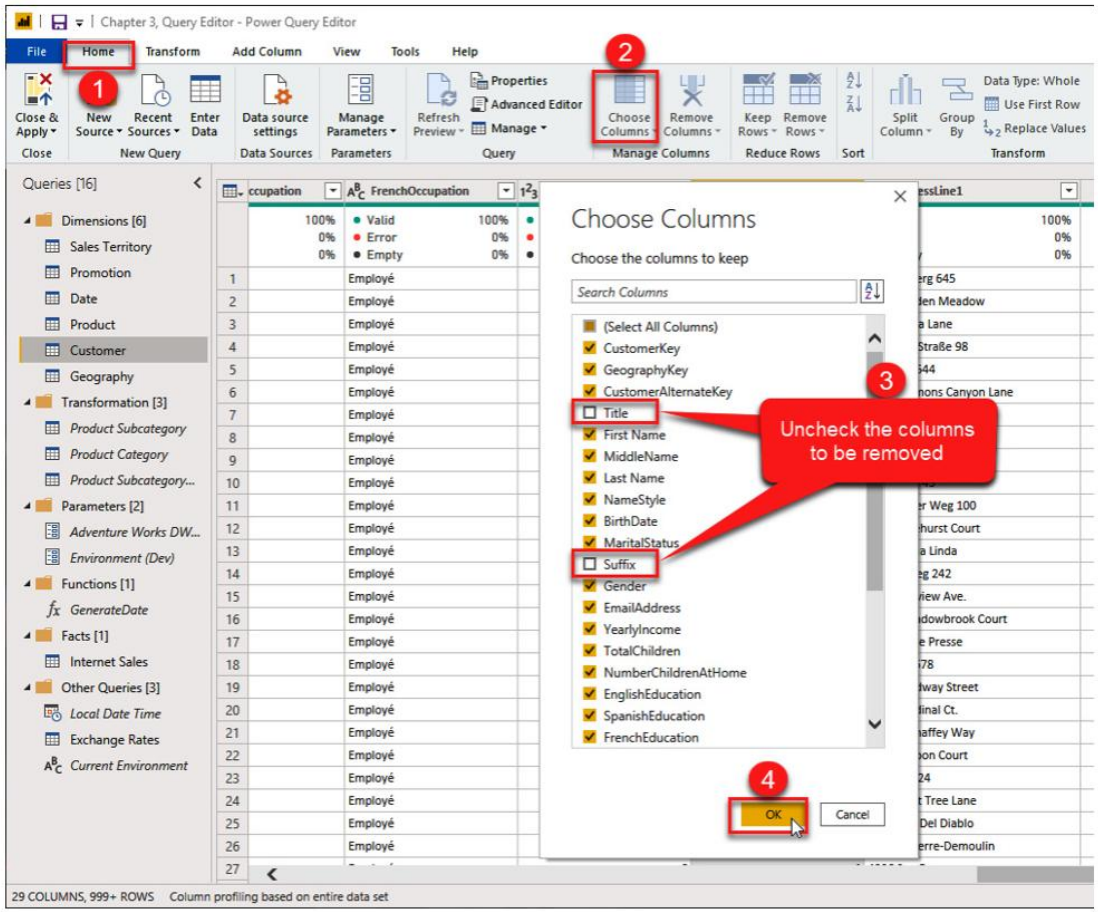

Figure 3.28 – Removing columns

Column distribution

Column distribution is another feature that provides more information about the data distribution, distinct values, and unique values. The **Column distribution** information can help data modelers with the cardinality of a column. **Column cardinality** is an essential topic in data modeling, especially for memory management and data compression.

NOTE

The general rule of thumb is that we want to get lower cardinality. When the xVelocity engine loads data into the data model, it better compresses the low-cardinality data. Therefore, the columns with lower cardinality have less (or no) unique values.

We can consider whether we load that column into the model with the **Column distribution** feature or remove it from the model. Removing unnecessary columns can potentially help us with file-size optimization and performance tuning.

To enable the **Column distribution** feature, click the corresponding feature from the **View** tab, as shown in the following screenshot:

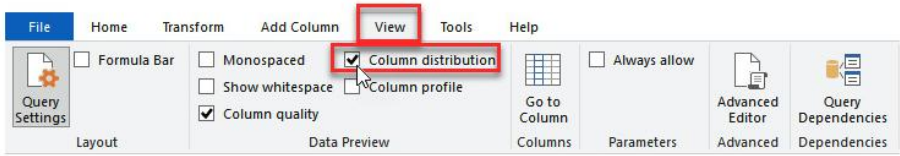

Figure 3.29 – Enabling Column distribution feature from Power Query Editor

After enabling this feature, a new box is added under the **Column quality** box visualizing the column distribution. If you hover over the **Distribution Box**, a flyout menu shows some more information about the column distribution, as depicted in the following screenshot:

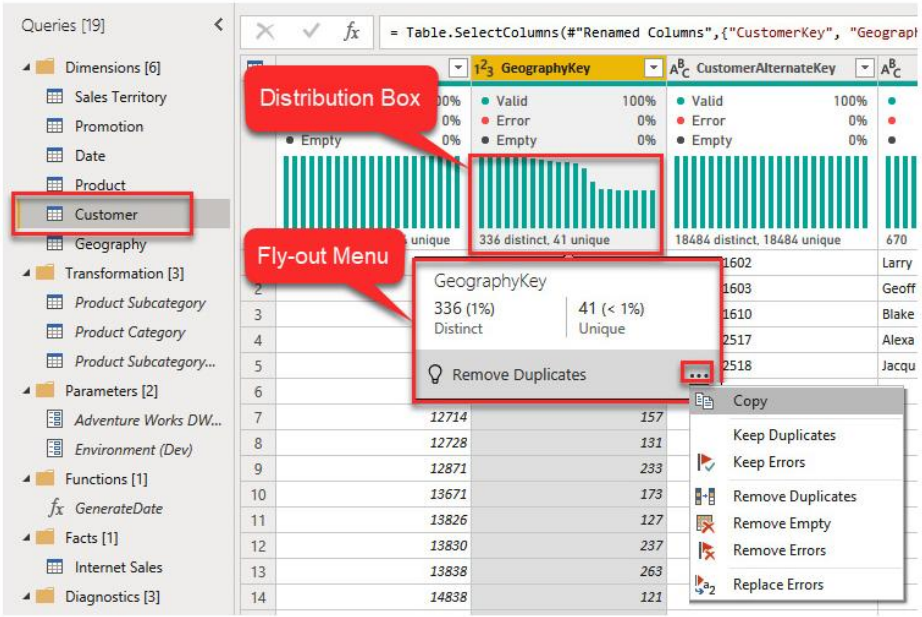

Figure 3.30 – Distribution box flyout menu

We can copy the distribution data from the flyout menu and take some other appropriate actions, as shown in the preceding screenshot.

