Chapter 6 of "Expert Data Modeling with Power BI" by Soheli Bakhshi is titled "Star Schema Preparation in Power Query Editor." In this chapter, the author focuses on the use of Power Query Editor to transform data and create a star schema for use in Power BI. Here are some detailed notes on the chapter:

The star schema is a widely-used data model that is optimized for querying data in a business intelligence (BI) system. The schema consists of a central fact table that is connected to several dimension tables. The fact table contains the data to be analyzed, while the dimension tables provide context and additional information.

Power Query Editor is a powerful tool for transforming data from various sources and preparing it for analysis in Power BI. The author explains the key features and functions of Power Query Editor and provides step-by-step instructions for preparing data using this tool.

The chapter includes several practical examples of how to use Power Query Editor to create a star schema. For example, the author provides instructions for merging multiple tables into a single table, transforming column data, splitting tables into multiple dimension tables, and creating relationships between tables.

The author emphasizes the importance of data quality and provides guidance on how to identify and fix common data quality issues, such as missing or incorrect values, duplicates, and inconsistencies.

The chapter also covers some advanced techniques for data modeling in Power BI, such as creating calculated columns and measures using the Data Analysis Expressions (DAX) language. The author provides examples of how to use DAX to create complex calculations and business metrics.

Overall, this chapter provides a detailed and practical guide to using Power Query Editor to prepare data for analysis in Power BI. The author covers all of the key concepts and features of the tool, and provides step-by-step instructions and practical examples to help readers get started. By following the guidance in this chapter, readers can create a robust and efficient star schema that will enable them to derive insights and make informed business decisions.

Chapter 6 of "Expert Data Modeling with Power BI" by Soheli Bakhshi provides detailed guidance on how to prepare data for the star schema in Power Query Editor. Here are some real-world examples of how the concepts discussed in this chapter could be applied:

1. Sales Analysis: A retail organization could use Power BI to analyze their sales data. By using the star schema, they could break down sales by various dimensions, such as product, store, and time. They could also include additional dimensions, such as customer demographics, to gain further insights into their sales trends and behavior.

2. Healthcare Analytics: A healthcare organization could use Power BI to analyze patient data. The star schema could be used to break down patient data by dimensions such as diagnosis, treatment, and patient demographics. This could help the organization identify patterns in patient behavior and treatment outcomes, which could inform future treatment plans.

3. Financial Reporting: A financial organization could use Power BI to analyze their financial data. By using the star schema, they could break down financial data by dimensions such as account type, period, and region. This could help the organization identify trends and make informed decisions about their financial strategy.

Overall, the star schema is a powerful tool for analyzing data in Power BI, and the techniques outlined in Chapter 6 can be applied to a wide variety of industries and use cases.

Chapter 6: Star Schema Preparation in Power Query Editor

We learned about some common data preparation steps in the previous chapter, including data type conversion, split column, merge columns, adding a custom column, and filtering rows. We also learned how to create summary tables using the Group By feature, appending data, and merging queries.

This chapter will use all the topics we discussed in the past few chapters and help you learn how to prepare a Star Shema in Power Query Editor. Data modeling in Power BI starts with preparing a Star Schema. In this chapter, we'll use the **`Chapter 6, Sales Data.xlsx`** file, which contains flattened data. This is a common scenario many of us have faced; we get a set of files containing data that have been exported from a source system, and we need to build a report to answer business questions. Therefore, having the required skills to build a Star Schema on top of a flat design comes in handy.

In this chapter, we will cover the following topics:

- Identifying dimensions and facts
- Creating Dimensions tables
- Creating fact tables

Identifying dimensions and facts

When we are talking about a Star Schema, we are automatically talking about **dimensions** and **facts**. In a Star Schema model, we usually keep all the numeric values in fact tables and put all the descriptive data in the **`Dimension`** tables. But not all numeric values fall into fact tables. A typical example is Product Unit Price. If we need to do some calculations regarding the Product Unit Price, it is likely a part of our fact table. However, if the Product Unit Price is used to filter or group the data, it is probably a part of a dimension.

Designing a data model in a Star Schema is not possible unless we have a level of understanding of the data. This chapter aims to look at the **`Chapter 6, Sales Data.xlsx`** data and create a Star Schema.

As we mentioned earlier, to create a Star Schema, we need to have a closer look at the data we are going to model and identify dimensions and facts. In this section, we will try to find these dimensions and facts. We will also discuss whether we will wrap the dimensions in separate tables or not and why. In the following few sections, we will look at this in more detail and implement the identified dimensions and facts.

Before we continue, let's get the data from our sample file into Power Query Editor, as shown in the following screenshot. In the past few chapters, we learned how to get the data from an Excel file; therefore, so we'll skip explaining the Get Data steps:

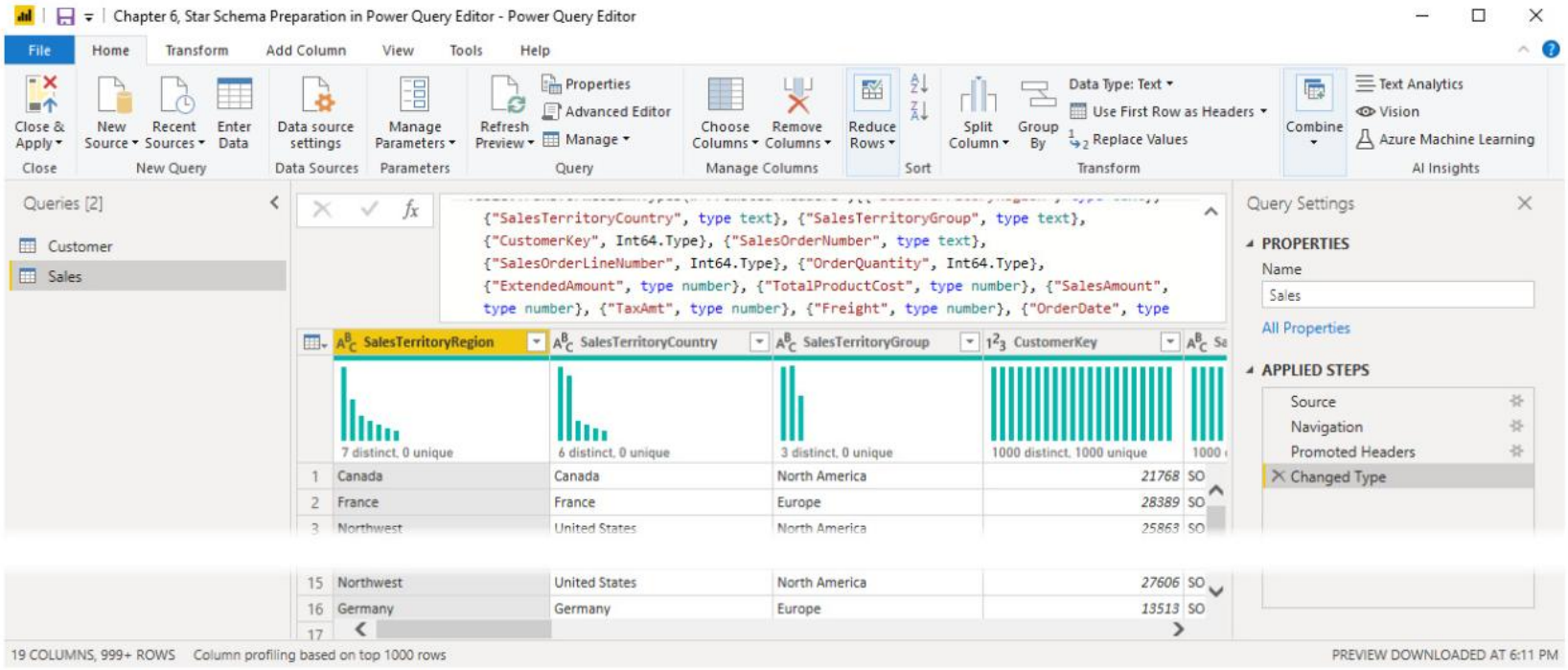

Figure 6.1 – Power Query Editor connected to the Chapter 6, Sales Data.xlsx sample data

Looking at the **`Chapter 6, Sales Data.xlsx`** sample data that we loaded into Power Query Editor, we must study the data and find out the following:

- The number of tables in the data source
- The linkages between existing tables
- The lowest required grain of **`Date`** and **`Time`**

The preceding points are the most straightforward initial points we must raise with the business within the initial discovery workshops. These simple points will help us understand the scale and complexity of the work. In the following few sections, we'll look at the preceding points in more detail.

Number of tables in the data source

Our sample file contains two sheets that translate into two **base tables**: **Sales** and **Customer**. We use the word **base tables** to extract the dimensions and facts from them; therefore, they will not be loaded into the model in their original shape. From a modeling perspective, there must be a linkage between the two tables, so we need to raise this with the business and study the data and see if we can find the linkage(s). The names of the tables are highlighted in the following screenshot:

Figure 6.2 – The Excel sample file contains two sheets that translate into two base tables in Power Query Editor

The crucial point is not to be tricked by the number of base tables. These tables can be wide and tall, and there may be many more dimension and fact tables once we've prepared the Star Schema model data. Our sample only contains two base tables, which then turn into five dimensions and one fact table once we've prepared the data for a Star Schema.

The linkages between existing tables

The columns of the two tables are as follows:

- **Sales**: **SalesTerritoryRegion**, **SalesTerritoryCountry**, **SalesTerritoryGroup**, **CustomerKey**, **SalesOrderNumber**, **SalesOrderLineNumber**, **OrderQuantity**, **ExtendedAmount**, **TotalProductCost**, **SalesAmount**, **TaxAmt**, **Freight**, **OrderDate**, **DueDate**, **ShipDate**, **Product**, **ProductCategory**, **ProductSubcategory**, **Currency**
- **Customer**: **CustomerKey**, **Title**, **FirstName**, **MiddleName**, **LastName**, **NameStyle**, **BirthDate**, **MaritalStatus**, **Suffix**, **Gender**, **EmailAddress**, **YearlyIncome**, **TotalChildren**, **NumberChildrenAtHome**, **EnglishEducation**, **EnglishOccupation**, **HouseOwnerFlag**, **NumberCarsOwned**, **AddressLine1**, **AddressLine2**, **Phone**, **DateFirstPurchase**, **CommuteDistance**, **City**, **StateProvinceCode**, **StateProvinceName**, **CountryRegionCode**, **EnglishCountryRegionName**, **PostalCode**, **SalesRegion**, **SalesCountry**, **SalesContinent**

By looking at these columns and studying their data, we can see that the **CustomerKey** column in both tables contains the trivial linkage between them. But there are more. The following are some other potential linkages:

- Linkage over geographical data such as **Region**, **Country**, **Territory Group**, **State/Province**, **City**, **Address**, and **Postal Code**
- Linkage over product data such as **Product Category**, **Product Subcategory**, and **Product**
- Linkage over sales order data such as **Sales Order** and **Sales Order Line**
- Linkage over **Date** and **Time** such as **Order Date**, **Due Date**, and **Ship Date**

Finding the lowest required grain of Date and Time

In most real-world cases, if not all cases, businesses must analyze data over **Date** or **Time** or both. In our example, the business need to analyze the data over both **Date** and **Time**. But we have to get more descriptive information about the level of **Date** and **Time** that the business requires to analyze the data. By studying the data, we find out that both the **Sales** and **Customer** tables have columns with **Date** or **DateTime** data types. The **Sales** table has three columns that contains the **DateTime** data type. The columns' data types are automatically detected by Power Query Editor. Studying the data more precisely shows that the time element of the values in both columns is always **12:00:00 AM**. Therefore, the data type of two columns, **DueDate** and **ShipDate**, must be **Date**, not **DateTime**. This means that the only column with the **DateTime** data type is the **OrderDate** column. The following screenshot shows that the data in the **OrderDate** column is stored in **Seconds**:

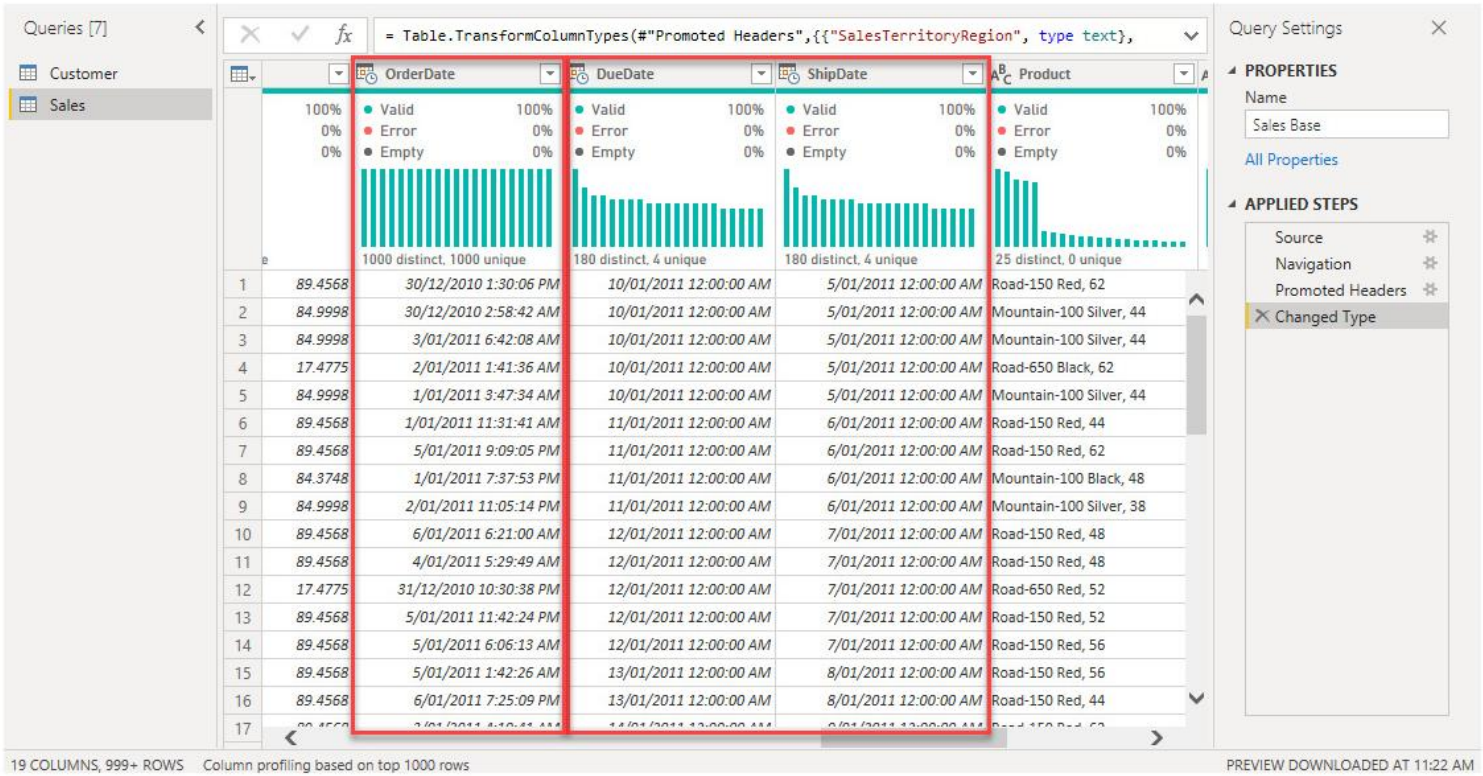

Figure 6.3 – Columns with the Date and DateTime data types in the Sales table

So, the grain of the **OrderDate** column can be in **Seconds**. We'll save this question and confirm it with the business later.

Let's also look at the **Customer** table. The **Customer** table also has two columns of the **Date** data type, **BirthDate** and **DateFirstPurchase**, as shown in the following screenshot:

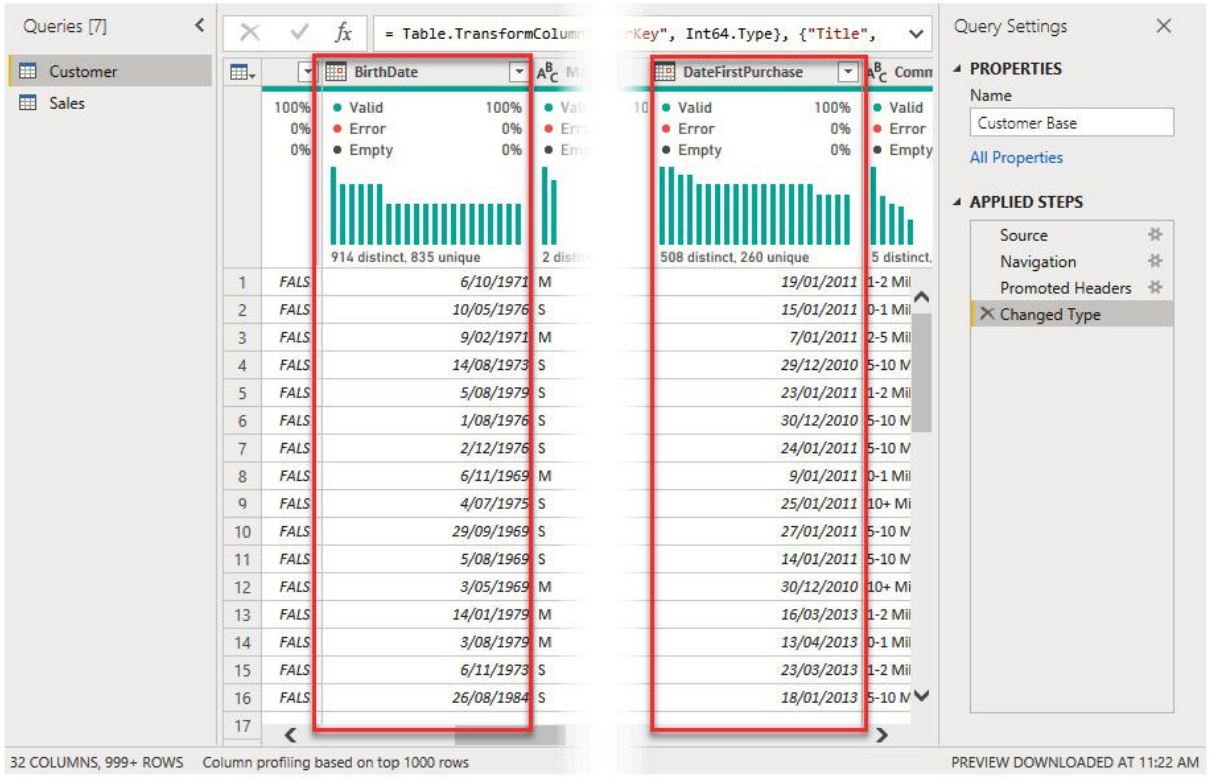

Figure 6.4 – The Date column in the Customer table

As the next step in our discovery workshops, we must ask the business to determine if they need to analyze the customer data surrounding date elements. We have to clarify the following with the business:

- Do they need to analyze the data over customers' **`BirthDate`**?
- Do they need to have the **`BirthDate`** data at different date levels (**`Year`**, **`Quarter`**, **`Month`**, and so on)?
- Do they need to analyze the data over **`DateFirstPurchase`**?
- Do they require to show the **`DateFirstPurchase`** data at various date levels?