Let's look at the **Column distribution** feature in action with a real-world scenario.

In **`Chapter 3, Query Editor.pbix`**, look at the **`Customer`** table. The **`Customer`** table is wide and tall. We want to nominate some columns for removal, to be discussed with the business, to optimize the file size and memory consumption:

1. Select the **`Customer`** table from the **Queries**.
2. Set the **Column profiling** to be calculated based on the entire dataset.
3. Quickly scan through the **Column Distribution** boxes of the columns to identify high cardinality columns.

These are the columns with high cardinality:

- **`CustomerKey`**
- **`CustomerAlternateKey`**
- **`EmailAddress`**
- **`Phone`**

These columns are highlighted in the following screenshot:

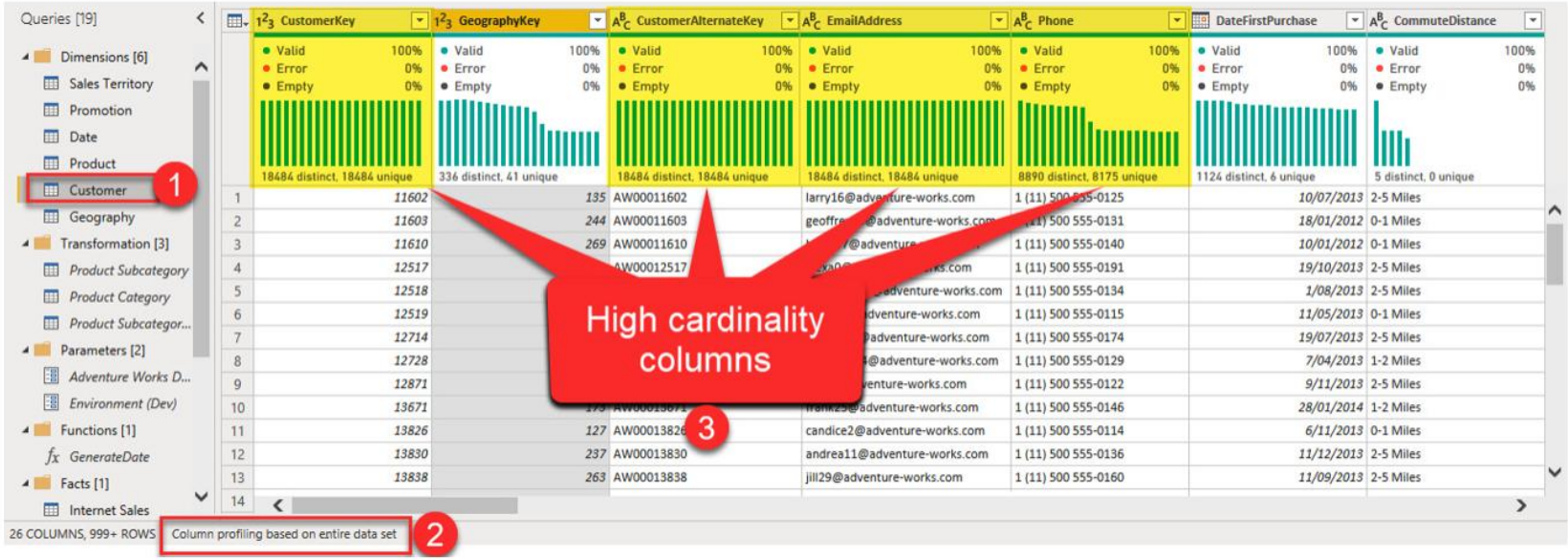

Figure 3.31 – Identifying high-cardinality columns in the Customer table

The **CustomerKey** column is not a candidate for removal as it participates in the relationship between the **Internet Sales** and **Customer** tables in the data model. We can remove the **CustomerAlternateKey** column. This is an excessive column with very high cardinality (100% unique values), and it also does not add any value from a data-analysis point of view. The two other columns are excellent candidates to discuss with the business to see if we can remove them from the **Customer** table.

If we remove all three columns, we can reduce the file size from 2,485 **kilobytes** (**KB**) to 1,975 KB. This is a significant saving in storage, especially in larger data models.

Column profile

So far, we have looked at the **Column quality** and **Column distribution** features. We can also enable the **Column profile** feature to see more information about a selected column's values. To enable this feature, tick the **Column profile** box under the **View** tab, as illustrated in the following screenshot:

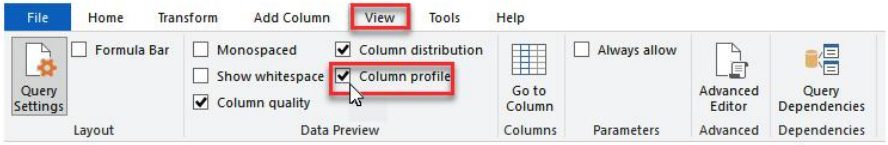

Figure 3.32 – Enabling Column profile from Power Query Editor

As the preceding screenshot shows, we can see **Column Statistics** and **Value Distribution** by enabling the **Column profile** feature. We can hover over the values to see the count number of that value and its percentage in a flyout menu. We can also take some actions on the selected value by clicking the ellipsis button on the flyout menu's bottom right, as illustrated in the following screenshot:

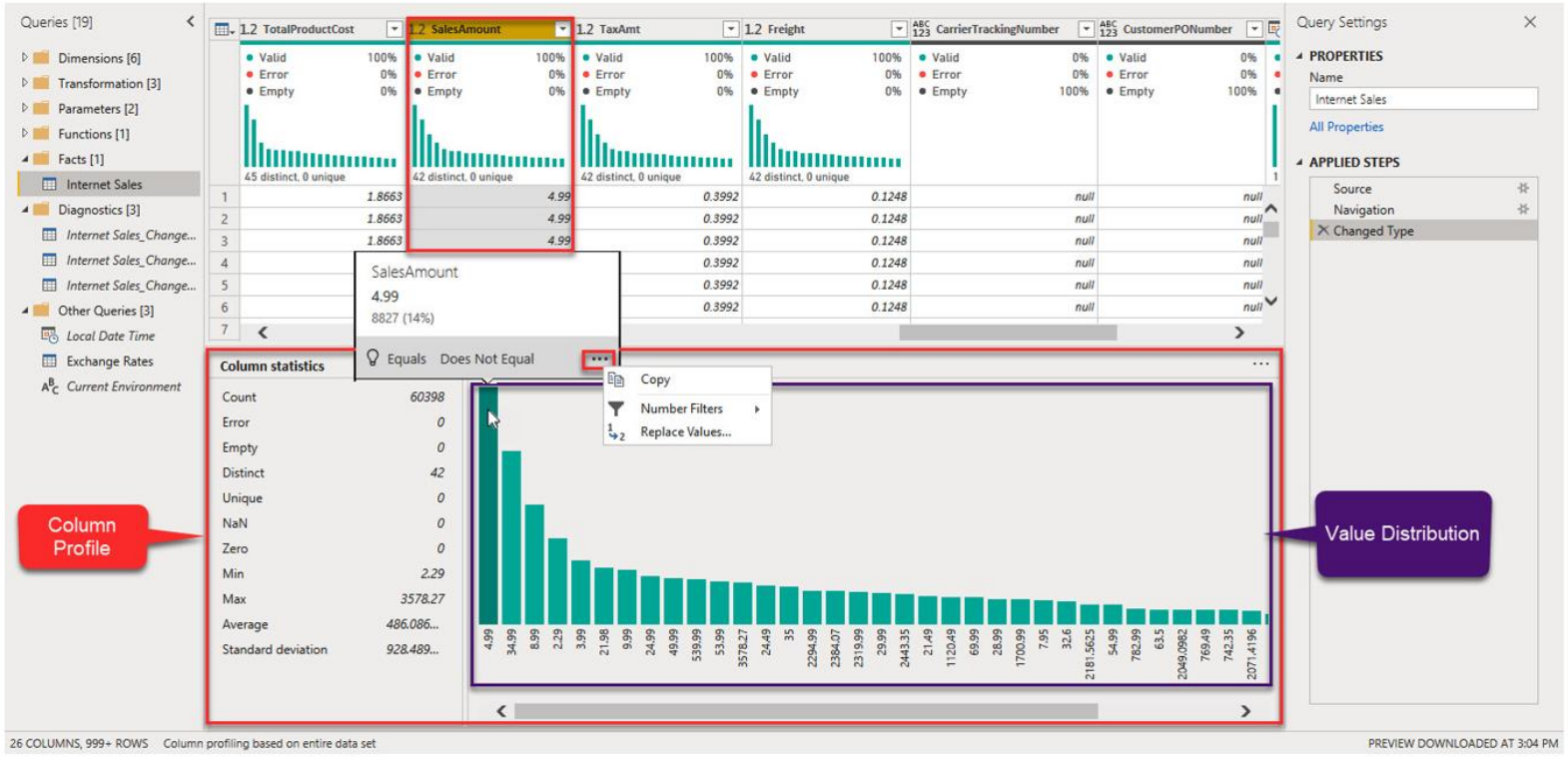

Figure 3.33 – Column profile

So far, we have discussed how the Power Query formula language works and how to use **Power Query Editor** in Power BI. We also looked at some features that can help us with our data modeling. In the next few sections, we discuss some more implementation-related topics such as query parameters, and we go through some real-world scenarios using these parameters.

Understanding query parameters

One of the most valuable features is the ability to define **query parameters**. We can then use defined query parameters in various cases. For instance, we can create a query referencing a parameter to retrieve data from different datasets, or we can parameterize filter rows. With query parameters, we can parameterize the following:

- **Data Source**
- **Filter Rows**
- **Keep Rows**
- **Remove Rows**
- **Replace Rows**

In addition, we can load the parameters' values into the data model so that we can reference them from measures, calculated columns, calculated tables, and report elements if necessary.

We can easily define a query parameter from **Power Query Editor**, as follows:

1. Click **Manage Parameters**.
2. Click **New**.
3. Enter a name.
4. Type in some informative description that helps the user to understand the purpose of the parameter.
5. Ticking the **Required** box makes the parameter mandatory.
6. Select a type from the drop-down list.
7. Select a value from the **Suggested Values** drop-down list.
8. Depending on the suggested values selected in the previous step, you may need to enter some values (This is shown in *Figure 3.34*). If you selected **Query** from the **Suggested Values** drop-down list, then you need to select a query in this step.
9. Again, depending on the selected suggested values, you may/may not see the default value. If you selected **List of values,** then you need to pick a default value.
10. Pick or enter a value as the **Current Value**.
11. Click **OK**.

The preceding steps are illustrated in the following screenshot:

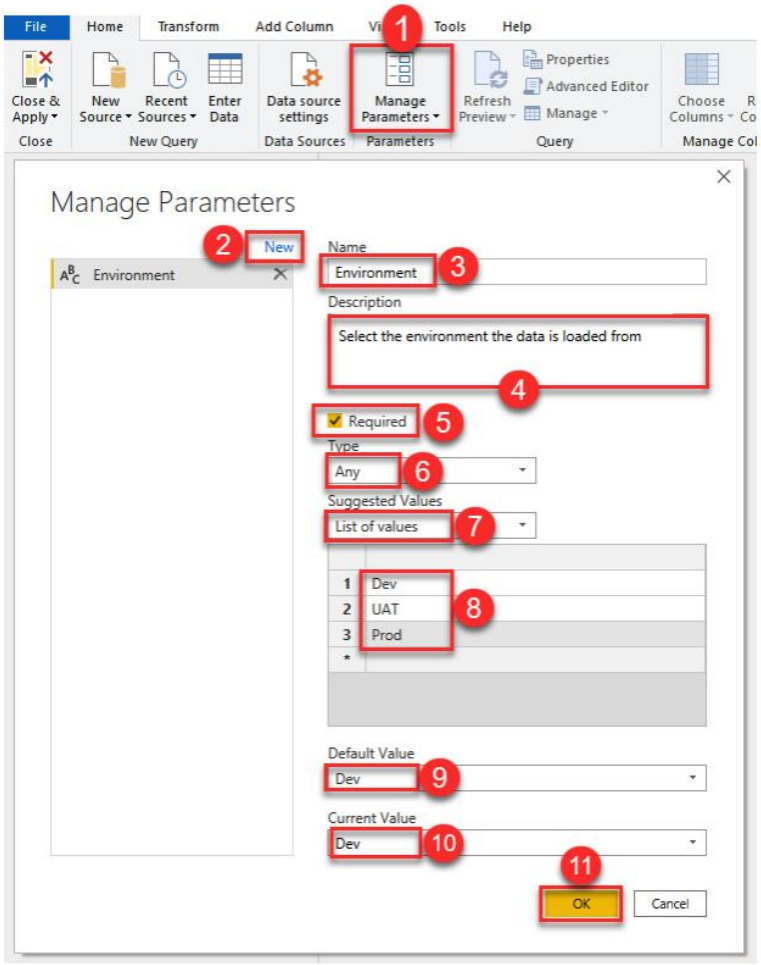

Figure 3.34 – Defining a new query parameter

NOTE

It is best practice to avoid hardcoding the data sources by parameterizing them. Some organizations consider their data source names and connection strings as sensitive data. They also only allow PBIT files to be shared within the organization or with third-party tools, as PBIT files do not contain data. So, if the data sources are not parameterized, they can reveal server names, folder paths, SharePoint ***Uniform Resource Locators*** *(****URLs****), and so on. Using query parameters with* ***Suggested Values*** *of* ***Any*** *value makes perfect sense to avoid any data leakage.*

The number of use cases for query parameters is quite vast. Let's have a look at a real-world scenario when using query parameters comes in handy.