In our imaginary discovery workshop with the business, we can see that they do not need to analyze the customers' birth dates or their first purchase dates. We can also see that they need to analyze **`OrderDate`** from the **`Sales`** table over **`Date`** at the **`Day`** level and over **`Time`** at the **`Minutes`** level. The data will also be analyzed by **`DueDate`** and **`ShipDate`** from the **`Sales`** table, over **`Date`** at the **`Day`** level.

With that, we will need to create a **`Date`** dimension and a **`Time`** dimension.

The next step is to identify the dimensions, their granularity, and their facts.

Defining dimensions and facts

To identify the dimensions and the facts, we have to conduct requirement gathering workshops with the business. We need to understand the business processes by asking WWWWWH questions; that is, **What**, **When**, **Where**, **Who**, **Why**, and **How**. This is a popular technique also known as **5W-1H**. The answers to these questions help us understand the business processes, which will help us identify our dimensions and facts. Let's have a look at some examples:

- The answer to the What question can be a product, item, service, and so on.
- The answer to the When question can be a date, time, or both and at what level, month, day, hour, minute, and so on.
- The answer to the Where question can be a physical location such as a store or warehouse, geographical location, and so on.

In the requirement gathering workshops, we try to determine what describes the business and how the business measures itself. In our scenario, let's imagine we've conducted several discovery workshops with the business, and we found out they require sales data for the following:

- Sales amount
- Quantity
- Costs (tax, freight, and so on)

The sales data must be analyzed by the following:

- Geography
- Sales Order
- Product
- Currency
- Customer
- Sales Demographic
- Date at the day level
- Time at the minute level

In the next few sections, we'll determine our dimensions and facts.

Determining the potential dimensions

To identify the dimensions, we generally look for descriptive data. Based on the results of the requirement gathering workshops, we look at our sample files. In both the **`Sales`** and **`Customer`** tables, we can potentially create the following dimensions:

Potential Dimension	Derived From	Dimension Attributes
Geography	Sales	SalesTerritoryRegion, SalesTerritoryCountry, SalesTerritoryGroup
Sales Order	Sales	SalesOrderNumber, SalesOrderLineNumber
Product	Sales	ProductCategory, ProductSubcategory, Product
Currency	Sales	Currency
Customer	Customer	CustomerKey, Title, FirstName, MiddleName, LastName, NameStyle, BirthDate, MaritalStatus, Suffix, Gender, EmailAddress, YearlyIncome, TotalChildren, NumberChildrenAtHome, Education, Occupation, HouseOwnerFlag, NumberCarsOwned, AddressLine1, AddressLine2, Phone, DateFirstPurchase, CommuteDistance
Sales Demographic	Customer	City, StateProvinceCode, StateProvinceName, CountryRegionCode, CountryRegionName, PostalCode, SalesRegion, SalesCountry, SalesContinent

Figure 6.5 – Potential dimensions derived from existing tables

Determining the potential facts

To identify the facts, we must look at the results of the requirement gathering workshops. With our current knowledge of the business, we have an idea of our facts. Let's look at the data in Power Query Editor and find the columns with the **`Number`** data type. Nevertheless, not all the columns with the **`Number`** data type contain facts. Facts must make sense to the business processes that were identified in earlier steps. With that in mind, in our exercise, the following list shows the potential facts:

Potential Fact	Derived From	Fact Columns
Sales	Sales	OrderQuantity, ExtendedAmount, TotalProductCost, SalesAmount, TaxAmt, Freight

Figure 6.6 – Potential facts derived from existing tables

NOTE

*In real-world scenarios, we conduct discovery workshops with **Subject Matter Experts** (**SMEs**) to identify the dimensions and facts when possible. But it is also a common scenario when the business supplies a set of source files and asks us to create analytical reports.*

Now that we have identified potential dimensions and facts, let's go a step further and start preparing the Star Schema. We have to take some actions before we can move on to the next steps.

As we mentioned earlier, we will not load the **`Sales`** and **`Customer`** tables into the data model in their current flat shape. Therefore, we will unload both tables. We explained how to unload tables in *Chapter 1, Introduction to Data Modeling in Power BI*, in the *Understanding denormalization* section.

The other step we should take is to change the columns' data types. If a **Changed Type** step is automatically added to **Applied Steps**, then we just need to inspect the detected data types and make sure the detected data types are correct. This is shown in the following screenshot:

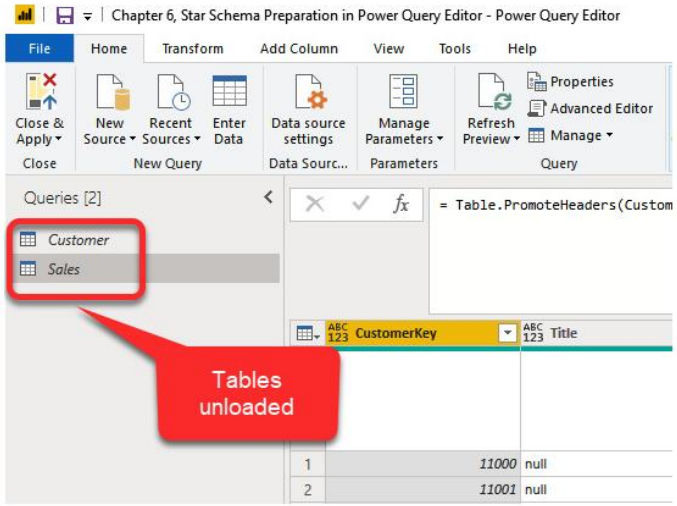

Figure 6.7 – Both the Sales and Customer tables unloaded

When the **Type Detection** setting is not set to **Never detect column types and headers for unstructured sources**, then Power Query automatically detects column headers and generates a **Changed Type** step for each table. We can configure this setting from Power Query Editor (or Power BI) by going to **Options and settings** -> **Options**, under **Type Detection**.

We can also control the auto-detecting data type behavior of Power Query at both the *Global* and *Current File* levels. The following steps show how to do that:

To configure the auto-detecting data type setting at the *Global* level, follow these steps:

1. Click the **File** menu.
2. Click **Options and settings**.
3. Click **Options**.
4. Click **Data Load**.
5. Under **Type Detection**, you have the option to select one of the following:
    - Always detect column types and headers for unstructured sources
    - Detect column types and headers for unstructured sources according to each file's settings
    - Never detect column types and headers for unstructured sources

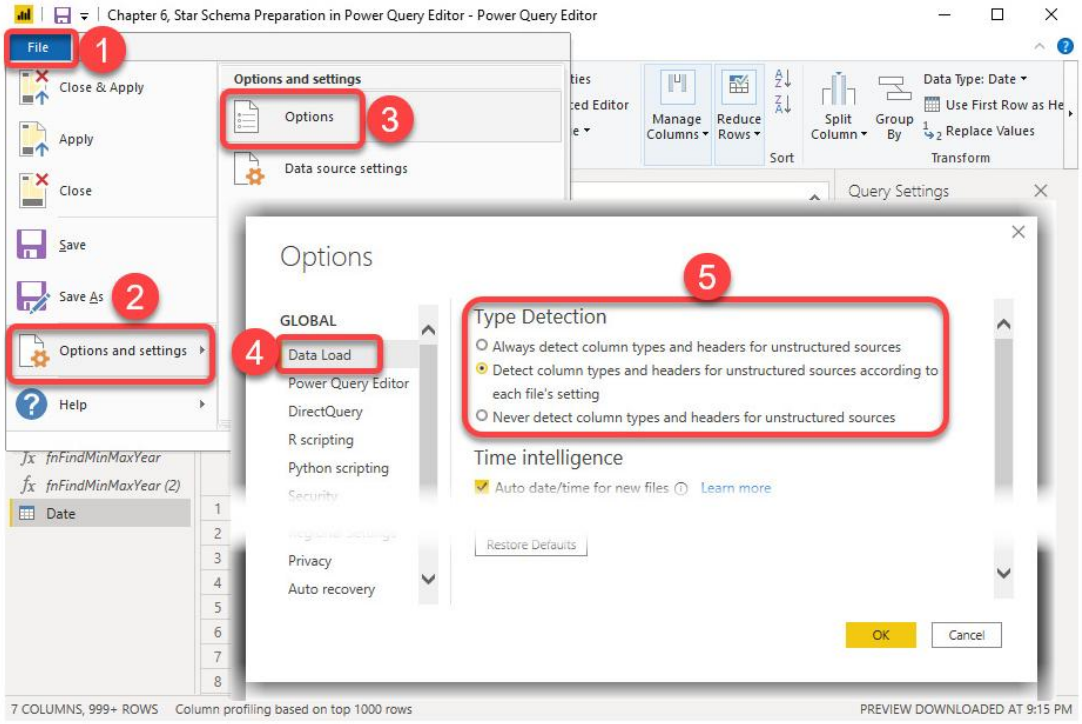

Figure 6.8 – Changing the Type Detection configuration at the Global level

6. Click **Data Load** under the **Current File** section.
7. Tick/untick the **Detect column types and headers for unstructured sources** option:

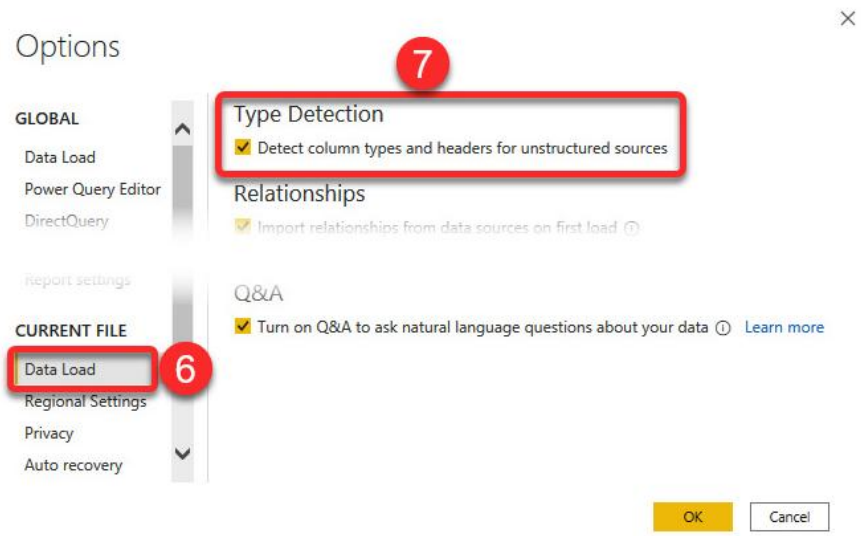

Figure 6.9 – Changing the Type Detection configuration at the Current File level

In this section, we identified potential dimensions and facts. In the next section, we will look at how to create physical dimensions from the potential ones.

Creating Dimensions tables

We should already be connected to the **`Chapter 6, Sales Data.xlsx`** file from Power Query Editor. We need to analyze each dimension from a business perspective and create dimensions, if they need to be created.

Geography

Looking at the identified business requirements shows that we have to have a dimension that keeps geographical data. When we look at the data, we can see that there are geography-related columns in the **Sales** table. We can create a separate dimension for **Geography** that's derived from the **Sales** table. However, this might not cover all business requirements.

Let's have another look at the **Potential Dimensions** table, shown in the following figure, which shows some geography-related columns in the **Customer** table. We need to find commonalities in the data to combine the data from both tables into a single **Geography** dimension. Using **Column Distribution** shows that the **CustomerKey** column is a **primary key** of the **Customer** table:

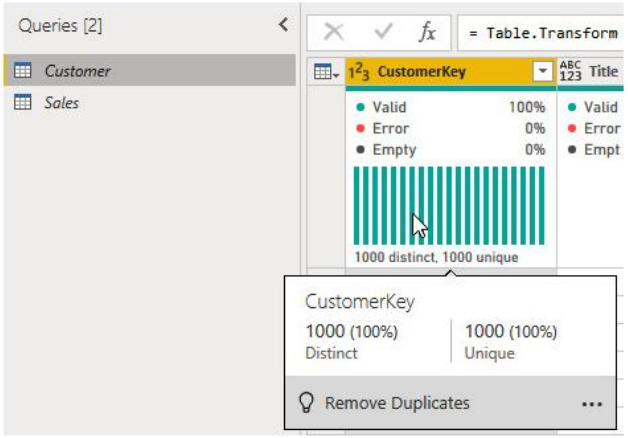

Figure 6.10 – Column Distribution shows that CustomerKey is the primary key for the Customer table

Enabling and using **Column Distribution** was explained in *Chapter 3*, *Data Preparation in Power Query Editor* (*Figure 29* and *Figure 30*).

Let's look at **Column Distribution** for the **SalesContinent**, **SalesCountry**, and **SalesRegion** columns from the **Customer** table and **SalesTerritoryGroup**, **SalesTerritoryCountry**, and **SalesTerritoryRegion** from the **Sales** table. It is clear that the number of distinct values in each column from the **Customers** tables matches the number of distinct values in the corresponding column from the **Sales** table. This is shown in the following screenshot:

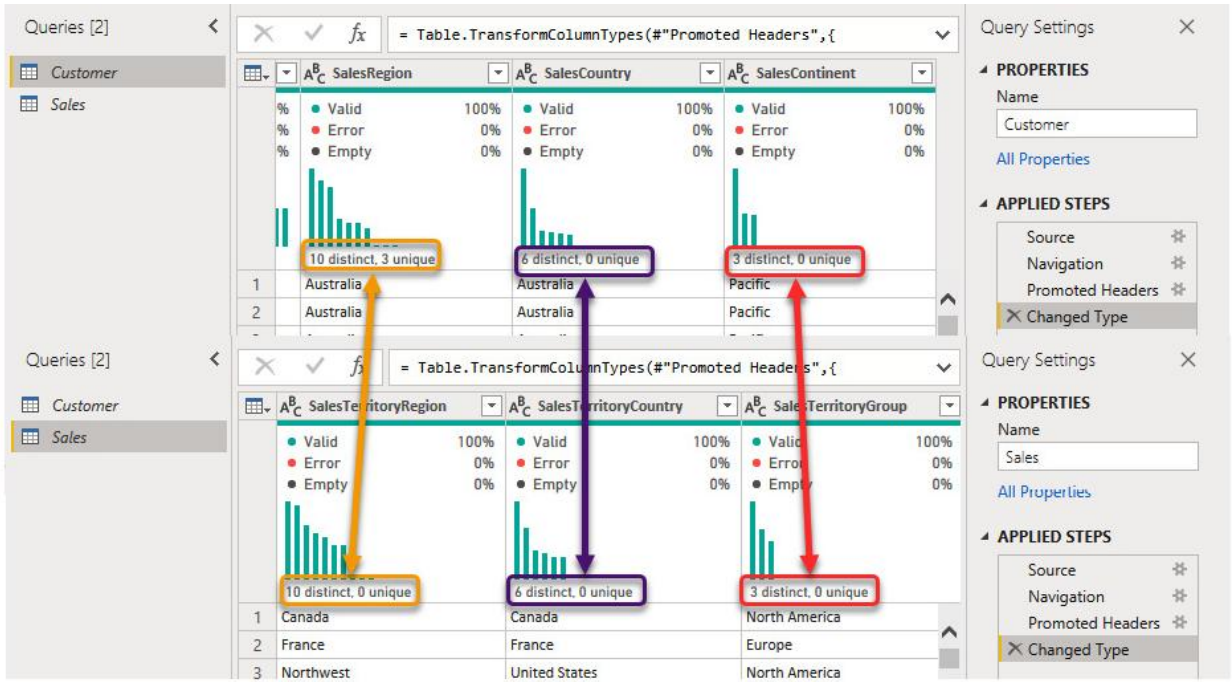

Figure 6.11 – Comparing Column Distribution for geography-related columns from the Customer table and the Sales table

To ensure the values of the **SalesContinent**, **SalesCountry**, and **SalesRegion** columns from the **Customer** table and the **SalesTerritoryGroup**, **SalesTerritoryCountry**, and **SalesTerritoryRegion** columns from the **Sales** table match, let's go through the following test process:

1. **Reference** the **Customer** table.
2. Rename the table **CustomerGeoTest**.
3. **Unload** the table.
4. Keep the **SalesContinent**, **SalesCountry**, and **SalesRegion** columns by **Removing Other Columns**.
5. Click the table icon at the top left of the **Data View** pane and click **Remove Duplicates**. The following image shows the preceding steps:

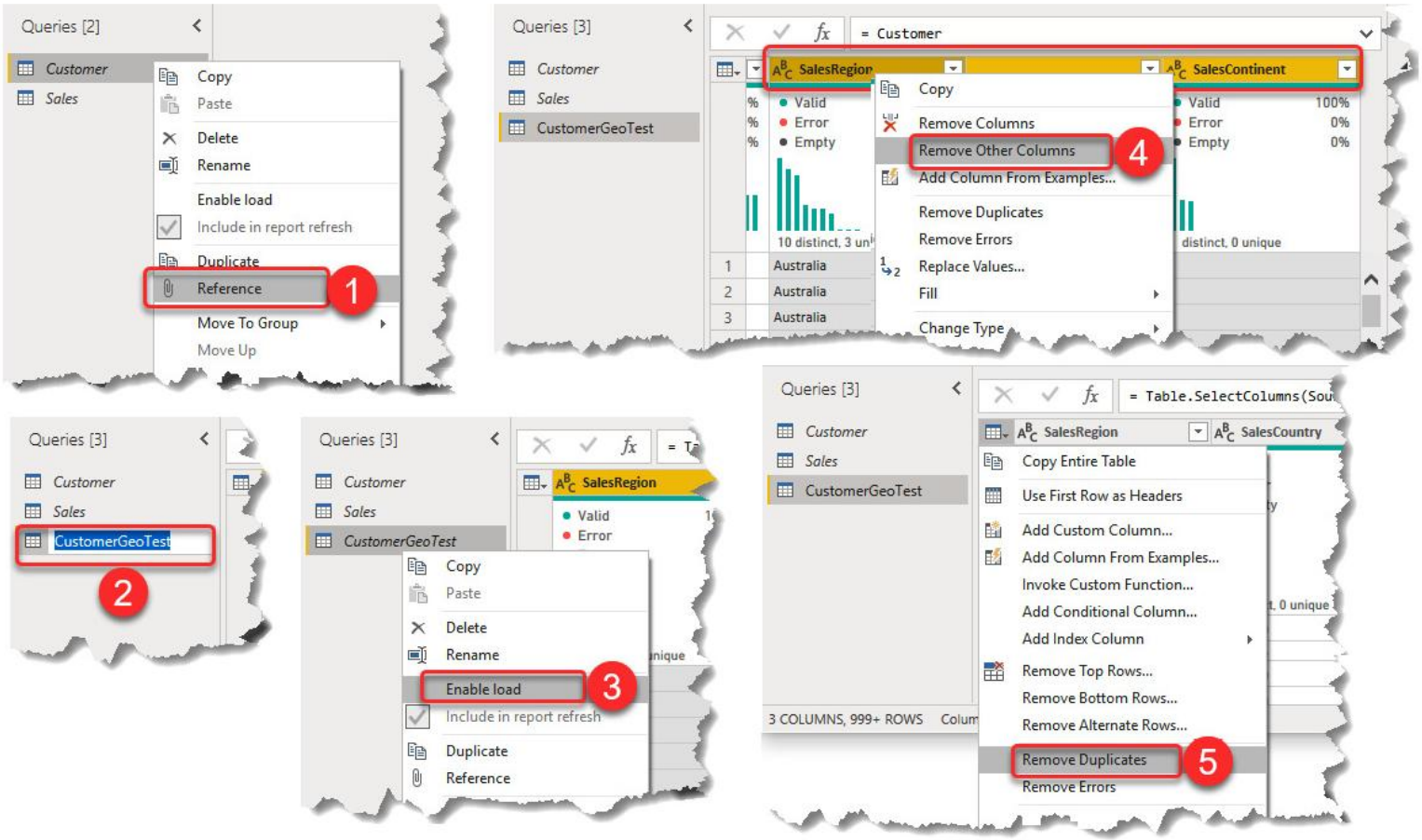

Figure 6.12 – Steps required to reference the Customer table and remove duplicates

The results of the preceding steps are as follows:

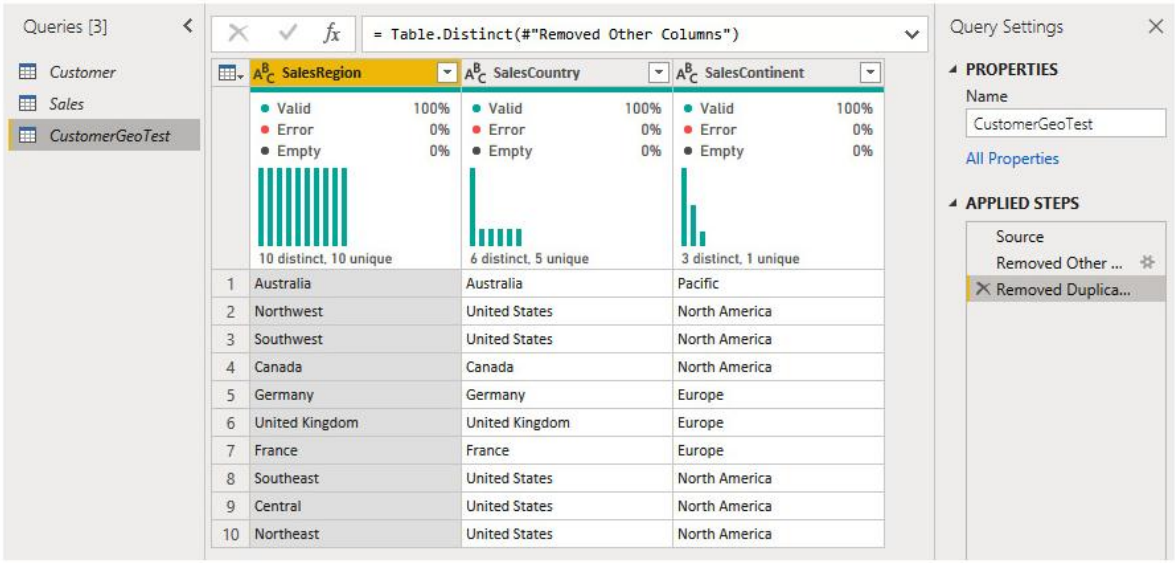

Figure 6.13 – The results of referencing the Customer table and removing duplicates from the geography-related columns

We must then go through the same process to remove the duplicates in the **`SalesTerritoryGroup`**, **`SalesTerritoryCountry`**, and **`SalesTerritoryRegion`** columns from the **`Sales`** table. The following image shows the latter results next to the results of removing the duplicates of the **`SalesContinent`**, **`SalesCountry`**, and **`SalesRegion`** columns from the **`Customer`** table:

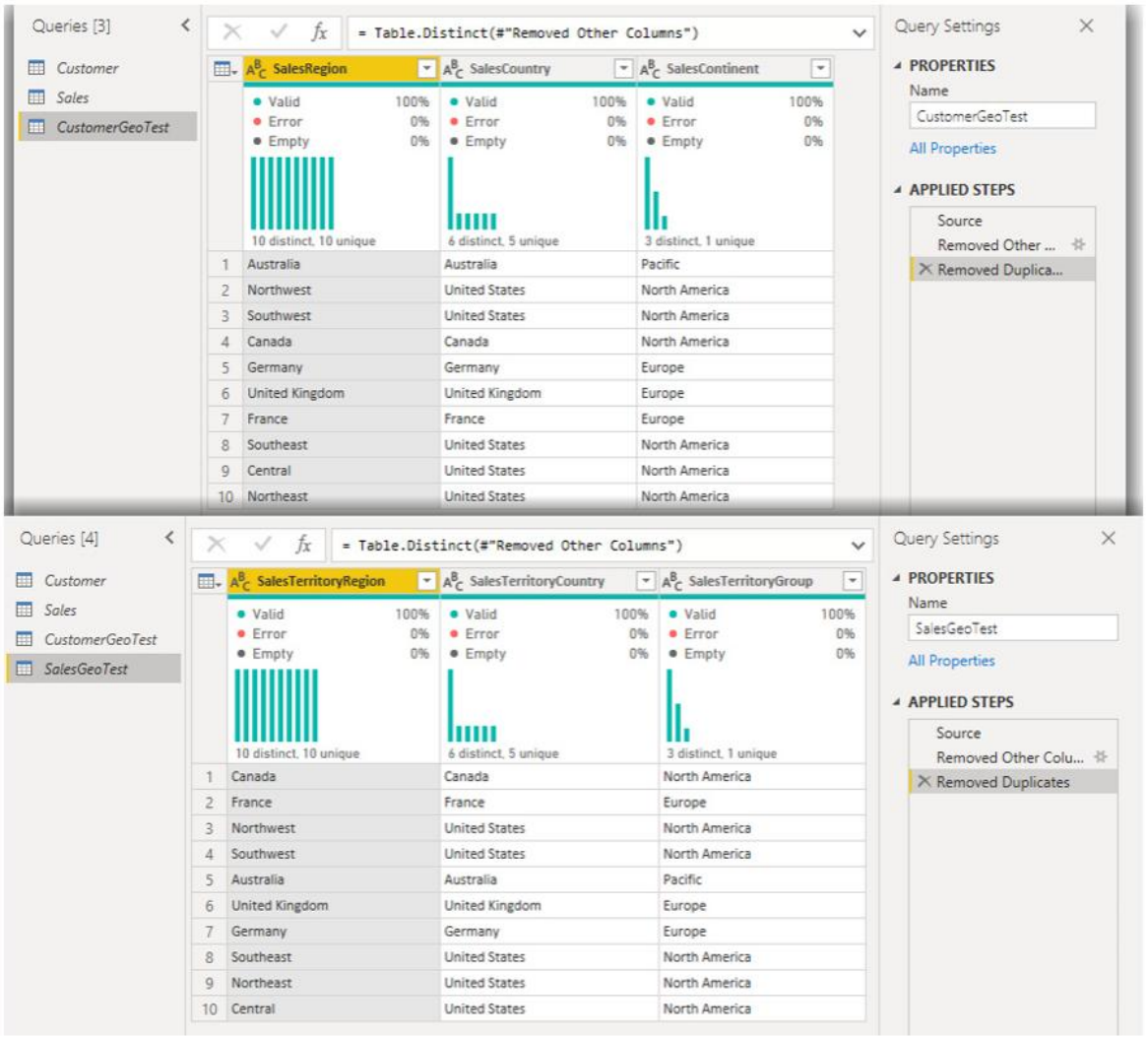

Figure 6.14 – Comparing the results of the CustomerGeoTest table and the SalesGeoTest table

As the preceding image shows, the only difference between the two is the columns' sorting order, which is not essential. As a result of the preceding exercise, we do not need to create a **Geography** dimension as the **SalesTerritoryGroup**, **SalesTerritoryCountry**, and **SalesTerritoryRegion** columns from the **Sales** table are redundant compared to the **SalesContinent**, **SalesCountry**, and **SalesRegion** columns from the **Customer** table, while the geography-related columns in the **Customer** table provide a higher level of detail.

Sales order

There are only two columns in the **Sales** table – **SalesOrderNumber** and **SalesOrderLineNumber** – that contain descriptive data for sales orders. Let's ask a question. Why do we need to create a dimension for this? Let's look at the data:

1. Change **Column profiling** to **based on the entire data set**.
2. Looking at **Column Distribution** for both columns shows that **SalesOrderNumber** has **27,659** distinct values and that **SalesOrderLineNumber** has only **8** distinct values:

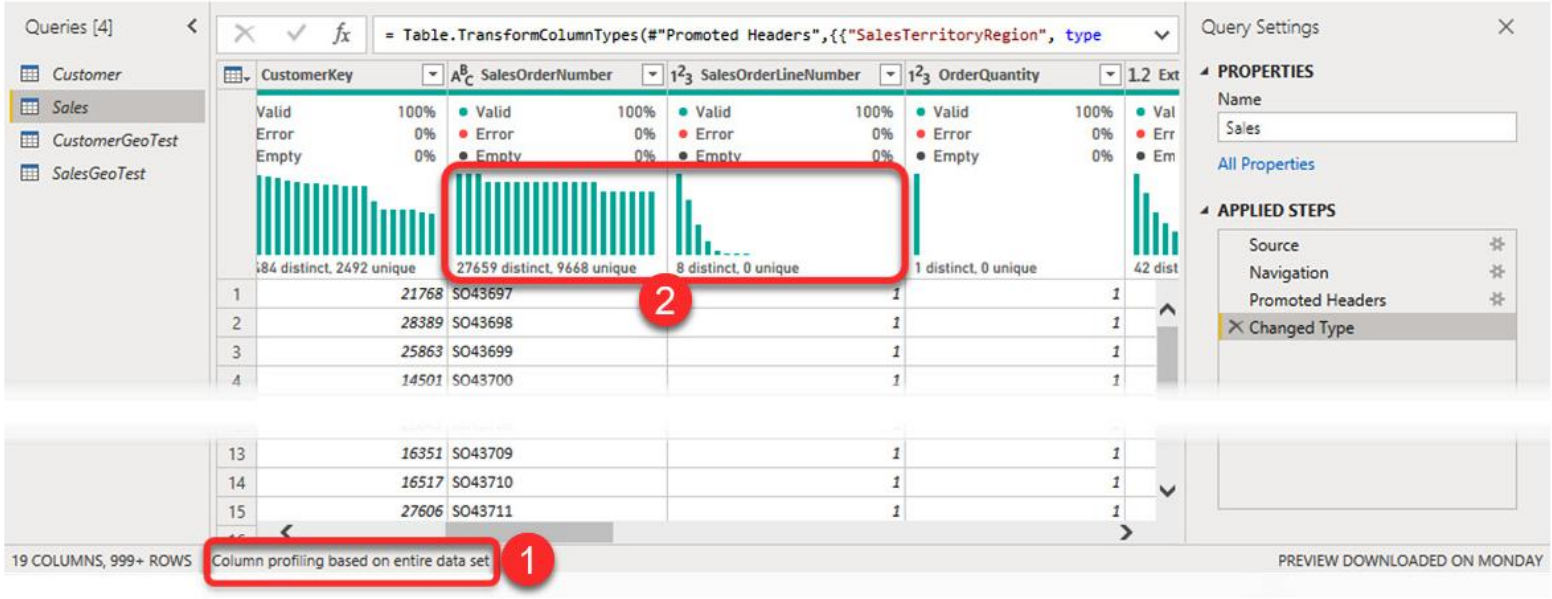

Figure 6.15 – Looking at the Column Distribution data shows that there are a lot of distinct values in SalesOrderNumber

Having many distinct values in **SalesOrderNumber** by itself decreases the chance of creating a new **Sales Order** dimension being a good idea. So, let's see if we can find more evidence to avoid creating the **Sales Order** dimension. If we merge the two columns, we will get a better idea of the number of rows we get in the new dimension if we happen to create one.

3. Select both the **SalesOrderNumber** and **SalesOrderLineNumber** columns.
4. Right-click on the title of a column and click **Merge Columns**.

5. From the **Merge Columns** window, stick to the defaults and click **OK**:

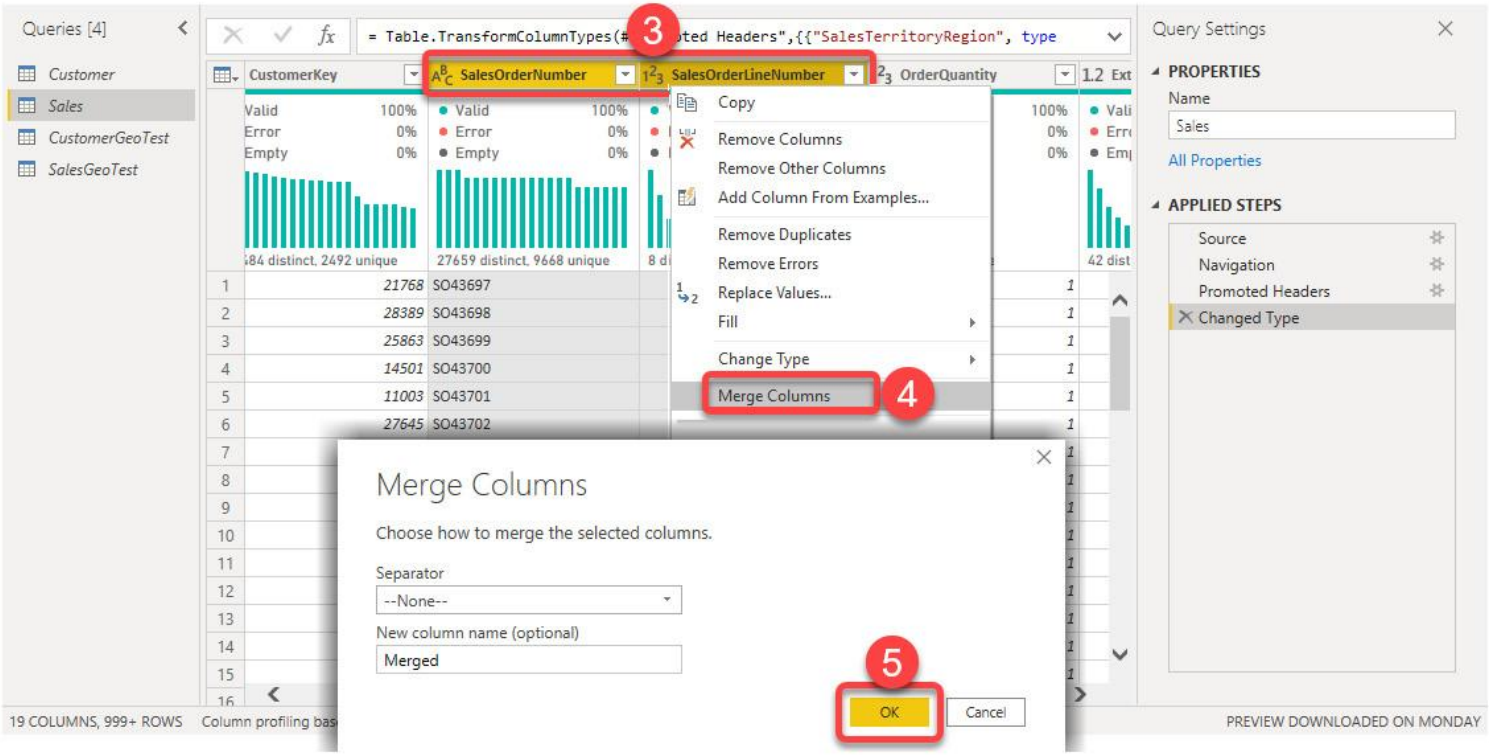

Figure 6.16 – Merging the SalesOrderNumber and SalesOrderLineNumber columns

The new **Merged** column reveals that it contains **60398 distinct and 60398 unique values**, which means the combination of **`SalesOrderNumber`** and **`SalesOrderLineNumber`** is the **`primary key`** value of the **`Sales`** table, as shown in the following screenshot. Therefore, even if we create a separate dimension, we will get the same number of rows as our fact table. Moreover, we cannot imagine any linkages to any other dimensions. Therefore, it is best to keep those two columns in the **`Sales`** table. These types of dimensions cannot be moved out of the fact table because of their data characteristics. They also do not have any other attributes or a meaningful linkage to any other dimensions. These type of dimensions are called **Degenerate Dimensions**:

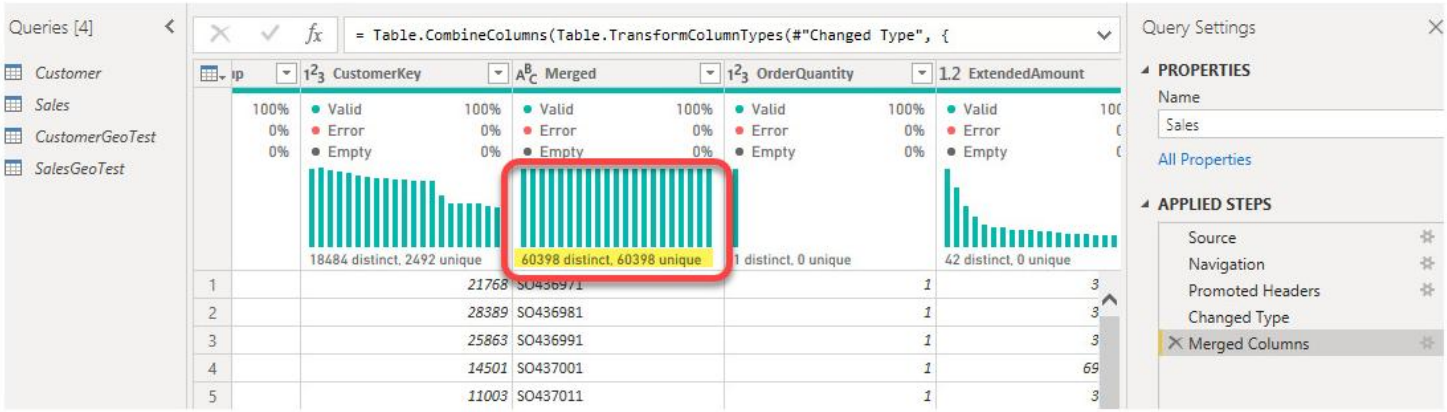

Figure 6.17 – Merged Column results

Now, we must remove the **Merged Column** steps we just created from our **Applied Steps**.