In this scenario, we want to parameterize the data sources. Parameterizing a data source is helpful in many ways, from connecting to different data sources to loading different combinations of columns. One of the most significant benefits of parameterizing data sources is to avoid hardcoding the server names, database names, files, folder paths, and so on.

The business has a specific BI governance framework that requires separate **Development** (**Dev**), **User Acceptance Testing** (**UAT**), and **Production** (**Prod**) environments. The business requires us to produce a sales analysis report on top of the enterprise data warehouse available in SQL Server. We have three different servers, one for each environment, hosting the data warehouse. While a Power BI report is in the development phase, it must connect to the Dev server getting the data from the Dev database. When the report is ready to go for testing in the UAT environment, both the server and the database must be switched to the UAT environment. When the UAT people have done their testing and the report is ready to go live, we need to switch the server and the database again to point to the Prod environment. To implement this scenario, we need to define two query parameters. One keeps the server names, and the other keeps the database names. Then, we set all relevant queries to use those query parameters. This will be much easier to manage if we start using the query parameters from the beginning of the project. But don't worry—if you currently have a Power BI report to hand and you would like to parameterize the data sources, the process is easy. Once you set it, you do not need to change any codes in the future to switch between different environments. Let's create a new query parameter, as follows:

1. In **Power Query Editor**, click **Manage Parameters**.
2. Click **New**.
3. Enter the parameter name as **Server Name**.
4. Type in some description.
5. Tick the **Required** field
6. Select the **Type** as **Text** from the drop-down list.
7. Select **List of values** from the **Suggested Values** drop-down list.
8. Enter the server names in the list.
9. Select the **devsqlsrv01\edw** as the **Default Value**.
10. Pick the **devsqlsrv01\edw** again as the **Current Value**.
11. Click **OK**.

The preceding steps are highlighted in the following screenshot:

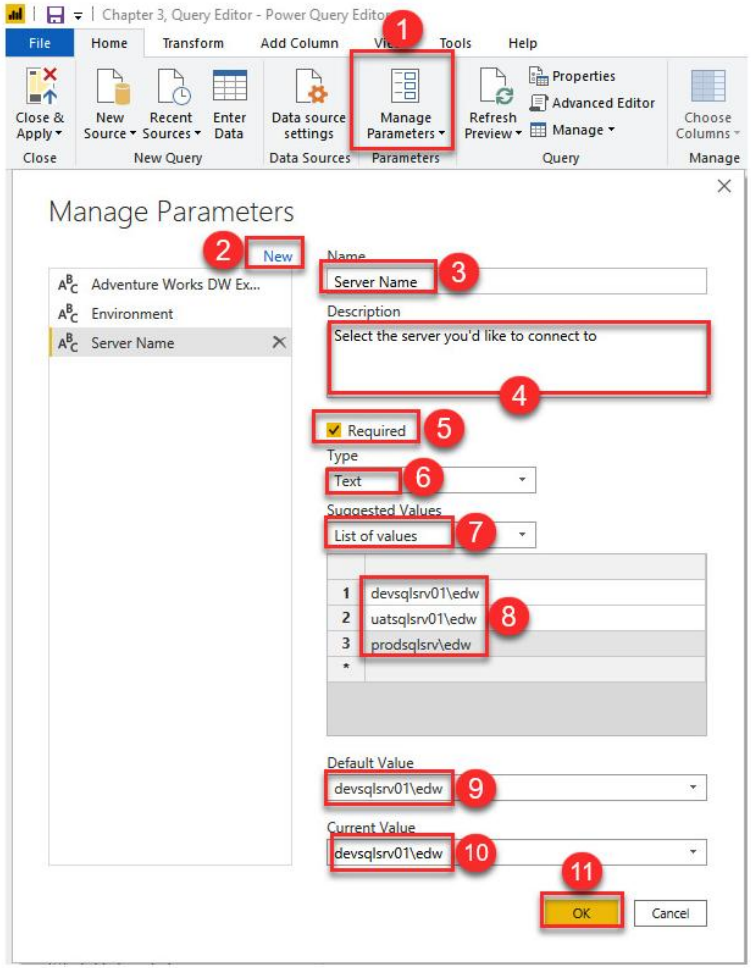

Figure 3.35 – Creating a query parameter holding the server names for different environments

We need to go through the same steps to create another query parameter for the database names. If the database names are the same, then we can skip this step. The following screenshot shows the other parameter we created for the database names:

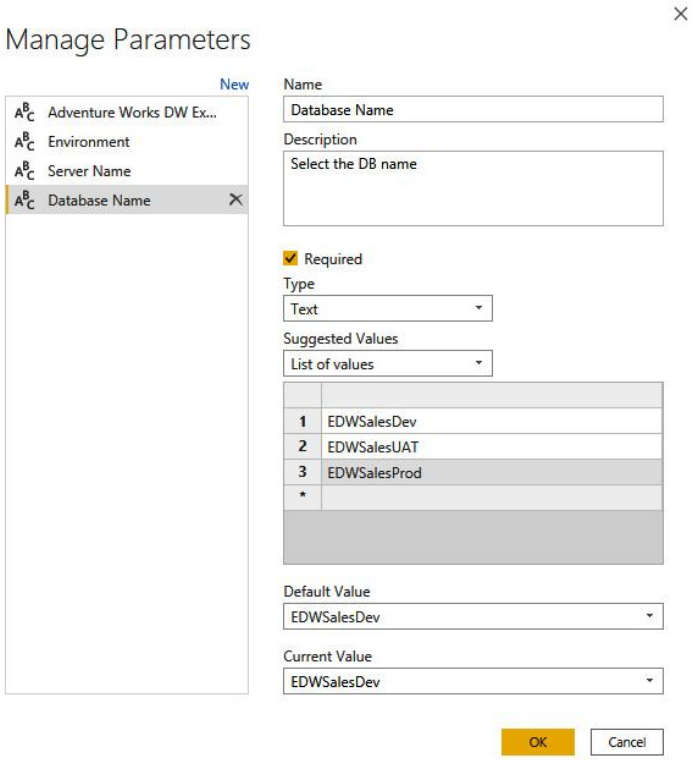

Figure 3.36 – Creating a query parameter holding the database names for different environments

If we already have some queries, then we need to modify the data sources as follows:

1. Click a query to parameterize.
2. Click the gear icon of the first step, **Source**.
3. Select **Parameter** from the **Server** drop-down.
4. Select the **Server Name** parameter from the **server parameters** drop-down.
5. Select **Parameter** again from the **Database** drop-down.
6. Select the **Database Name** parameter from the **database parameters** drop-down.
7. Click **OK**.