Product

The **Product** dimension is by far the most obvious one that has three descriptive columns. We can derive the **Product** dimension from the **Sales** table by referencing the **Sales** table. Then, we can remove other columns to keep the **Product**, **ProductSubcategory**, and **ProductCategory** columns. As the next step, we must remove the duplicates from the **Product** table. Moreover, we need to generate a **ProductKey** column as the primary key of the **Product** table. Next, we need to merge the **Sales** table with the **Product** table to get a **ProductKey**. We can also rename the columns to more user-friendly versions. We will rename **ProductSubcategory** to **Product Subcategory** and **ProductCategory** to **Product Category**.

NOTE

Moving forward, we will reference the Sales table many times. Therefore, we'll rename the Sales table Sales Base.

The following steps show how to implement the preceding process in Power Query Editor:

1. Rename the **Sales** table **Sales Base**:

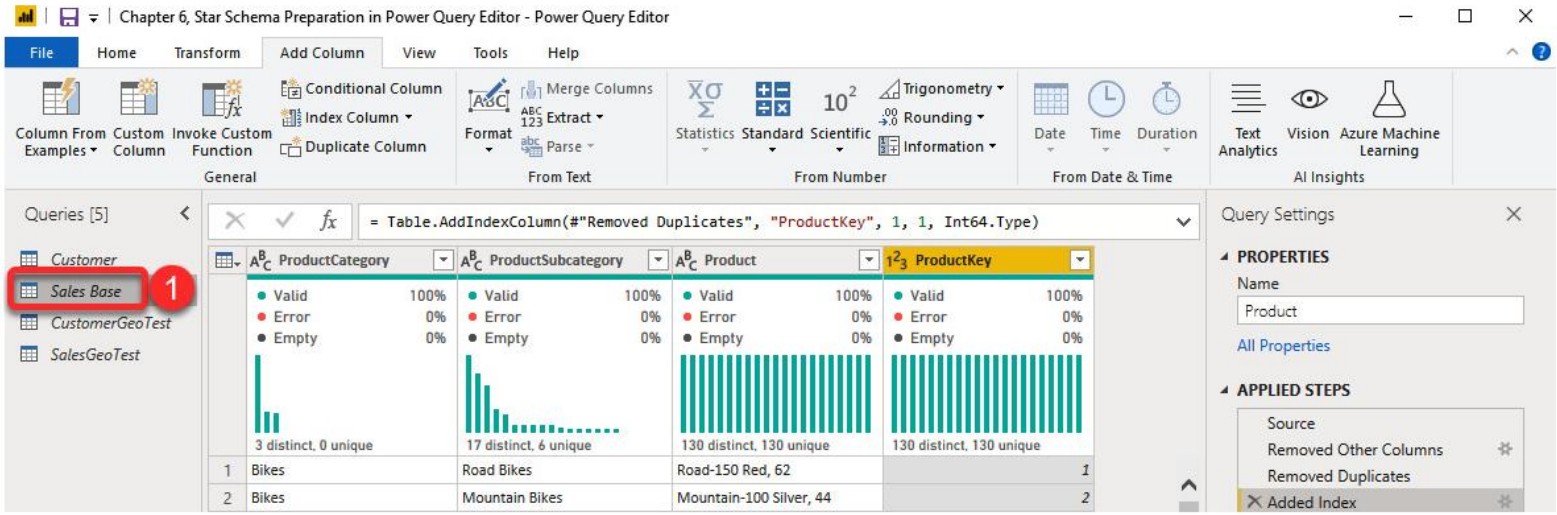

Figure 6.18 – Renaming the Sales table Sales Base

2. **Reference** the **`Sales Base`** table:

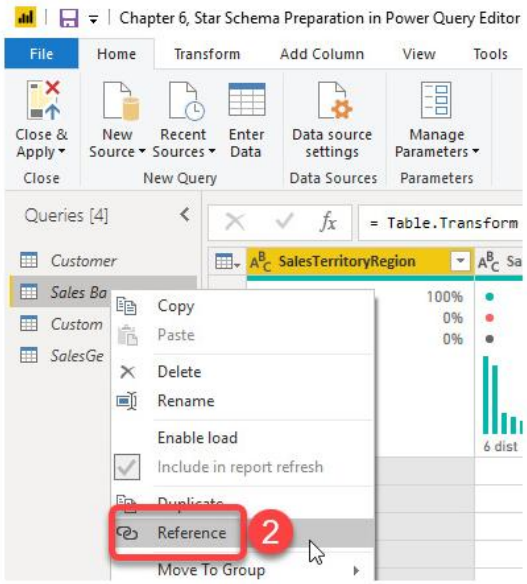

Figure 6.19 – Referencing the Sales Base table

3. **Rename** the referencing table **`Product`**.
4. Select the **`ProductCategory`**, **`ProductSubcategory`**, and **`Product`** columns, respectively.

   NOTE

   Power Query will order the columns by order of our selection in Step 4.

5. Right-click one of the selected columns and click **Remove Other Columns**:

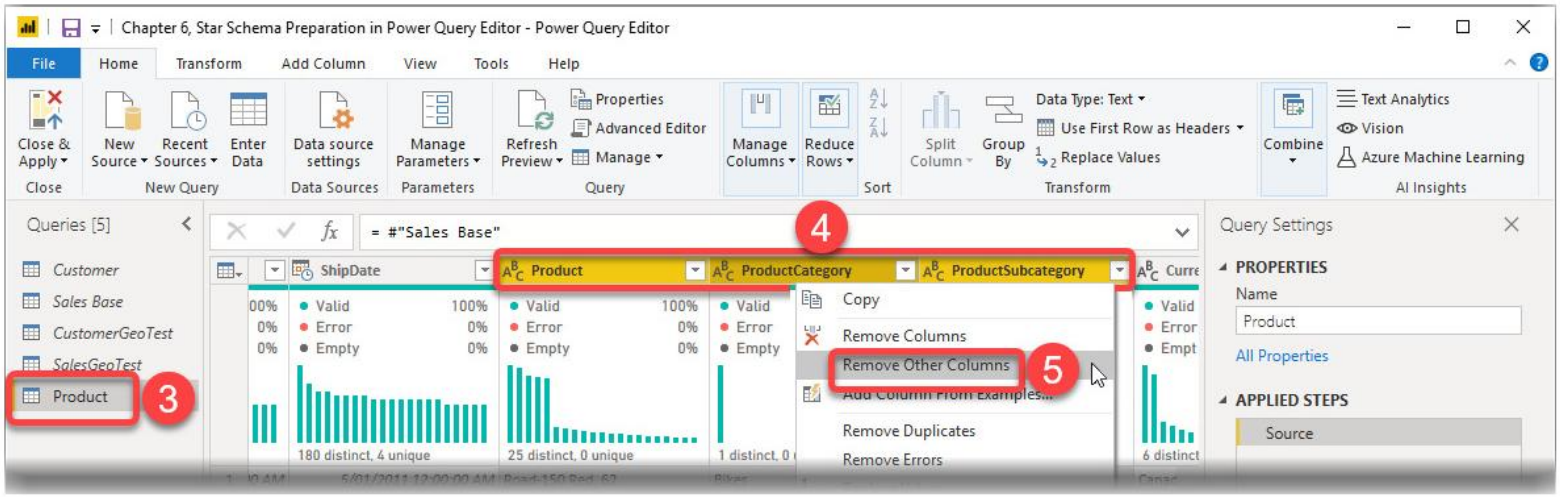

Figure 6.20 – Remove Other Columns option

6. Click the table button at the top left of the **Data view** pane.

7. Click **Remove Duplicates**:

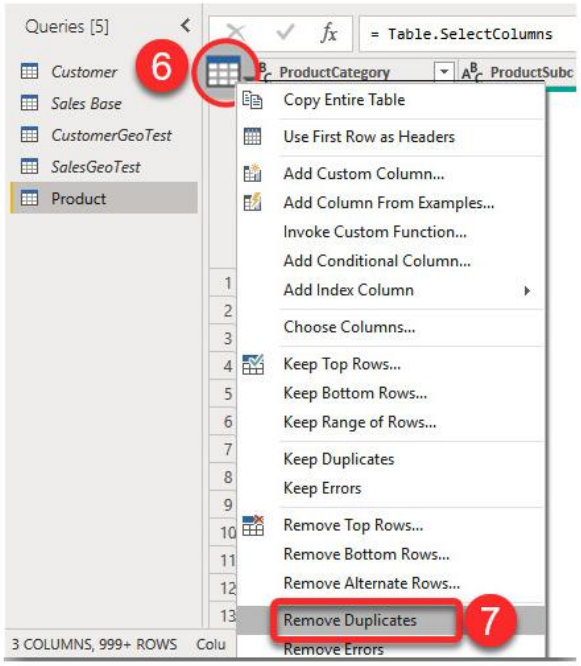

Figure 6.21 – Removing duplicates from all rows

So far, we've got distinct values for each row of data in the **Product** table. Now, we need to create a unique identifier for each row:

8. Click the **Add Column** tab from the ribbon.
9. Click the **Index Column** dropdown button.
10. Click **From 1**:

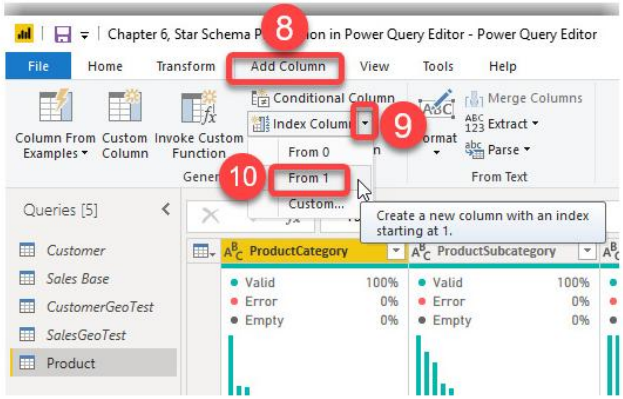

Figure 6.22 – Adding an Index column with an index starting from 1

11. So far, we've created an index column with a default name of **Index**. We need to rename this column **ProductKey**. We can edit the Power Query expression we generated in the **Added Index** step rather than renaming it as a new **Rename Column** step.
12. Click the **Added Index** step from **Applied Steps**. Then, from the formula bar, change **Index** to **ProductKey**, as shown in the following screenshot:

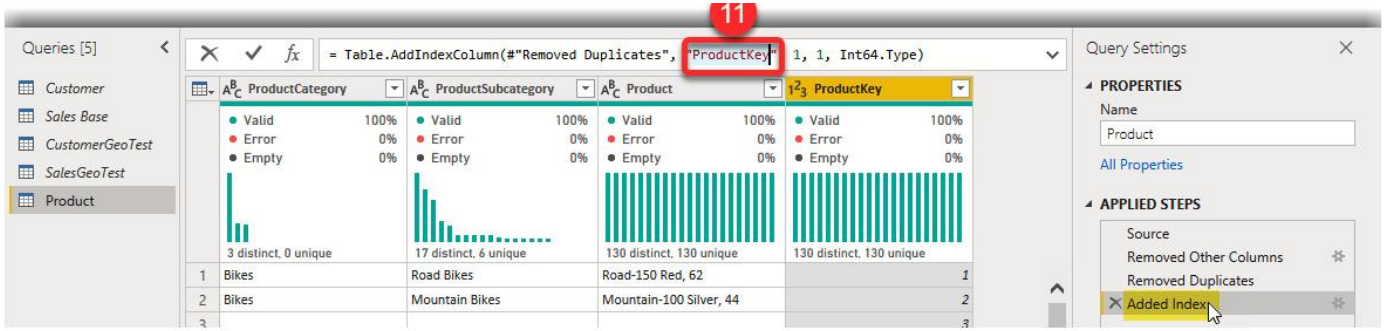

Figure 6.23 – Changing the default index column's name

With that, we've created the **Product** dimension.

Currency

The **Currency** column in the **Sales Base** table holds the currency description for each transaction. So, by definition, it is a dimension. Let's raise the Why question here. Why do we need to create a separate table for **Currency**? To answer this question, let's analyze the situation in more detail. As shown in the following screenshot, the column distribution box, when set to work based on the entire dataset, shows that the **Currency** column's cardinality is low, with only 6 distinct values:

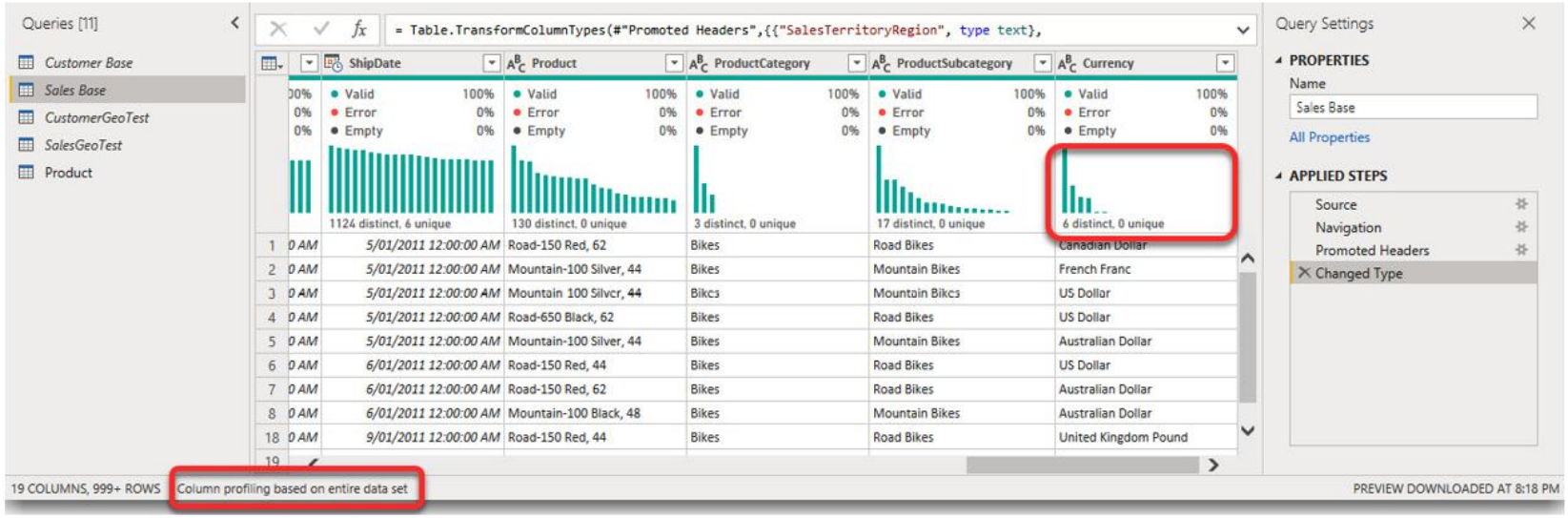

Figure 6.24 – Column distribution for the Currency column over the entire dataset

Since the **Columnstore** indexing in the **xVelocity** engine provides better data compression and better performance over low cardinality columns, we can expect minimal or no sensible performance or storage gains by creating a new dimension table for **Currency**. This is the case in our scenario. We do not gain better performance or save a lot of storage or memory by creating a separate **Currency** dimension. Besides, we do have other attributes providing more descriptions for currencies. Last but not least, **Currency** does not have any meaningful linkages to any other dimensions. As a result, we can keep the **Currency** column as a **Degenerate Dimension** in the fact table.

Customer

We can derive the **Customer** table from the original **Customer** table from the source. To do this, we'll rename the original **Customer** table **Customer Base**.

Let's look at the **Sales Base** table to see how each row is related to the **Customer Base** table. As shown in the following screenshot, the **Sales Base** table has a **CustomerKey** column. The **Column Quality Box** of **CustomerKey** in the **Sales Base** table reveals that there is a customer key for every single sales transaction in the **Sales Base** table (**0% Empty**). Therefore, every row of the **Customer Base** table describes sales transactions from the customer's viewpoint:

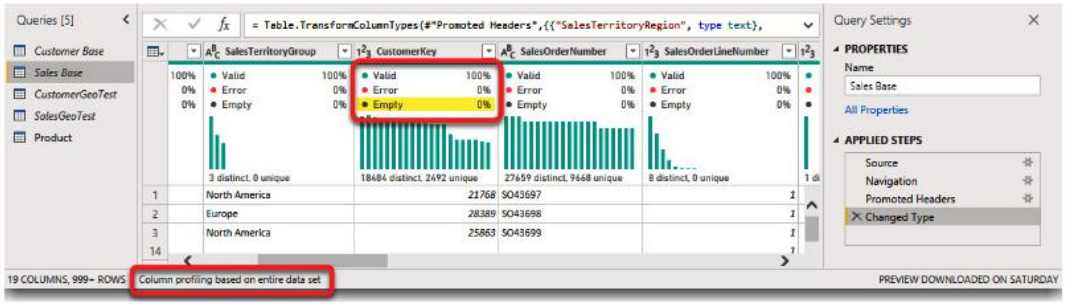

Figure 6.25 – The Column Quality information shows 0% Empty for CustomerKey

Each row keeps descriptive information about a customer. Therefore, having a **Customer** dimension is inevitable. So, let's create the **Customer** table by following these steps:

1. **Reference** the **Customer Base** table:

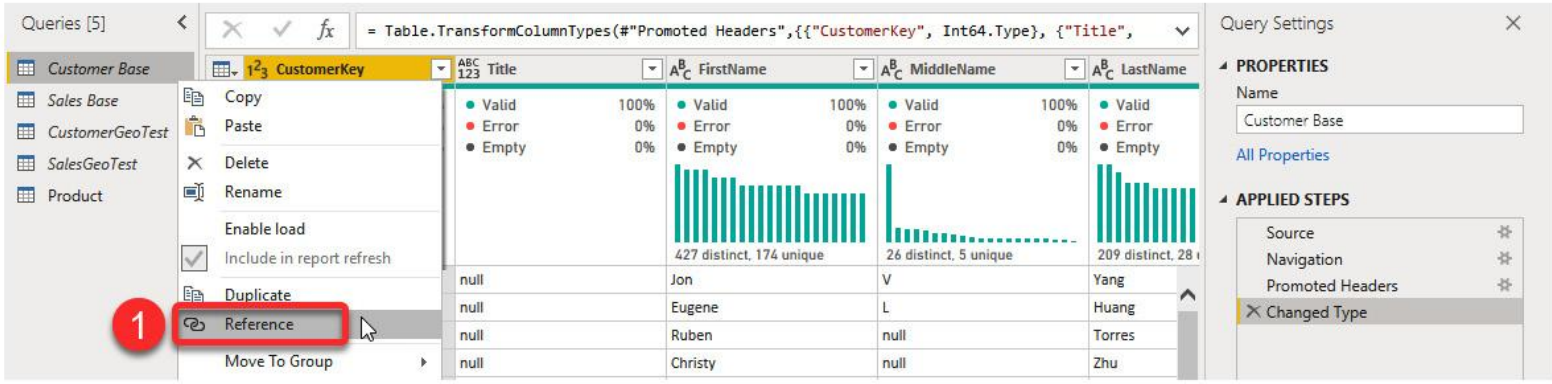

Figure 6.26 – Referencing the Customer Base table

2. **Rename** the referencing table **`Customer`**.

   We need to keep the following columns by removing the other columns; that is, **`CustomerKey`**, **`Title`**, **`FirstName`**, **`MiddleName`**, **`LastName`**, **`NameStyle`**, **`BirthDate`**, **`MaritalStatus`**, **`Suffix`**, **`Gender`**, **`EmailAddress`**, **`YearlyIncome`**, **`TotalChildren`**, **`NumberChildrenAtHome`**, **`Education`**, **`Occupation`**, **`HouseOwnerFlag`**, **`NumberCarsOwned`**, **`AddressLine1`**, **`AddressLine2`**, **`Phone`**, **`DateFirstPurchase`**, and **`CommuteDistance`**.

3. The simplest way to do so is to click **Choose Columns** from the **Home** tab.

4. Keep the preceding columns and deselect the rest.

5. Click **OK**:

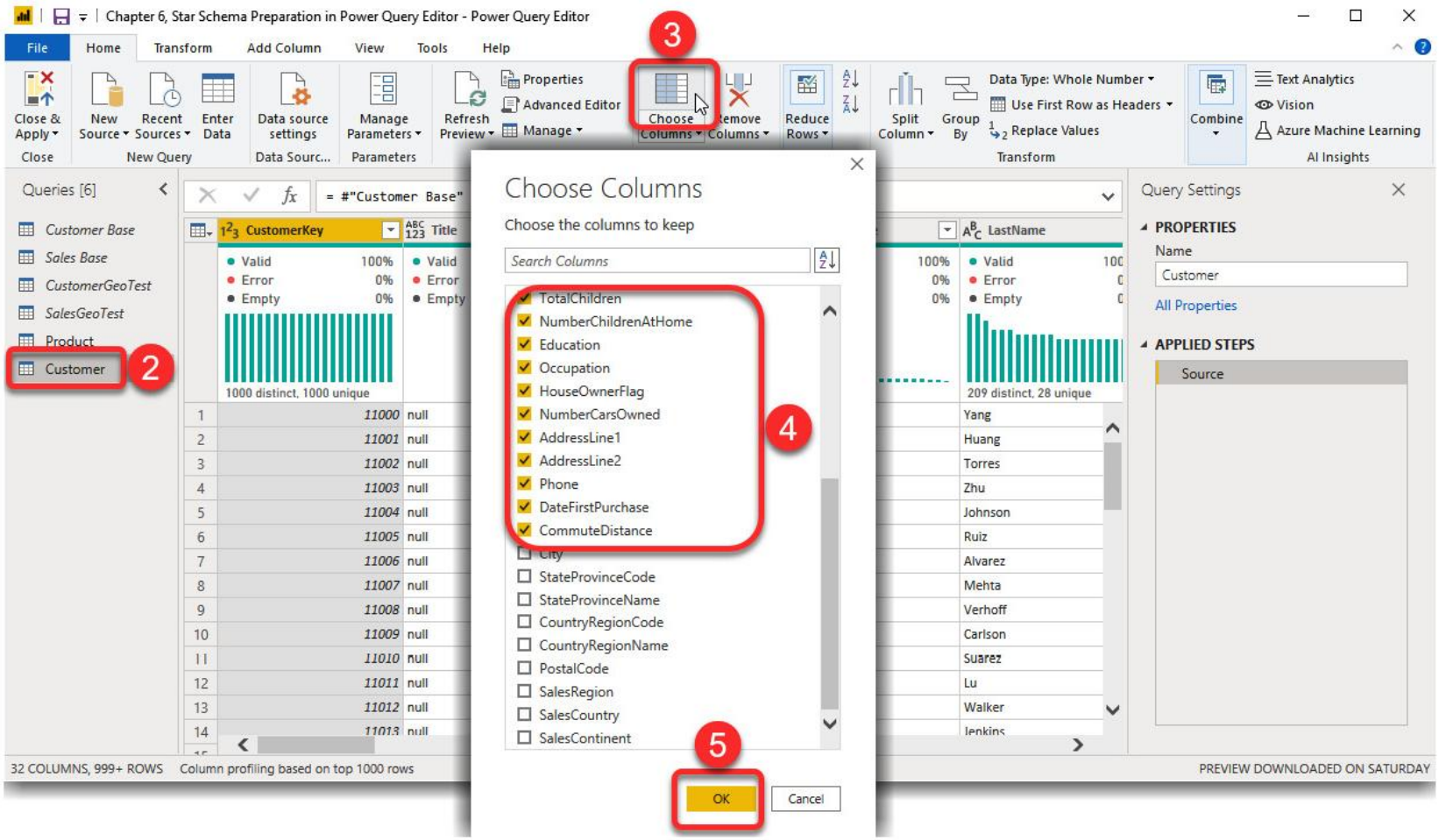

Figure 6.27 – Removing the unnecessary columns

With that, we've created the **`Customer`** dimension. Since the **`Customer`** dimension already has a **`CustomerKey`**, we do not need to take any more actions.

As you can see, we removed all geography-related columns from the **`Customer`** dimension. We will create a separate dimension for them next.

Sales Demographic

We previously looked at creating a **Geography** dimension, which revealed that the geography columns in the **Customer Base** table could give us more details, which will help us create more accurate analytical reports with lower granularity. Now, let's create a new dimension to keep **Sales Demographic** descriptions that are derived from **Customer Base**, as follows:

1. **Reference** the **Customer Base** table from Power Query Editor.
2. **Rename** the new table **Sales Demographic**.
3. Click the **Choose Columns** button from the **Home** tab.
4. Untick all the columns other than **City**, **StateProvinceCode**, **StateProvinceName**, **CountryRegionCode**, **CountryRegionName**, **PostalCode**, **SalesRegion**, **SalesCountry**, and **SalesContinent**.
5. Click **OK**:

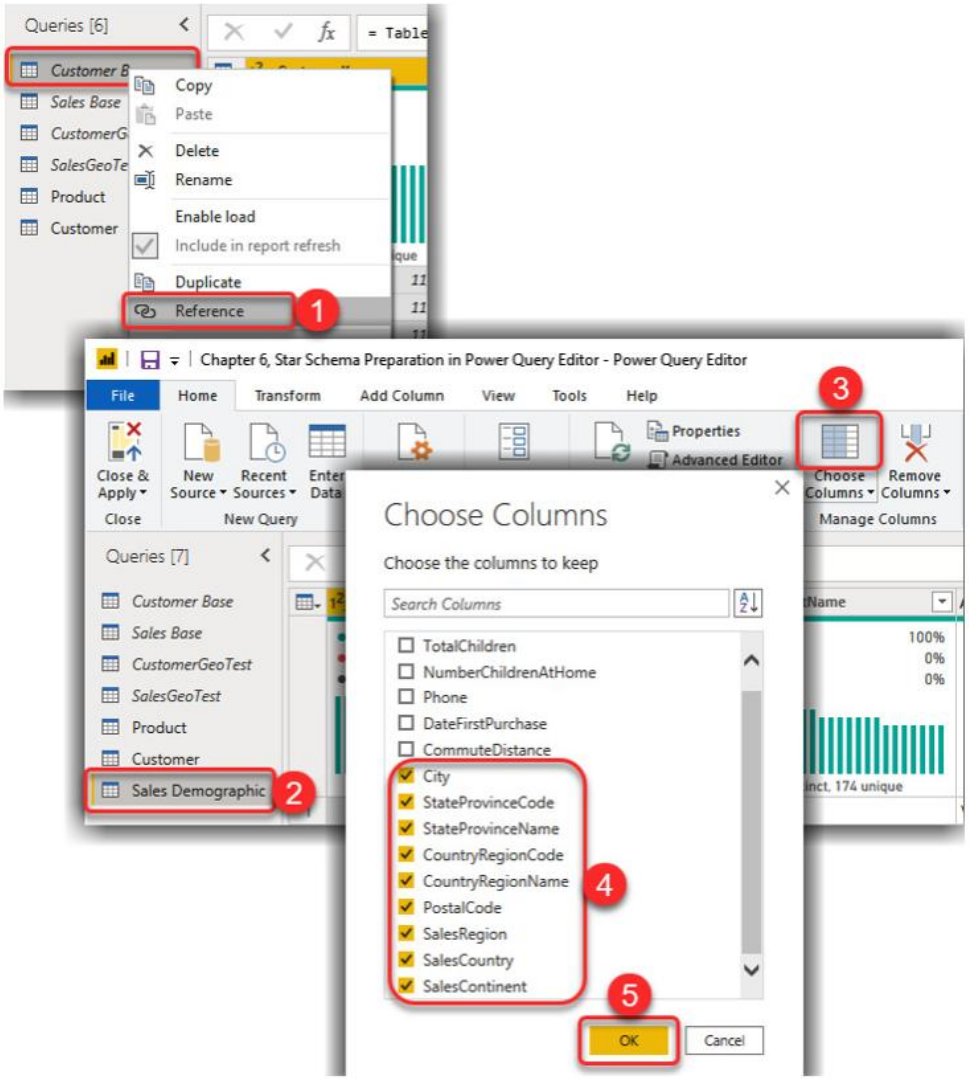

Figure 6.28 – Referencing the Customer Base table to create a Sales Demographic table and keep the relevant columns

Looking at the data in the **CountryRegionName** and **SalesCountry** columns shows that the two columns contain the same data. Therefore, we need to remove one of them by double-clicking the **Remove Other Columns** step and unticking the **CountryRegionName** column.

The next step is to remove the duplicate rows. This guarantees that the dimension does not contain duplicate rows in the future, even if there are currently no duplicate rows.

6. Click the table transformation button at the top left of the **Data view** pane.
7. Click **Remove Duplicates**:

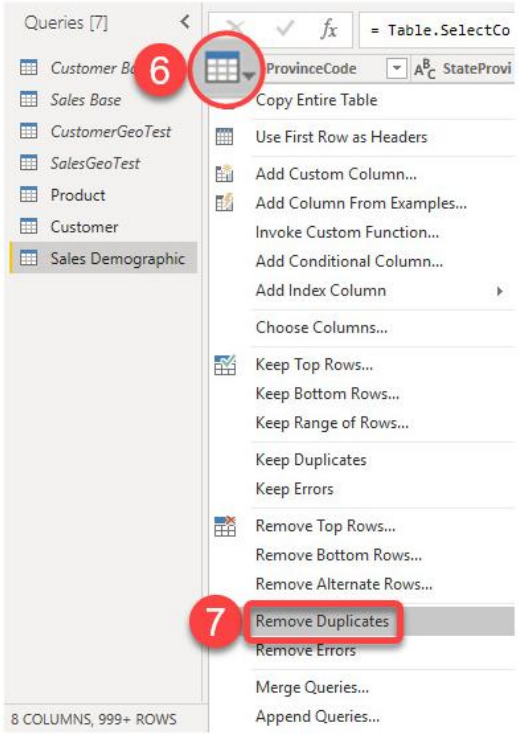

Figure 6.29 – Removing duplicates from the Sales Demographic table

Now, we need to add an **Index** column to create a **primary key** for the **Sales Demographic** dimension.

8. Click the **Index Column** dropdown button from the **Add Column** tab.
9. Click **From 1**:

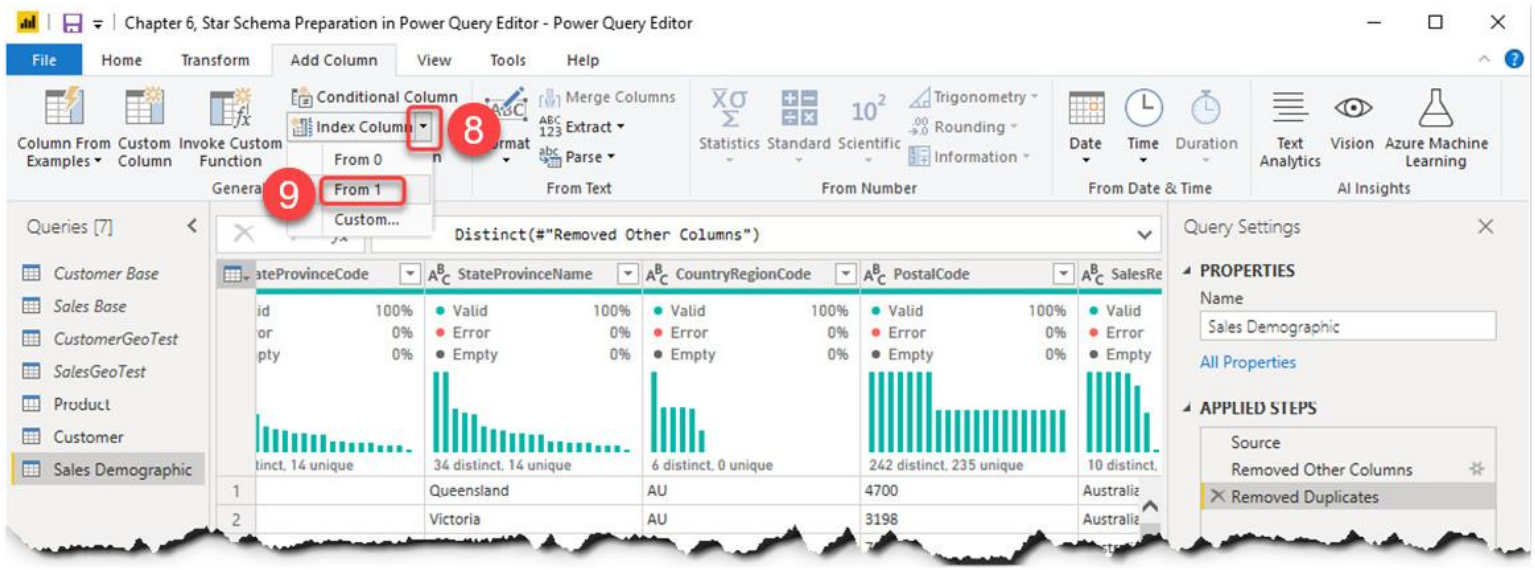

Figure 6.30 – Adding an Index column to the Sales Demographic table

10. Replace **`Index`** with **`SalesDemographicKey`** from formula bar.

11. Click the **Submit** button ✓:

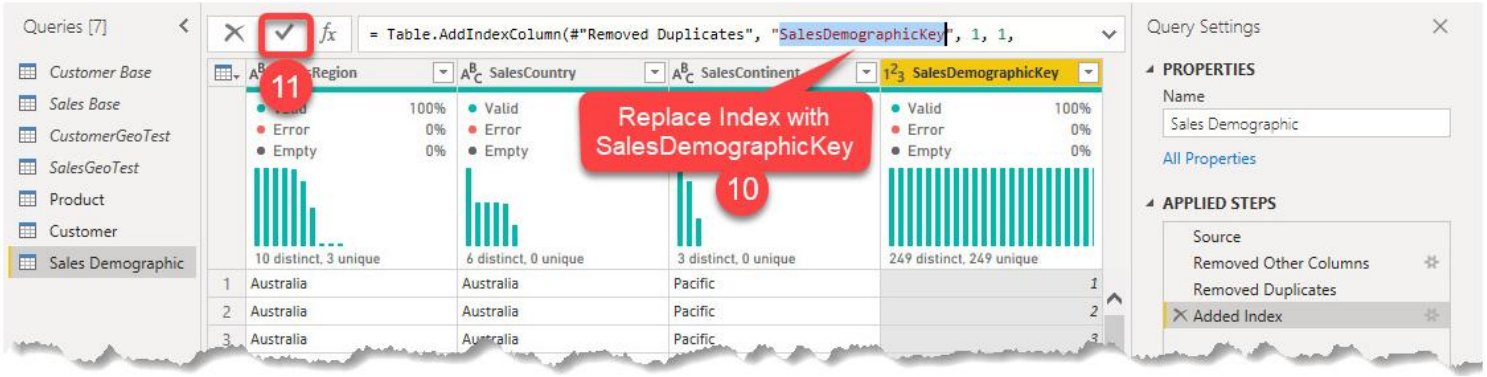

Figure 6.31 – Replacing Index with SalesDemographicKey

So far, we've created all the potential dimensions that can be derived from the **`Sales Base`** and **`Customer Base`** tables. As we discussed earlier in this chapter, we also need to create a **`Date`** dimension and a **`Time`** dimension. We will do so in the following sections.

Date

As a result of our requirement gathering session with the business, we found that we need to have a **Date** dimension and a **Time** dimension as the business needs us to analyze the **Sales** data over date and time elements. As we discussed in *Chapter 2*, *Data Analysis eXpressions and Data Modeling*, the **Date** dimension can be created using DAX. We also discussed the advantages and disadvantages of using the **CALENDARAUTO()** function in DAX. In this section, we'll create a custom function in Power Query Editor to generate a simple **Date** dimension. This custom function will accept two input parameters: **Start Year** and **End Year**. Then, it will generate a **Date** table starting from **1st Jan** of **Start Year** and ending on **31st Dec** of **End Year**.