The preceding steps are highlighted in the following screenshot:

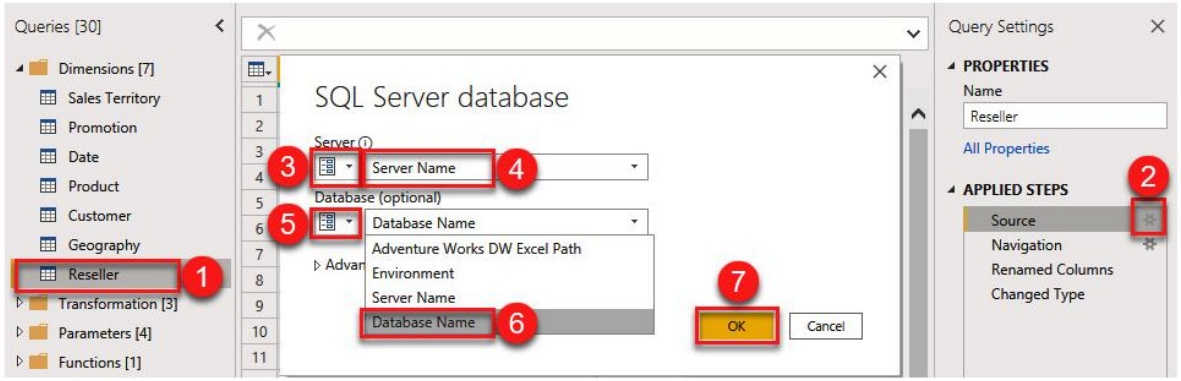

Figure 3.37 – Parameterizing a data source

We need to go through similar steps to parameterize other queries. After we have finished the parameterization, we only need to change the parameters' values whenever we want to switch the data sources. To do so from **Power Query Editor**, proceed as follows:

1. Click the **Manage Parameters** drop-down button.
2. Click **Edit Parameters**.
3. Select the UAT **Server Name**.
4. Select the UAT **Database Name**.
5. Click **OK**.

The preceding steps are highlighted in the following screenshot:

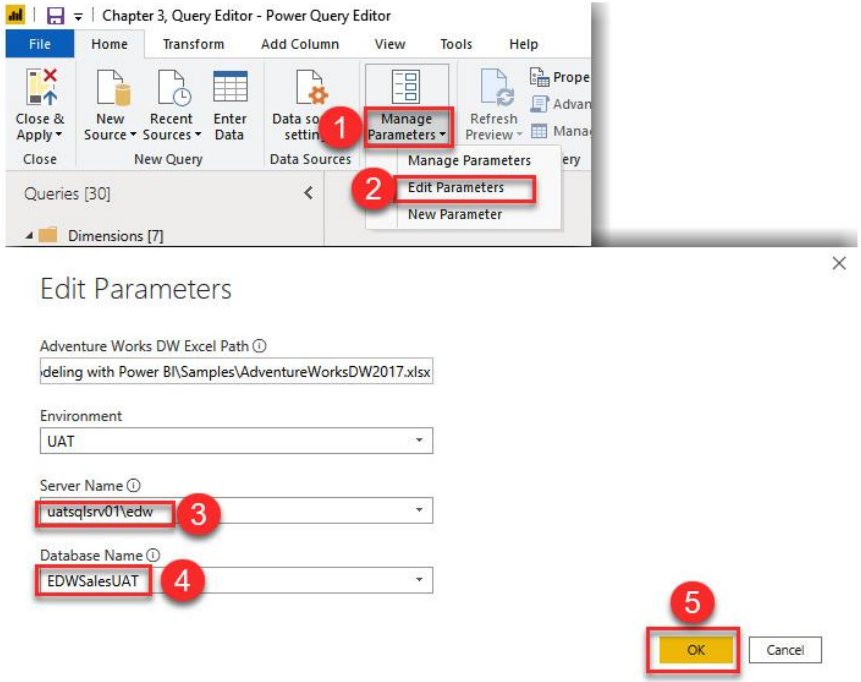

Figure 3.38 – Changing query parameters' values from Power Query Editor

We can also change the parameters' values from the main Power BI Desktop window, as follows:

1. Click the **Transform data** drop-down button.
2. Click **Edit parameters**.
3. Select the UAT **Server Name**.
4. Select the UAT **Database Name**.
5. Click **OK**.

The preceding steps are highlighted in the following screenshot:

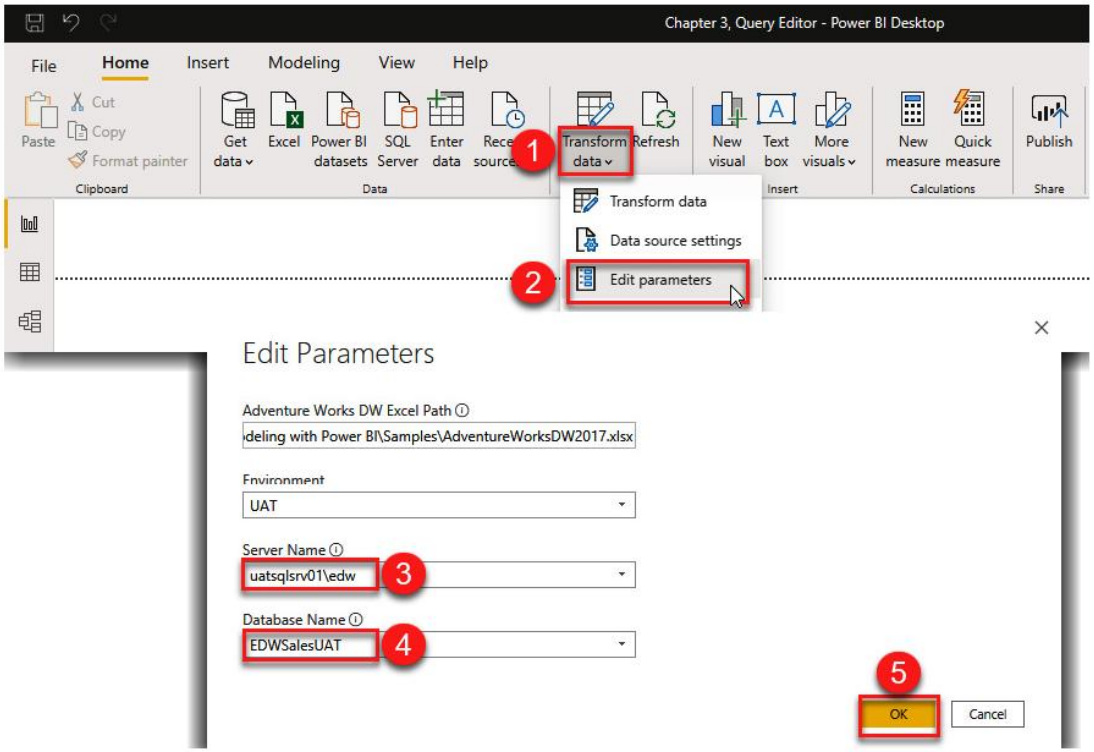

Figure 3.39 – Changing query parameters' values from the main Power BI Desktop window

Understanding custom functions

In many cases, we may face a situation where we repeatedly need to calculate something. In such cases, it makes absolute sense to create a **custom function** that takes care of all the calculation logic needed. After defining a custom function, we can invoke this function many times. As stated in the *Introduction to Power Query M formula language in Power BI* section, under *Function value*, we can create a custom function by putting the list of parameters (if any) in parentheses, along with the output data type and the goes-to symbol **=>**, followed by a definition of the function.

The following example shows a straightforward form of a custom function that gets a date input and adds one day to it:

```
SimpleFunction = (DateValue as date) as date =>
Date.AddDays(DateValue, 1)
```

We can simply invoke the preceding function as follows:

```
SimpleFunction(#date(2020,1,1))
```

The result of invoking the function is **2/01/2020**.

We can define a custom function as an inline custom function and invoke it within a single query in the **Advanced Editor**. The following screenshot shows how we use the preceding code to define **SimpleFunction** as an inline custom function and invoke the function within the same query:

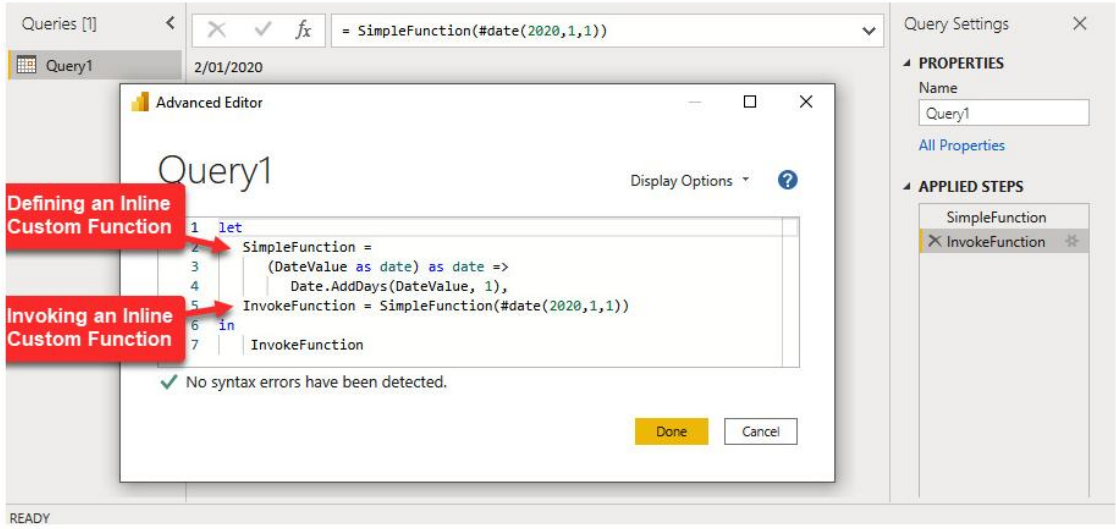

Figure 3.40 – Defining and invoking inline custom functions

Let's take a step further and look at a real-world scenario to see how we can save a massive amount of development time by creating a custom function.

We are tasked to build a data model on top of 100 tables. The initial investigation shows that the tables have between 20 and 150 columns. The column names are in camel case and are not user-friendly. We have two options: to manually rename every single column, or to find a way to rename all table columns in one go. While we have to do this for all tables, we can still save a lot of development time by renaming each table's columns in a single step. To achieve this goal, we can create a custom function. We will invoke that function in all tables later.

Let's look at the original column names in one table. The following screenshot shows the original column names of the **Product** table, which are not user-friendly. Note that in the status bar, we can see the number of columns a table has—in this case, the **Product** table has 37 columns. So, it would be very time-consuming if we were to manually rename every column and split the words to make them more readable:

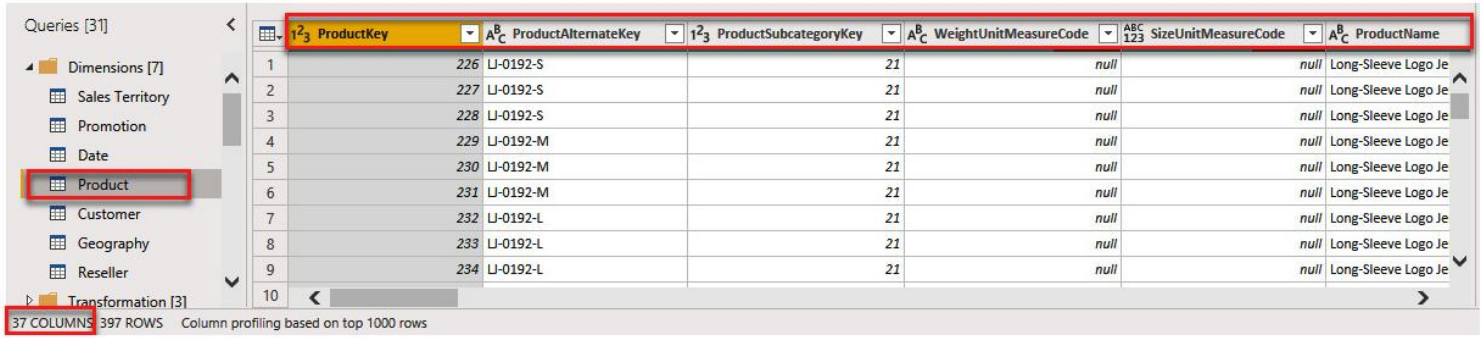

Figure 3.41 – Original column names in the Product table

In our sample, splitting the column names when the character case is transitioning from lowercase to uppercase would be enough. Proceed as follows:

1. In **Power Query Editor**, create a blank query by clicking the **New Source** drop-down button and selecting **Blank Query**, as illustrated in the following screenshot:

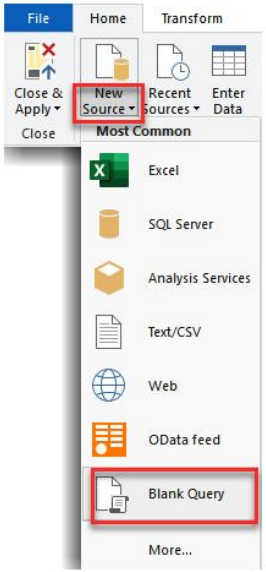

Figure 3.42 – Creating a blank query from Power Query Editor

2. Open **Advanced Editor**.
3. Copy and paste the script shown next in *Step 4* in the **Advanced Editor**, and click **OK**.
4. Rename the query to **`fnRenameColumns`**, as shown in the following code snippet:

```
let
    fnRename = (ColumnName as text) as text =>
        let
            SplitColumnName = Splitter.SplitTextByCharacterTransition({"a".."z"}, {"A".."Z"})(ColumnName)
        in
    Text.Combine(SplitColumnName, " ")
in
    fnRename
```

The following screenshot shows what the created function looks like in **Power Query Editor**:

Figure 3.43 – Creating a custom function

The preceding function accepts text values, then splits the text value to a list of texts whenever a case transition from lowercase to uppercase happens within the text. Then, we combine the split text and use a space character between the text. To understand how the preceding custom function works, we need to read through the documentation of the **`Splitter.SplitTextByCharacterTransition()`** function on the Microsoft Docs website.

Note that the **`Splitter.SplitTextByCharacterTransition()`** function returns a function, therefore the **`Splitter.SplitTextByCharacterTransition({"a".."z"}, {"A".."Z"})(ColumnName)`** part of the preceding script applies the **`SplitTextByCharacterTransition`** function to the function input parameter, which is **`ColumnName`**, resulting in a list of texts.

Now, let's call the **fnRenameColumns** function in the **Product** table, as follows:

1. Enable **Formula Bar** if it is not enabled already from the **View** tab, as shown in the following screenshot:

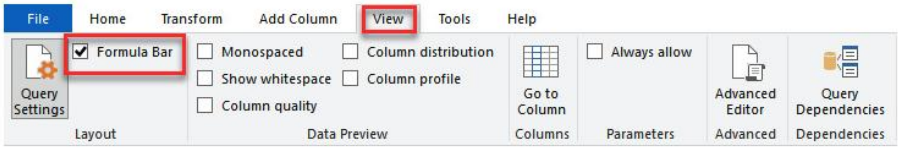

Figure 3.44 – Enabling the Formula Bar option in Power Query Editor

2. Select the **Product** table from the **Queries** pane.
3. From the **Formula Bar**, click the **Add Step** button (fx) to add a new step. This is quite handy as it shows the last step name, which we will use next. The following screenshot shows what the new added step looks like:

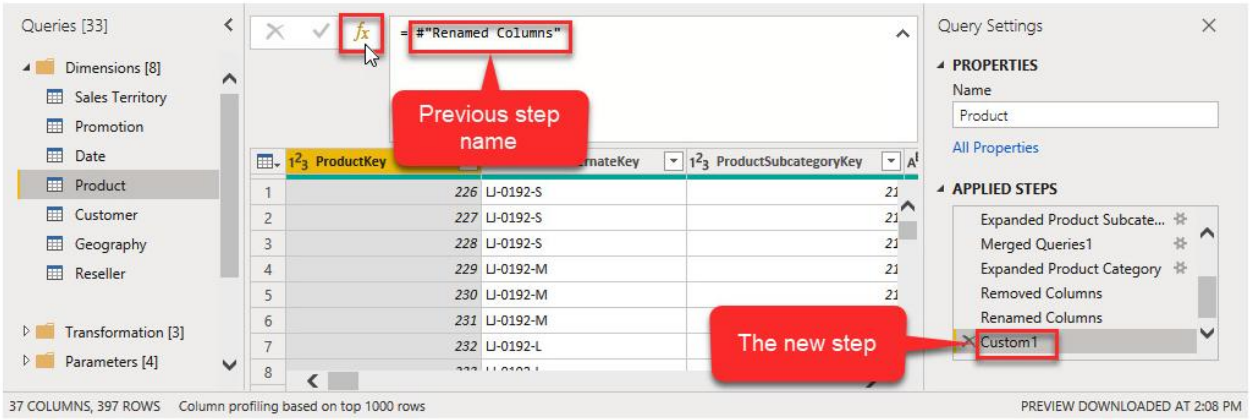

Figure 3.45 – Adding a new step from the Formula Bar

4. We now use the **`Table.TransformColumnNames()`** function, which transforms column names of a given table by a given name-generator function. This table comes from the previous step, and the name-generator function is the **`fnRenameColumns`** function we created earlier. So, the function will look like this:

```
Table.TransformColumnNames(#"Renamed Columns", fnRenameColumns)
```

5. After committing to the running of this step, all columns in the **`Product`** table rename immediately, as the following screenshot shows:

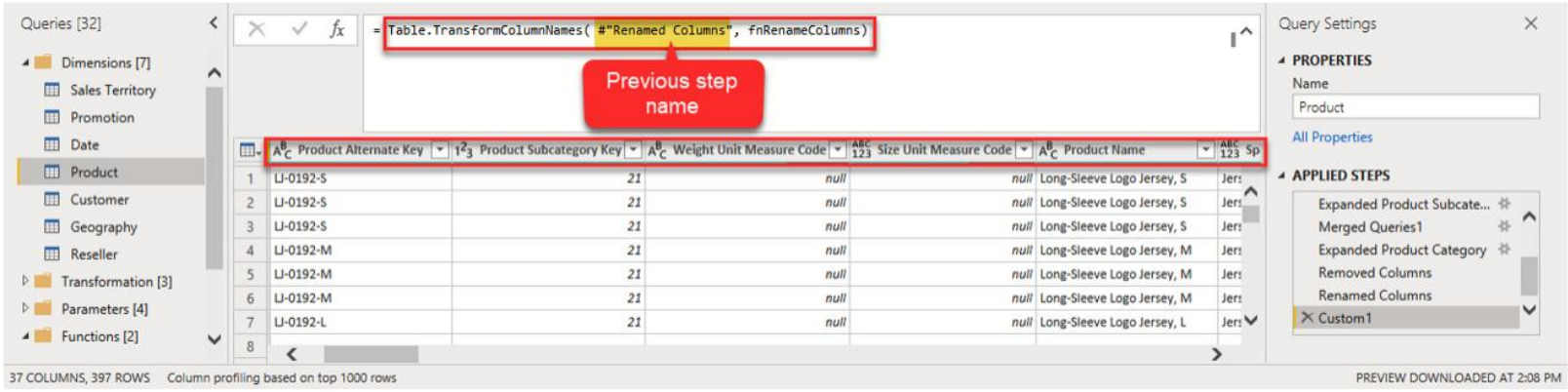

Figure 3.46 – Renaming all columns at once

6. The very last step is to rename the new step to something more meaningful. To do so, right-click the step and click **Rename** from the context menu and type in a new name, as shown in the following screenshot:

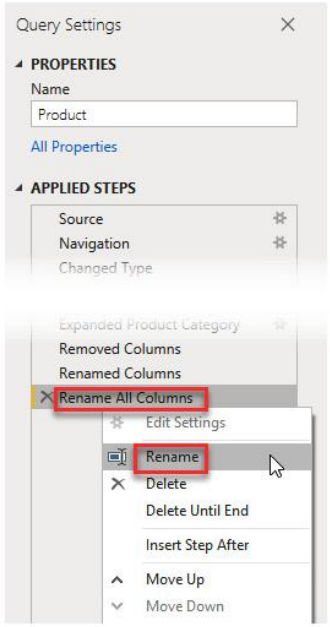

Figure 3.47 – Renaming a query step

Recursive functions

We can reference a function from the function itself, which makes that function a recursive function. **Factorial** is a mathematical calculation that multiplies all positive whole numbers from any chosen number down to 1. In mathematics, an exclamation mark shows a factorial calculation (for example, **n!**). The following formula shows the mathematical calculation of **Factorial**:

n! = n * (n - 1)!

As the preceding calculation suggests, a **Factorial** calculation is a recursive calculation. In a **Factorial** calculation, we can choose a positive integer (an integer larger than **0**) when **0** is an exception; if **0** is chosen, then the result is **1**. Here are some examples to make the **Factorial** calculation clearer:

```
10! = 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 =  3,628,800
5! = 5 * 4 * 3 * 2 * 1 = 120
1! = 1
0! = 1
```

To write a recursive function, we need to use an **@** operator to reference the function within itself. For example, the following function calculates the factorial of a numeric input value:

```
let
   Factorial =
        (ValidNumber as number) as number =>
            if ValiedNumber < 0
            then error "Negative numbers are not allowed to calculate Factorial. Please select a positive number."
            else
                if ValidNumber = 0
                then 1
                else ValidNumber * @Factorial(ValiedNumber - 1)
in
    Factorial
```

As you see in the preceding code block, we raise an error message if the input value is not valid. We return **1** if the input is **0**; otherwise, we calculate the **Factorial** recursively.

The following screenshot shows the result of invoking a **Factorial** function with an invalid number:

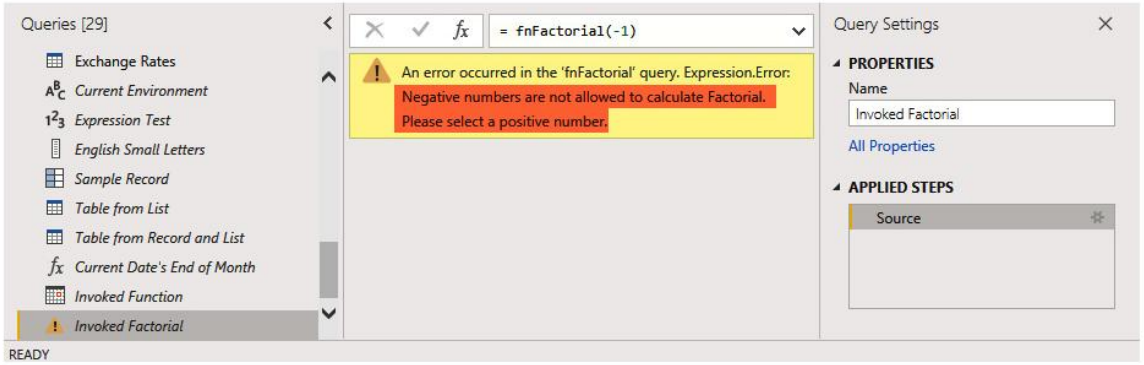

Figure 3.48 – Raising an error when invoking a Factorial function with an invalid value

The following screenshot shows the result of invoking a **`Factorial`** function to calculate **`10!`**:

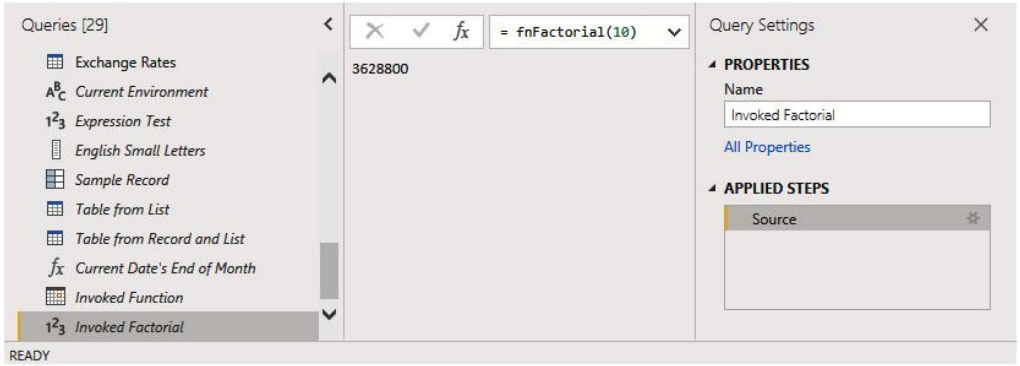

Figure 3.49 – The result of invoking a Factorial function to calculate 10!

Summary

In this chapter, we introduced different aspects of the Power Query M formula language and looked at how we can use **Power Query Editor**. We also looked at some real-world scenarios and challenges that can directly affect our productivity and learned how to manage our data preparation step more efficiently.

In the next chapter, we will discuss getting data from various data sources and several connection modes, and how they can affect our data modeling.