The generated dates in the **Date** column are continuous and don't have any gaps in-between dates. The following steps show how to use the following expression to create and invoke the custom function:

1. The custom function can be created by copying the following Power Query expression:

```
// fnGenerateDate
(#''Start Year'' as number, #''End Year'' as number) =>
    let
        GenerateDates = List.Dates(#date(#''Start Year'',1,1), Duration.Days(Duration.From(#date(#''End Year'', 12, 31) - #date(#''Start
        Year'' - 1,12,31))), #duration(1,0,0,0) ),
        #''Converted to Table'' = Table.TransformColumnTypes(Table.FromList(GenerateDates, Splitter.SplitByNothing(), {''Date''}),
        {''Date'', Date.Type}),
        #''Added Custom'' = Table.AddColumn(#''Converted to Table'', ''DateKey'', each Int64.From(Text.Combine({Date.ToText([Date],
        ''yyyy''), Date.ToText([Date], ''MM''), Date.ToText([Date], ''dd'')})), Int64.Type),
        #''Year Column Added'' = Table.AddColumn(#''Added Custom'', ''Year'', each Date.Year([Date]), Int64.Type),
        #''Quarter Column Added'' = Table.AddColumn(#''Year Column Added'', ''Quarter'', each ''Qtr
        ''&Text.From(Date.QuarterOfYear([Date])) , Text.Type),
        #''MonthOrder Column Added'' = Table.AddColumn(#''Quarter Column Added'', ''MonthOrder'', each Date.ToText([Date], ''MM''),
        Text.Type),
        #''Short Month Column Added'' = Table.AddColumn(#''MonthOrder Column Added'', ''Month Short'', each Date.ToText([Date], ''MMM''),
        Text.Type),
        #''Month Column Added'' = Table.AddColumn(#''Short Month Column Added'', ''Month'', each Date.MonthName([Date]), Text.Type)
    in
        #''Month Column Added''
```

NOTE

*The preceding code is also available in this book's GitHub repository, in the **Chapter 6, Generate Date Dimension.m** file, via the following URL: https://github.com/PacktPublishing/Expert-Data-Modeling-with-Power-BI/blob/master/Chapter%206%2C%20Generate%20Date%20Dimension.m.*

2. In **Power Query Editor**, click the **New Source** drop-down button.
3. Click **Blank Query**:

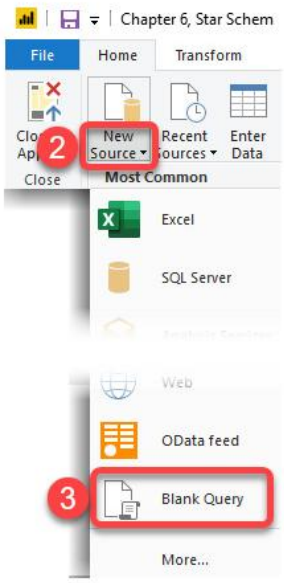

Figure 6.32 – Adding a Blank Query in Power Query Editor

4. Rename the new query from **`Query1`** to **`fnGenerateDate`**.
5. Click the **Advanced Editor** button from the **Home** tab.
6. Delete the existing code and paste the expressions we copied in the first step.
7. Click **Done**:

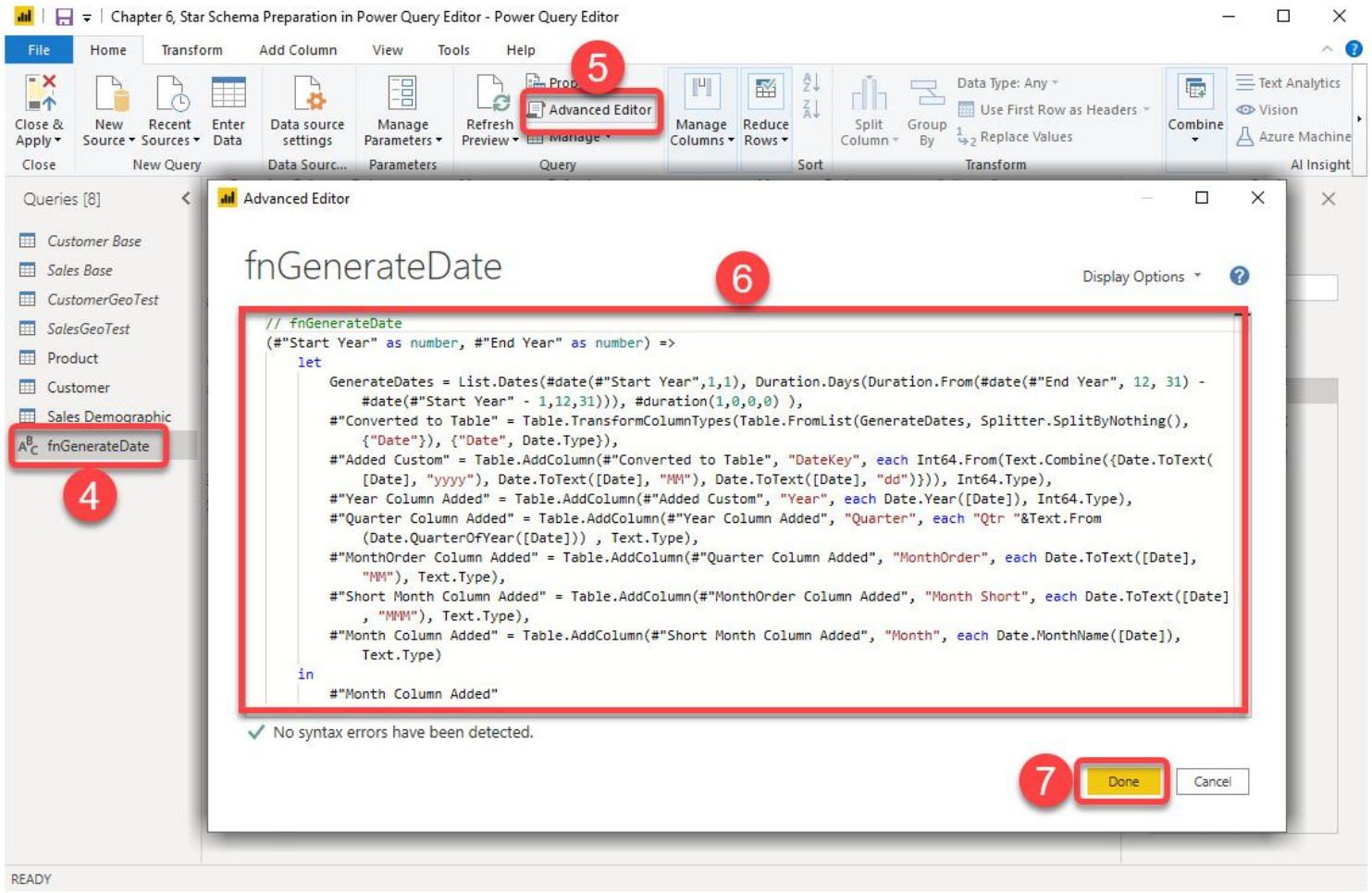

Figure 6.33 – Creating the fnGenerateDate custom function in Power Query Editor

The preceding process creates the **fnGenerateDate** custom function in Power Query Editor. The next step is to invoke the function by entering the **Start Year** and **End Year** parameters. In real-world scenarios, the date range of the **Date** dimension is dictated by the business. The business says what start date suits the business and what date in the future is the best fit for the business.

So, we can easily invoke the **fnGenerateDate** function by passing the **Start Date** and **the End Date** parameters. But in some other cases, we need to find the minimum and maximum dates of all the columns with **Date** or **DateTime** data types contributing to our data analysis. There are various ways to overcome those cases, such as the following:

- We can get the minimum and maximum dates by eyeballing the data if the dataset is small.
- We can sort each of the **OrderDate**, **DueDate**, and **ShipDate** values in ascending order to get the minimum dates, and then we can sort those columns in descending order to get the maximum dates.
- We can use the **List.Min()** function for each of the aforementioned columns to get the minimum dates. Then, using the **List.Max()** function for each column gives us the maximum dates.
- We can find the minimum and maximum dates using DAX.
- We can use the **Column profile** feature in Power Query Editor.

Regardless of the method we choose, once we've found our **Start Date** and **End Date**, which in our sample are **2010** and **2014**, respectively, we must invoke the **fnGenerateDate** function, as follows:

1. Select the **fnGenerateDate** custom function from the **Queries** pane.
2. Type in **2010** for the **Start Year** parameter and **2014** for the **End Year** parameter.
3. Click **Invoke**:

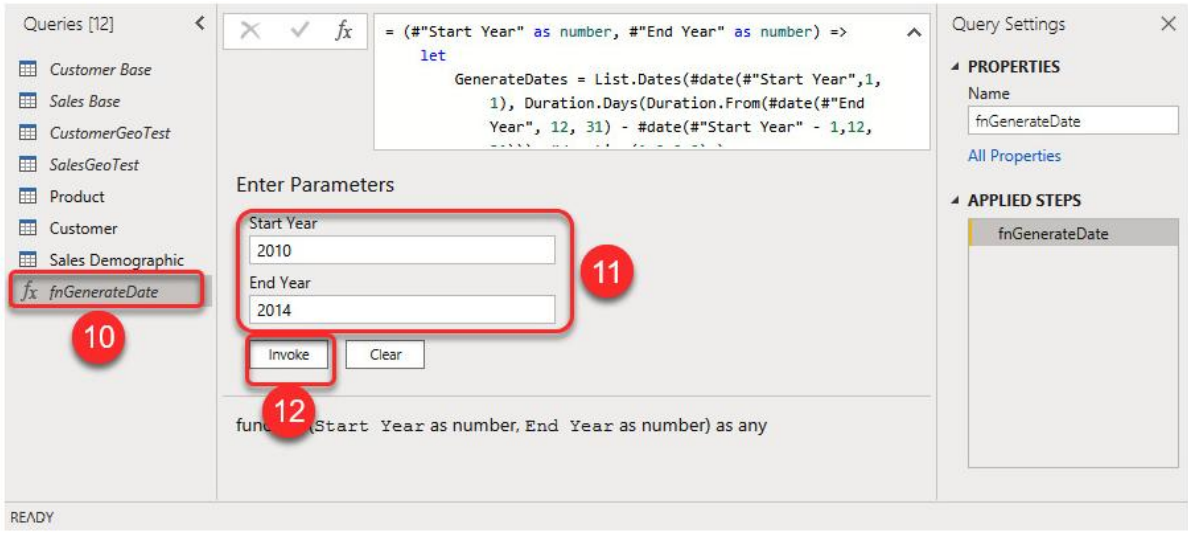

Figure 6.34 – Invoking the fnGenerateDate function

Invoking the **fnGenerateDate** function creates a new table named **Invoked Function**. Rename it **Date**, as shown in the following screenshot:

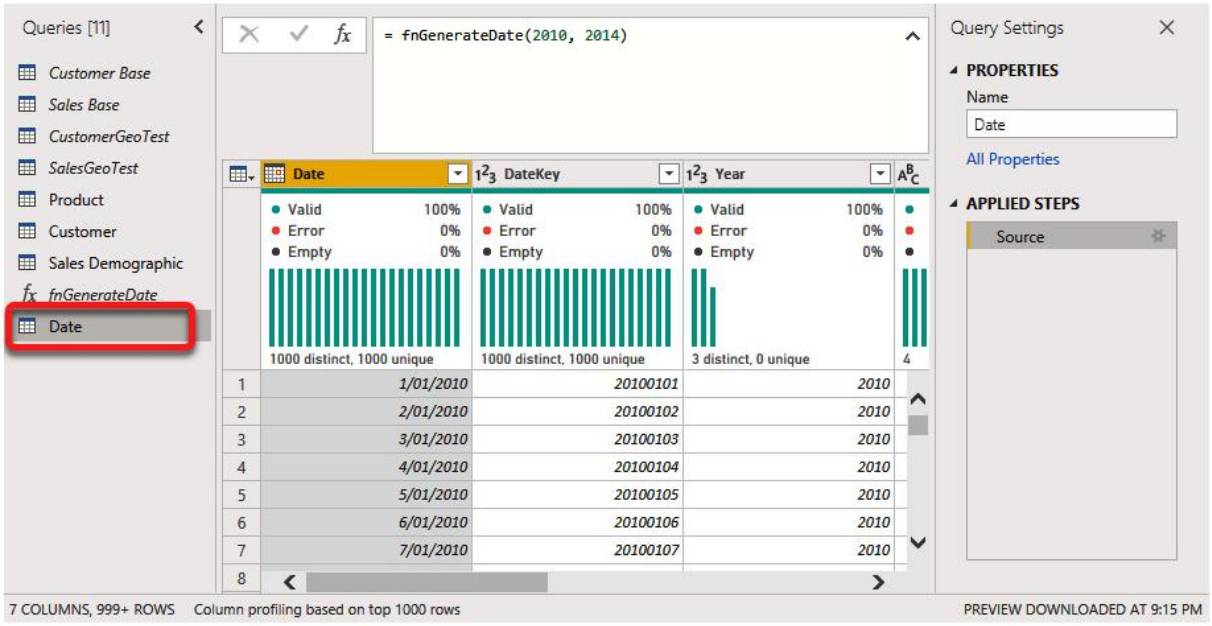

Figure 6.35 – Renaming the invoked custom function Date

So far, we've created the **Date** dimension. Now, let's create the **Time** dimension.

Time

As we mentioned previously, in the requirement gathering workshops with the business, we found out that both the **Date** and **Time** dimensions are required. In the previous section, we created a custom function to generate a **Date** dimension. As we discussed in *Chapter 2*, *Data Analysis eXpressions and Data Modeling*, the **Time** dimension can be created using DAX. In this section, we'll discuss how to create the **Time** dimension in Power Query.

The reason we need to create the **Time** dimension is trivial. We need it so that we can analyze our data over different elements of time, such as hour, minute, second, or time buckets (or time bands) such as 5 min, 15 min, 30 min, and so on.

The following Power Query expression creates a **Time** dimension with 5 min, 15 min, 30 min, 45 min, and 60 min time bands:

```
let
Source = Table.FromList({1..86400}, Splitter.SplitByNothing()),
    #''Renamed Columns'' = Table.RenameColumns(Source,{{''Column1'', ''ID''}}),
    #''Time Column Added'' = Table.AddColumn(#''Renamed Columns'', ''Time'', each Time.From(#datetime(1970,1,1,0,0,0) + #duration(0,0,0,
        [ID]))),
    #''Hour Added'' = Table.AddColumn(#''Time Column Added'', ''Hour'', each Time.Hour([Time])),
    #''Minute Added'' = Table.AddColumn(#''Hour Added'', ''Minute'', each Time.Minute([Time])),
    #''5 Min Band Added'' = Table.AddColumn(#''Minute Added'', ''5 Min Band'', each Time.From(#datetime(1970,1,1,0,0,0) + #duration(0, 0,
        Number.RoundDown(Time.Minute([Time])/5) * 5, 0))  +  #duration(0, 0, 5, 0)),
    #''15 Min Band Added'' = Table.AddColumn(#''5 Min Band Added'', ''15 Min Band'', each Time.From(#datetime(1970,1,1,0,0,0) + #duration(0,
        0, Number.RoundDown(Time.Minute([Time])/15) * 15, 0))  +  #duration(0, 0, 15, 0)),
    #''30 Min Band Added'' = Table.AddColumn(#''15 Min Band Added'', ''30 Min Band'', each Time.From(#datetime(1970,1,1,0,0,0) +
        #duration(0, 0, Number.RoundDown(Time.Minute([Time])/30) * 30, 0))  +  #duration(0, 0, 30, 0)),
    #''45 Min Band Added'' = Table.AddColumn(#''30 Min Band Added'', ''45 Min Band'', each Time.From(#datetime(1970,1,1,0,0,0) +
        #duration(0, 0, Number.RoundDown(Time.Minute([Time])/45) * 45, 0))  +  #duration(0, 0, 45, 0)),
    #''60 Min Band Added'' = Table.AddColumn(#''45 Min Band Added'', ''60 Min Band'', each Time.From(#datetime(1970,1,1,0,0,0) +
        #duration(0, 0, Number.RoundDown(Time.Minute([Time])/60) * 60, 0))  +  #duration(0, 0, 60, 0)),
    #''Removed Other Columns'' = Table.SelectColumns(#''60 Min Band Added'',{''Time'', ''Hour'', ''Minute'', ''5 Min Band'', ''15 Min
        Band'', ''30 Min Band'', ''45 Min Band'', ''60 Min Band''}),
    #''Changed Type'' = Table.TransformColumnTypes(#''Removed Other Columns'',{{''Time'', type time}, {''Hour'', Int64.Type}, {''Minute'',
        Int64.Type}, {''5 Min Band'', type time}, {''15 Min Band'', type time}, {''30 Min Band'', type time}, {''45 Min Band'', type time},
        {''60 Min Band'', type time}})
in
#''Changed Type''
```

NOTE

The preceding code is also available in this book's GitHub repository, in the ***Chapter 6, Generate Time Dimension.m*** *file, via the following URL:*
https://github.com/PacktPublishing/Expert-Data-Modeling-with-Power-BI/blob/master/Chapter%206%2C%20Generate%20Time%20Dimension.m.

Now that we have the preceding code at hand, we need to create a new **Blank Query**, name it **`Time`**, and then copy and paste the preceding expressions into **Advanced Editor**, as shown in the following screenshot:

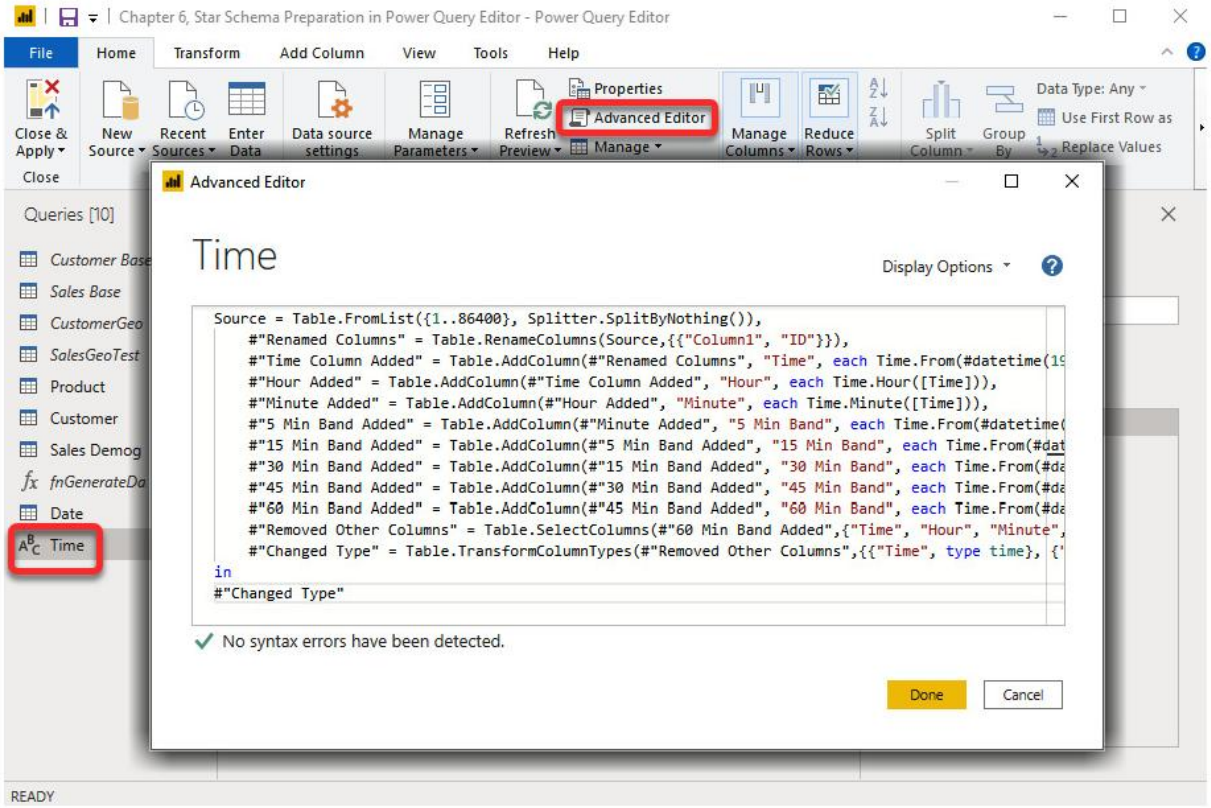

Figure 6.36 – Creating the Time dimension in Power Query Editor

In this and the previous section, we created the **`Date`** and **`Time`** dimensions in Power Query. You may be wondering how this is different from creating those dimensions in DAX, which brings us to the next section.

Creating Date and Time dimensions – Power Query versus DAX

In the last two sections, we discussed creating the **Date** and the **Time** dimensions in Power Query. We also discussed creating both dimensions with DAX in *Chapter 2*, *Data Analysis eXpressions and Data Modeling*. In this section, we'll discuss the differences between these two approaches.

Generally speaking, once we've load the tables into the data model, both approaches would work the same and have similar performance. But there are also some differences which may make us pick one approach over the other, as follows:

- We can create the **Date** and **Time** dimensions in Power Query, either by creating a custom function or a static query. Either way, we can use the query to create Power BI Dataflows and make them available across the organization. This is not currently possible with DAX.
- The calculated tables that are created in DAX are not accessible within the Power Query layer. Therefore, we can't do any equations in Power Query over the **Date** or **Time** dimensions if necessary.
- If we need to consider local holidays in the **Date** table, we can connect to the public websites over the internet and mash up that data in Power Query. This option is NOT available in DAX.
- If we need to consider all the columns with **Date** or **DateTime** data types across the data model, then using **CALENDARAUTO()** in DAX is super handy. A similar function does not currently exist in Power Query.
- Our knowledge of Power Query and DAX is also an essential factor to consider. Some of us are more comfortable with one language than the other.

Creating fact tables

Now that we've created all the dimensions, it is time to create a fact table that contains numeric values and the primary keys of the dimensions as foreign keys. Looking at the **Sales Base** and **Customer Base** data shows us that the **Sales Base** table holds many transactions with numeric values. Therefore, a fact table can be derived from the **Sales Base** table, which we will call **Sales**, by following these steps:

1. Reference the **Sales Base** table and then rename the new table **Sales**.

   We want to get **ProductKey** from the **Product** table. To do so, we can merge the **Sales** table with the **Product** table.

2. Click **Merge Queries**.
3. Select the **ProductCategory**, **ProductSubcategory**, and **Product** columns, respectively.
4. Select the **Product** table from the dropdown list.
5. Again, select the **ProductCategory**, **ProductSubcategory**, and **Product** columns.
6. Select **Left outer (all from first, matching from second)** from the **Join Kind** dropdown list.
7. Click **OK**:

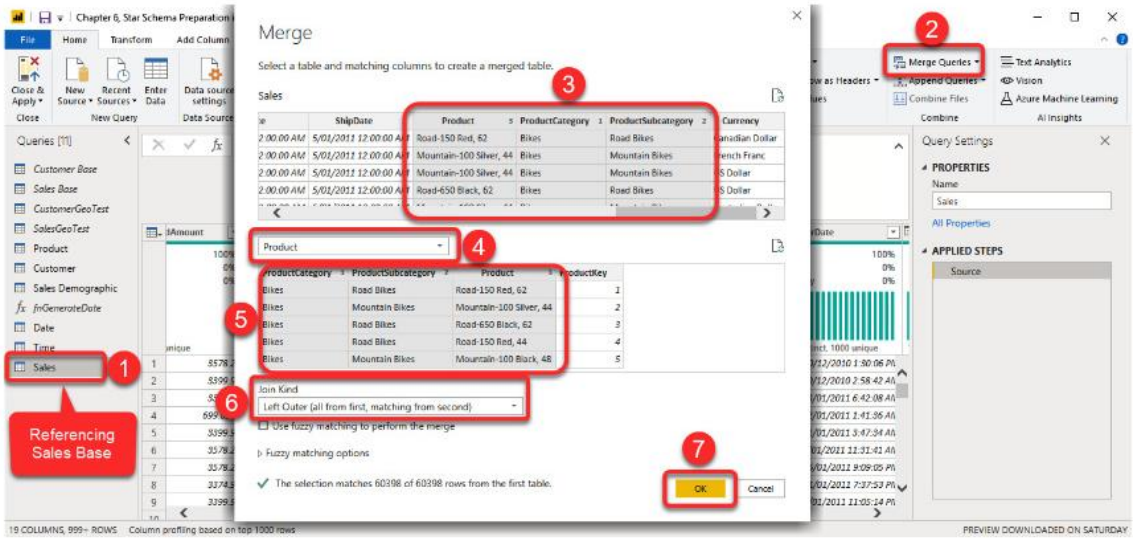

Figure 6.37 – Merging the Sales table with the Product table

8. This creates a new structured column named **`Product.1`** and a new transformation step in **Applied Steps**. The default name for this step is **Merged Queries**. Rename it **Merged Sales with Product**.
9. Expand the **`Product.1`** column.
10. Tick the **`ProductKey`** column (keep the rest unticked).
11. Untick the **Use original column name as prefix** option.
12. Click **OK**:

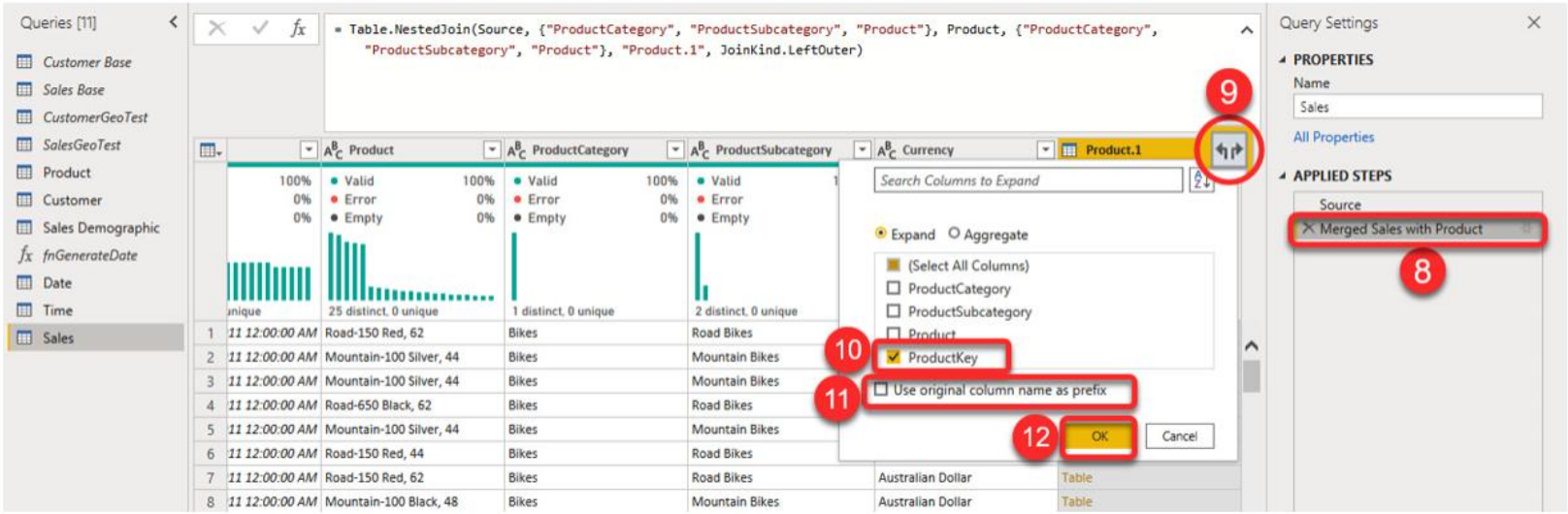

Figure 6.38 – Expanding the Product.1 structured column

Now, we need to get **SalesDemographicKey** from the **Sales Demographic** table. The columns that make a unique identifier for each row in the **Sales Demographic** table are **SalesCountry**, **City**, **StateProvinceName**, and **PostalCode**. However, the **Sales** table does not contain all those columns. Besides, the **Sales Demographic** dimension is derived from the **Customer Base** table. Therefore, we have to merge the **Sales** table with the **Customer Base** table via the **CustomerKey** column, and then merge again with the **Sales Demographic** table to reach **SalesDemographicKey**.

13. Click **Merge Queries** again.
14. Select the **CostomerKey** column under the **Sales** section.
15. Select **Customer Base** from the dropdown list.
16. Select **CustomerKey**.
17. Select **Left Outer** for **Join Kind**.

18. Click **OK**:

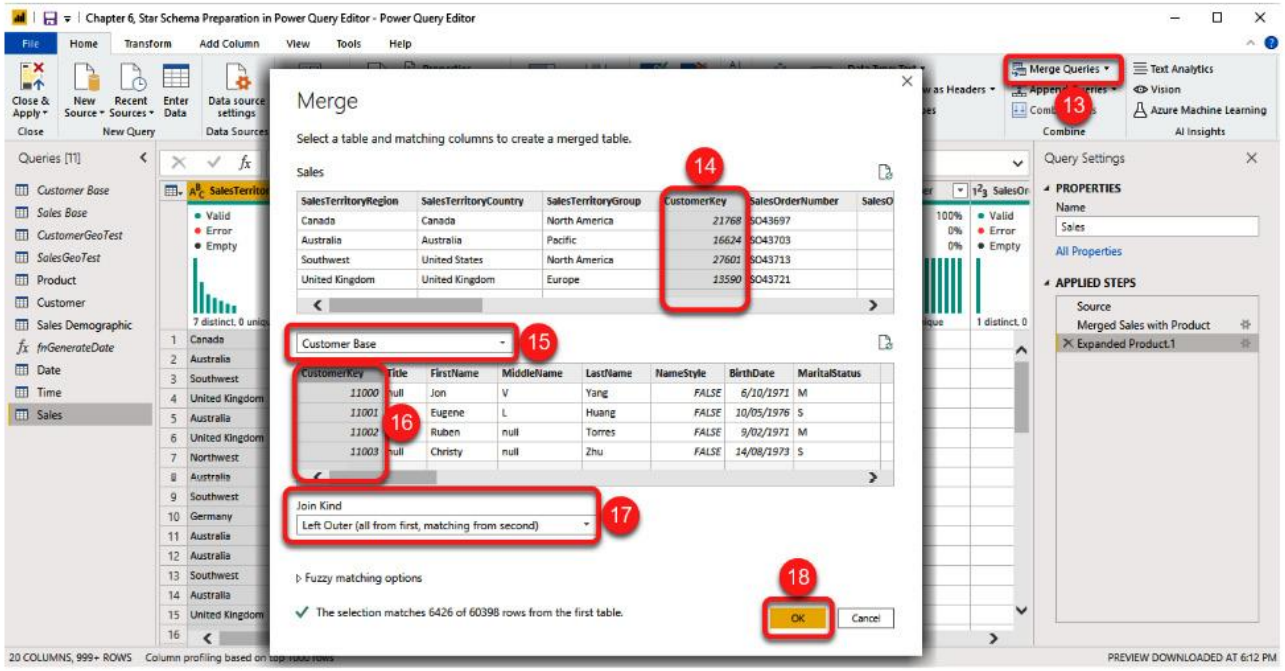

Figure 6.39 – Merging the Sales table with Customer Base

This creates a new structured column named **`Customer Base`**. It also creates a new transformation step in **Applied Steps** named **Merged Queries**.

19. Rename this step **Merged Sales with Customer Base**.
20. Expand the **`Customer Base`** structured column.
21. Keep the **`SalesCountry`**, **`City`**, **`StateProvinceName`**, and **`PostalCode`** columns ticked and untick the rest.

22. Untick the **Use original column name as prefix** option.

23. Click **OK**:

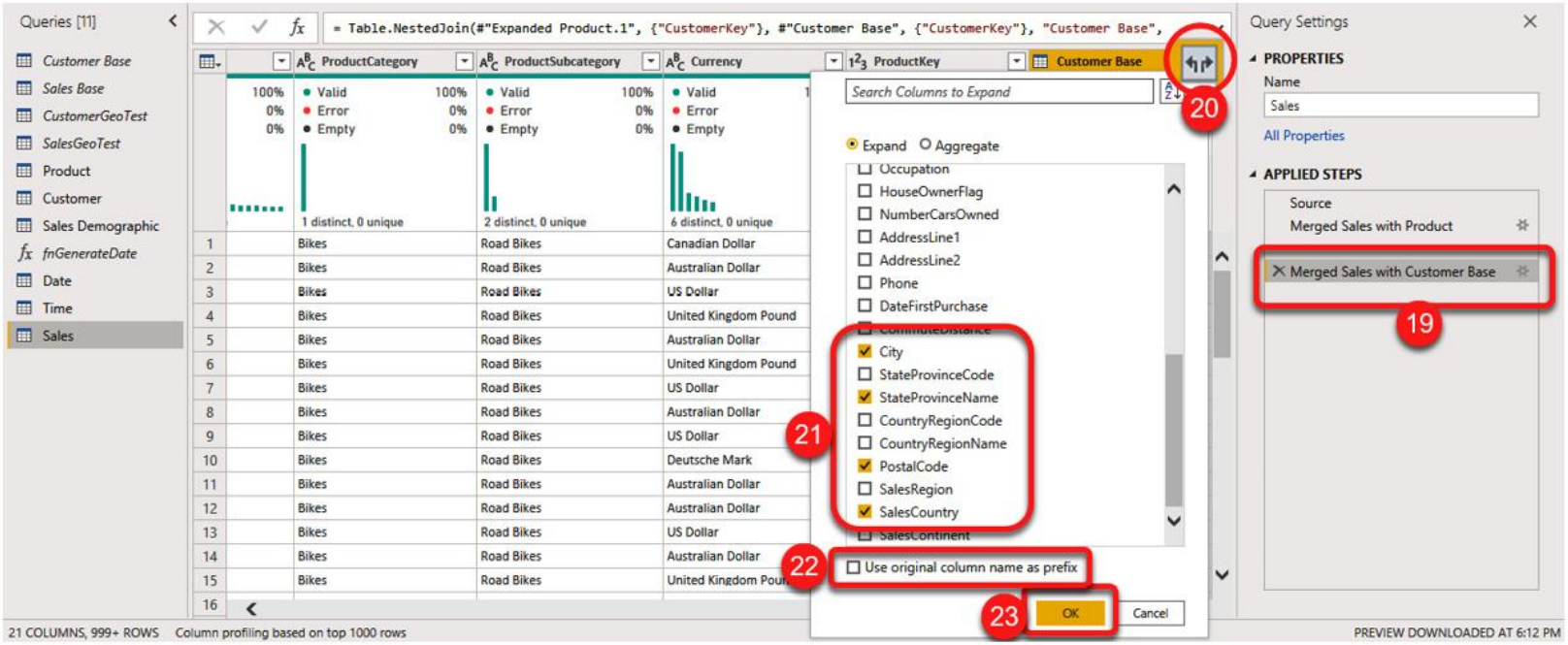

Figure 6.40 – Expanding the Customer Base structured column

Now, we need to merge the results with the **`Sales Demographic`** table and get our **`SalesDemographicKey`**.

24. After clicking **Merge Queries** again, select the **`SalesCountry`**, **`City`**, **`StateProvinceName`**, and **`PostalCode`** columns.

25. Select the **`Sales Demographic`** table from the dropdown.

26. Select the **`SalesCountry`**, **`City`**, **`StateProvinceName`**, and **`PostalCode`** columns, respectively. Remember, the sequence is important.

27. Keep **Join Kind** set to **Left Outer**.

28. Click **OK**:

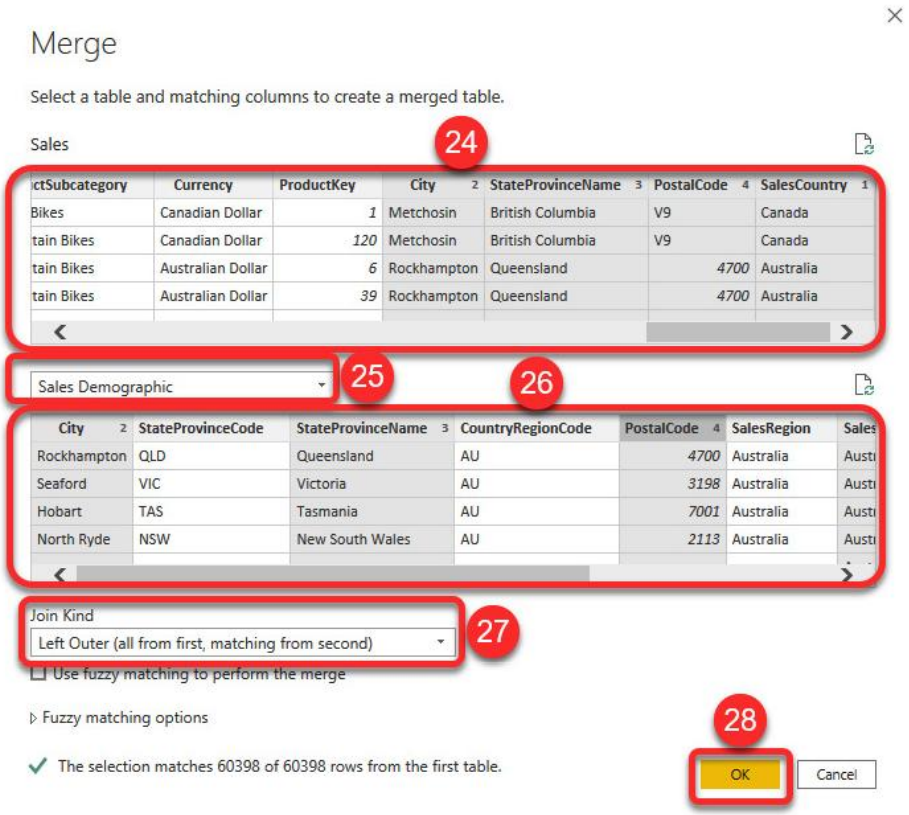

Figure 6.41 – Merging Sales with Sales Demographic

This creates a new structured column named **`Sales Demographic`**. A new **Merged Queries** step is also created in **Applied Steps**.

29. Rename the **Merged Queries** step **Merged Sales with Sales Demographic**.
30. Expand the **`Sales Demographic`** structured column.
31. Untick all the columns except for the **`SalesDemographicKey`** column.
32. Untick the **Use original column name as prefix** option.
33. Click **OK**:

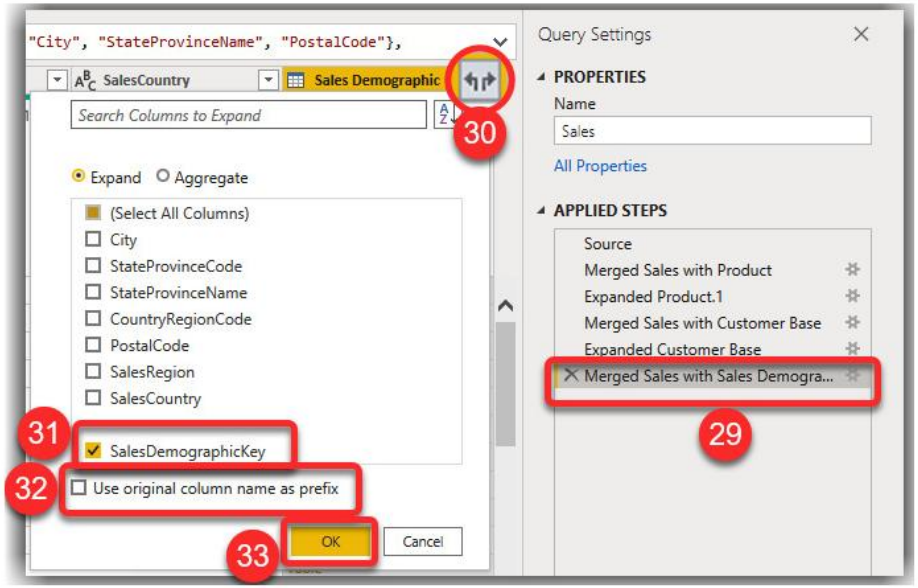

Figure 6.42 – Merging Sales with Sales Demographic

Now that we've added **`SalesDemographicKey`** to the **`Sales`** table, it is time to look at the columns in the **`Sales`** table with either **`Date`** or **`DateTime`** data types. The **`Sales`** table has three columns with the **`DateTime`** data type. Looking closer at the data shows that **`OrderDate`** is the only one that is actually in **`DateTime`**, while the other two represent **`Date`** values as the **`Time`** part of all values is **`12:00:00 AM`**. Therefore, it is better to convert the data type into **`Date`**. The **`OrderDate`** column represents both **`Date`** and **`Time`**. The only remaining part is to get the **`Date`** and **`Time`** values separately so that they can be used in the data model relationships. So, we need to split **`OrderDate`** into two separate columns: **`Order Date`** and **`Order Time`**. Follow these steps to do so:

34. Select the **`OrderDate`** column.
35. Click the **Split Column** button from the **Home** tab of the ribbon.
36. Click **By Delimiter**.
37. Select **Space** as the delimiter.
38. Click **Left-most delimiter** for **Split at**.
39. Click **OK**:

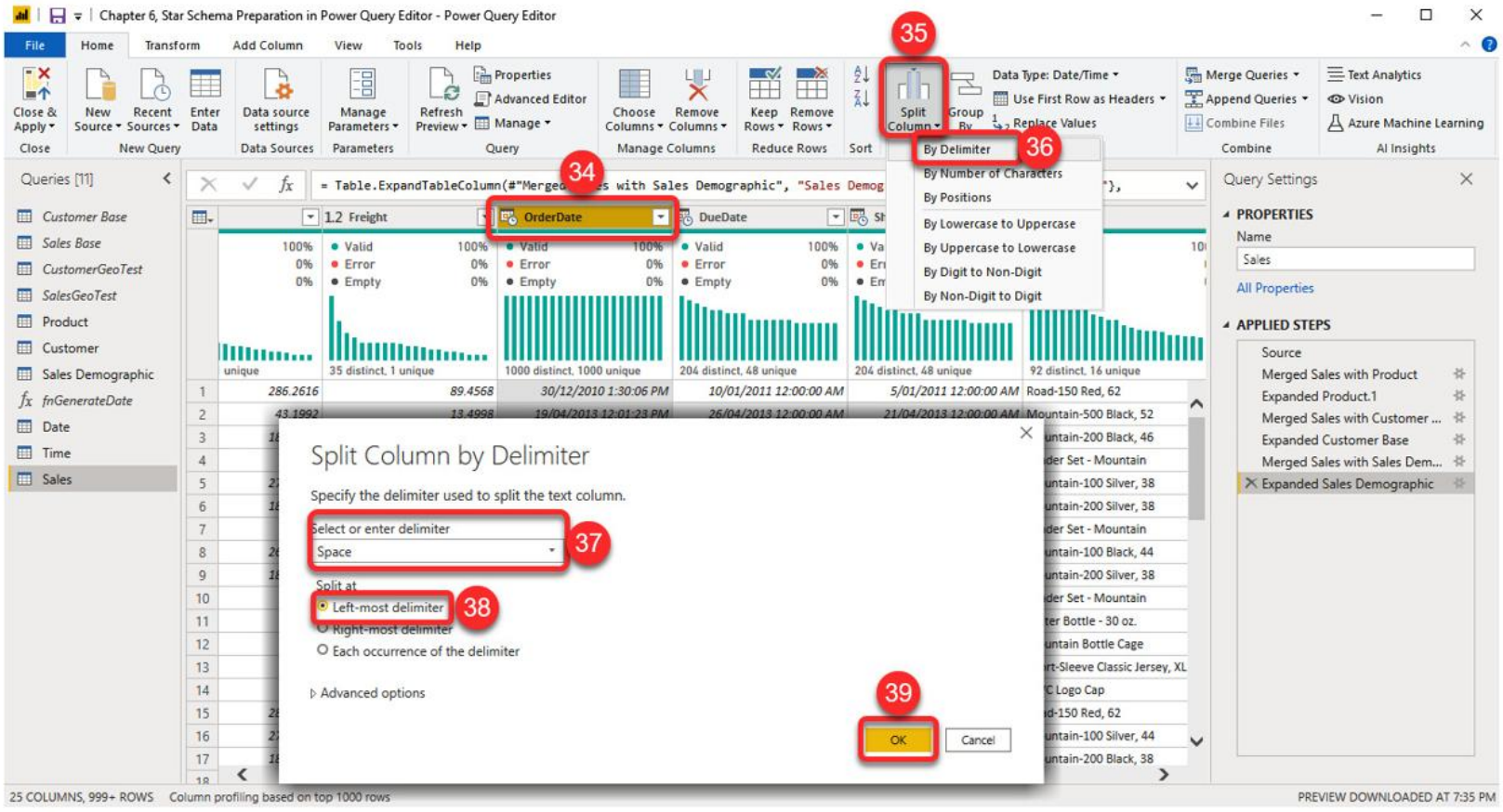

Figure 6.43 – Splitting OrderDate by delimiter

40. Rename the **Split Column by Delimiter** step to **Split OrderDate by Delimiter** from **Applied Steps**.

41. From the formula bar, change **`OrderDate.1`** to **`Order Date`** and change **`OrderDate.2`** to **`Order Time`**. Then, **Submit** these changes:

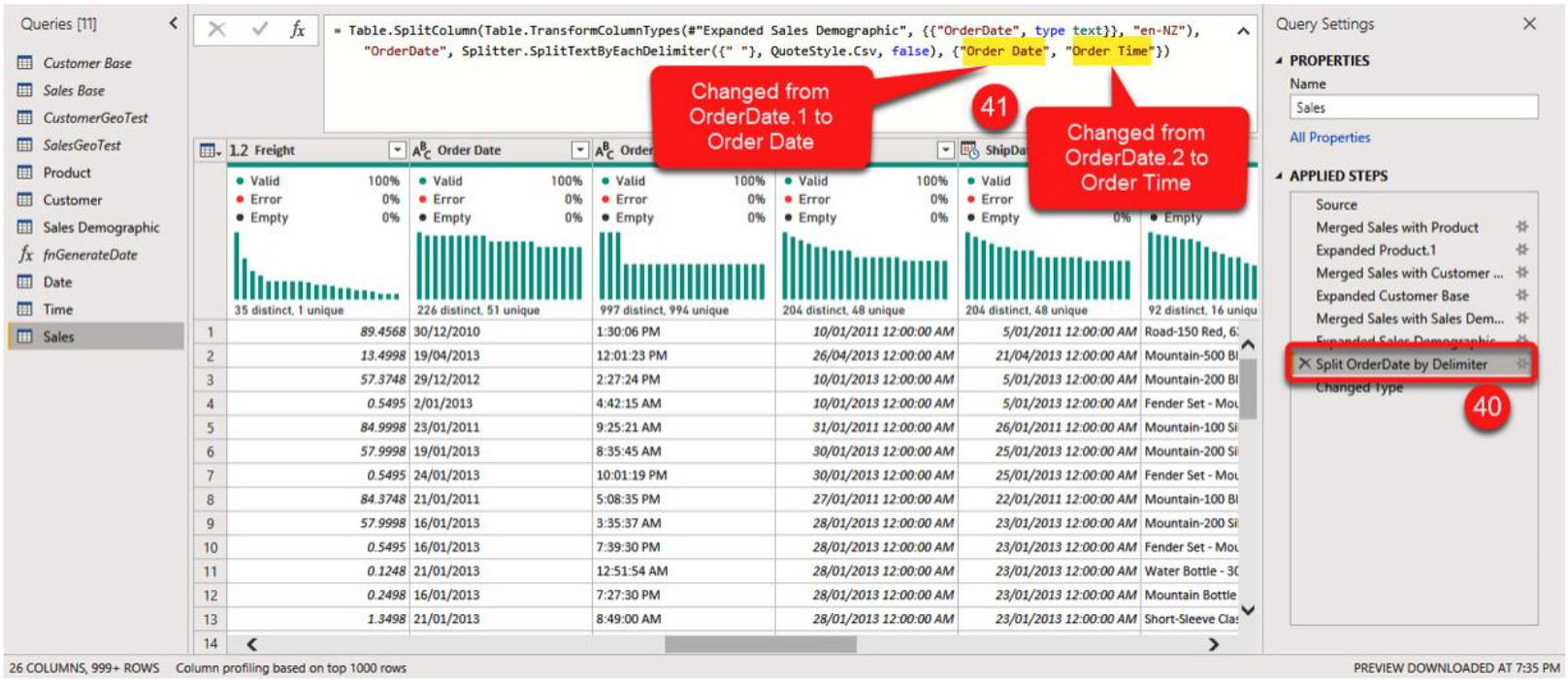

Figure 6.44 – Changing the split column names from the formula bar

A **Changed Type** step will be added automatically if the **Type Detection** setting is set to detect the data types. Keep this step. Now, we need to change the data type of the **`DueDate`** and the **`ShipDate`** columns from **`DateTime`** to **`Date`**.

42. Click the **Changed Type** step from the **Applied Step** pane.
43. Select both **DueDate** and **ShipDate**.
44. Click the **Data Type** drop-down button from the **Home** tab.
45. Select **Date**:

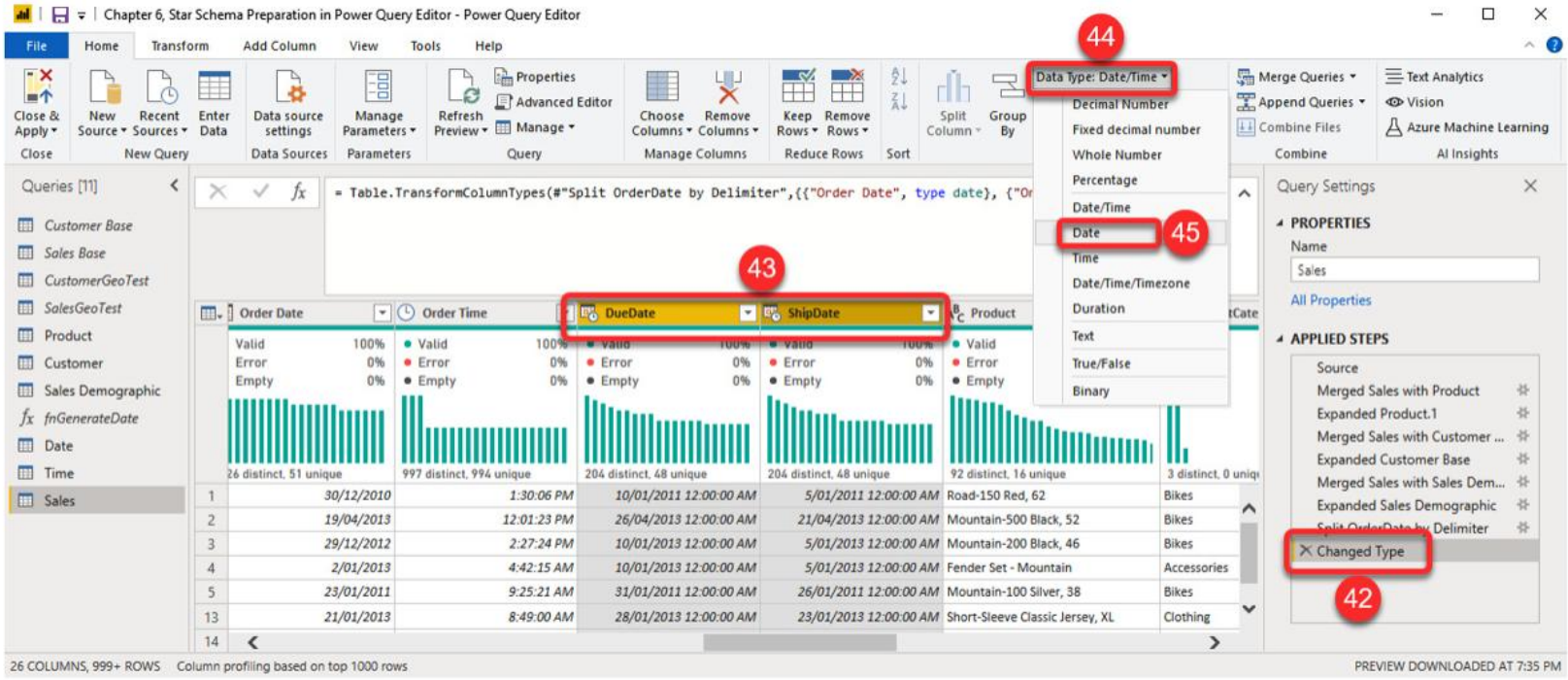

Figure 6.45 – Changing the DueDate and ShipDate data types

So far, we've added all the key columns from the dimensions to the fact table, which will be used to create the relationships between the dimensions and the fact table in the data model layer. The only remaining piece of the puzzle is to clean up the **Sales** table by removing all the unnecessary columns.

46. Click the **Choose Column** button from the **Home** tab.
47. Keep the following columns ticked and untick the rest; that is, **CustomerKey**, **SalesOrderNumber**, **SalesOrderLineNumber**, **OrderQuantity**, **ExtendedAmount**, **TotalProductCost**, **SalesAmount**, **TaxAmt**,**Freight**, **Order Date**, **Order Time**, **DueDate**, **ShipDate**, **Currency**, and **ProductKey**.
48. Click **OK**:

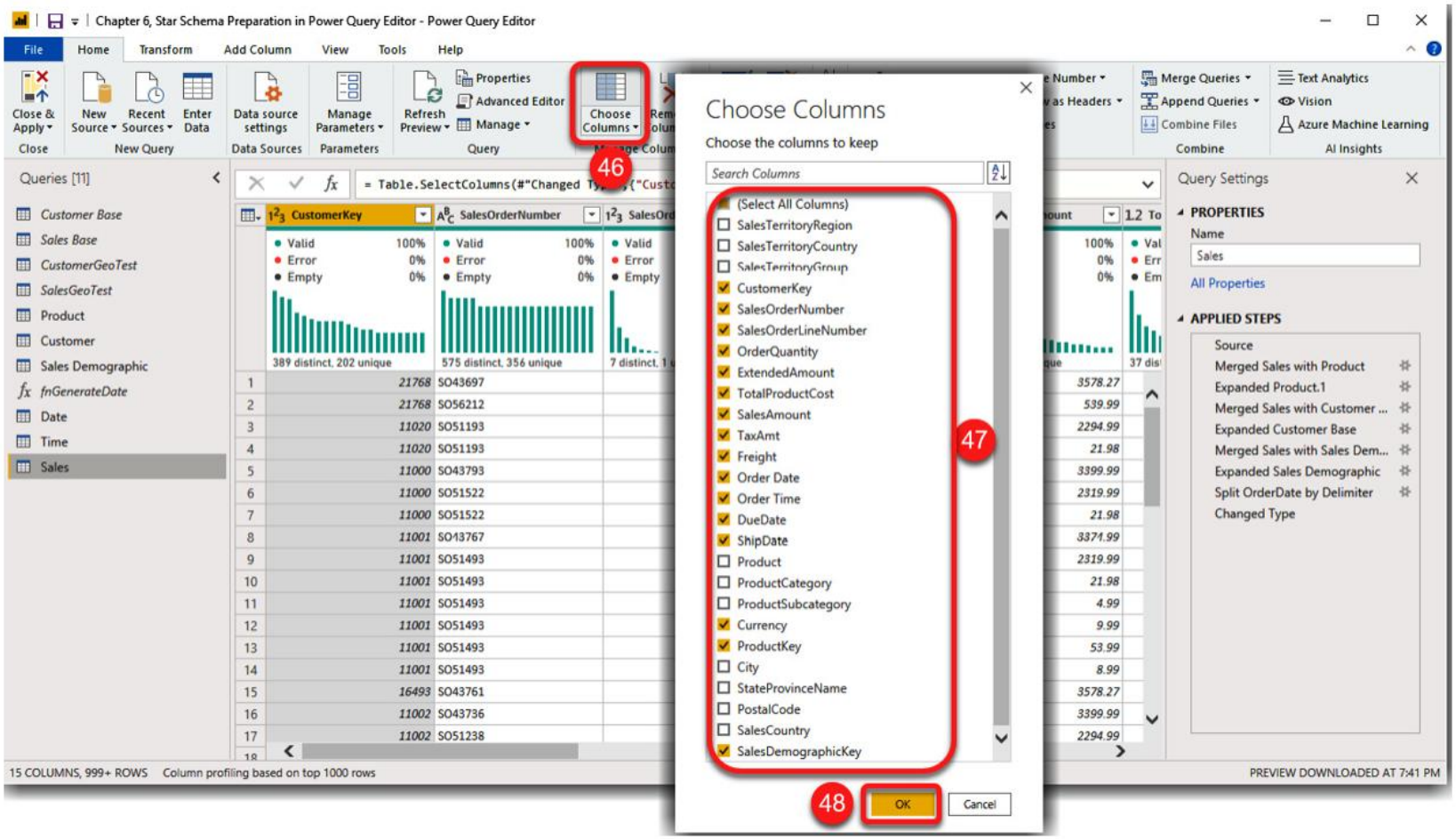

Figure 6.46 – Removing unnecessary columns from the Fact table

Looking at the results of the preceding transformation steps, the **Sales** fact table contains the following:

- The **OrderQuantity**, **ExtendedAmount**, **TotalProductCost**, **SalesAmount**, **TaxAmt**, **Freight** columns, which are facts.
- **CustomerKey**, **Order Date**, **Order Time**, **DueDate**, **ShipDate**, **ProductKey**, and **SalesDemographicKey** are foreign keys that will be used in the data modeling layer to create relationships between the **Sales** table and its dimensions.
- **SalesOrderNumber**, **SalesOrderLineNumber**, and **Currency** are degenerated dimensions.

Summary

In this chapter, we prepared the data in a Star Schema, which has been optimized for data analysis and reporting purposes on top of a flat data structure. We identified potential dimensions and discussed the reasons for creating or not creating separate dimension tables. We then went through the transformation steps to create the justified dimension tables. Finally, we added all the dimension key columns to the fact table and removed all the unnecessary columns, which gave us a tidy fact table that only contains all the necessary columns.

In the next chapter, we will cover an exciting and rather important topic: *Data preparation common best practices*. By following these best practices, we can avoid a lot of reworks and maintenance costs.