Get the best out of Power BI by building optimized data models for reporting and business needs

Soheil Bakhshi

Foreword by Christian Wade (Principal Program Manager, Microsoft)

Expert Data Modeling with Power BI

Get the best out of Power BI by building optimized data models for reporting and business needs

Soheil Bakhshi

BIRMINGHAM—MUMBAI

Expert Data Modeling with Power BI

Copyright © 2021 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Kunal Parikh

Publishing Product Manager: Sunith Shetty

Senior Editor: Mohammed Yusuf Imaratwale

Content Development Editor: Nazia Shaikh

Technical Editor: Devanshi Deepak Ayare

Copy Editor: Safis Editing

Project Coordinator: Aishwarya Mohan

Proofreader: Safis Editing

Indexer: Rekha Nair

Production Designer: Shankar Kalbhor

First published: June 2021

Production reference: 1120521

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80020-569-7

www.packt.com

I dedicate this book to my lovely wife, Elica Mehr, and our precious daughter, Avina.

Without their unconditional support, this matter would not have been possible.

I also owe this success to my parents, who always believed in me and encouraged me to follow my dreams.

Foreword

I am happy to know that Soheil, who I have seen grow as an MVP with expert knowledge of Power BI, has authored this hands-on book that covers a range of topics. Learning by example is something every committed student wants to do. The effort needed to go through online documentation and extract useful information can be prohibitive. Many students can become frustrated during the process and leave the task unfinished because it is easy for any beginner to lose sight of the learning task while trying to find good, real-world, example-based content.

Soheil's book includes many resources from beginner to expert level for Power BI. Power BI is the leading platform and tool for enterprise and self-service BI. It is a joy to use and is very flexible. This book is appealing both to beginners and to experts who build systems for thousands of users. I'm happy to recommend Soheil's book to any enthusiast, regardless of their level of expertise. His book navigates the world of Power BI in a very engaging way. It starts from the basic building blocks and elaborates on many aspects of data modeling, including star schema, managing bi-directional relationships, many-to-many relationships, calculated tables, preparing the data structure in Power Query to achieve a star schema design, RLS, OLS, and composite models. Any student will appreciate the hands-on approach of this book. Every concept is explained with examples.

Christian Wade

Principal Program Manager, **Microsoft**

Contributors

About the author

Soheil Bakhshi is the founder of Data Vizioner and is a sought-after BI consultant. Having worked in data and analytics for more than 20 years, Soheil's experience lies in Microsoft BI, data warehousing, and the Power BI platform. He possesses MSCE and MCSA certifications and is a Microsoft **MVP** (**Most Valuable Professional**). He has a passion for sharing knowledge via his website `www.biinsight.com` and for speaking at conferences and Power BI community events locally and globally. In following his desire for simplicity and efficiency, he has been behind Power BI community tools and commercial products such as Power BI Exporter and Power BI Documenter.

About the reviewers

Felipe Vilela worked for many years with system development and then started working in BI/data warehousing, mainly using MicroStrategy, more than 8 years ago. He worked with many companies in Brazil and the United States, implementing MicroStrategy projects, customizing, and administrating MicroStrategy. He taught BI/data warehousing and MicroStrategy to many companies using the company's, personal, and MicroStrategy's official courses. He also has a blog (`www.vilelamstr.com`). He was one of the MicroStrategy mobile app developers at MicroStrategy World 2016 and 2017. He has more than 30 MicroStrategy certifications, including the MCEP certification.

Nikita Barsukov is an experienced data scientist, focusing on delivering end-to-end analytical solutions. After growing up in Ukraine and studying in Finland and Sweden, he started his career in IT as a software developer in testing, only to realize his true passion lies in data science and building analytics tools that bring others joy and enhance data discovery. Nikita is currently employed by Microsoft; he is a part of a team of like-minded colleagues that develop analytics solutions for Power Platform and Dynamics 365. Outside of work, Nikita listens to podcasts and audiobooks; plays board games with friends, and occasionally even with himself; and enjoys a good run, craft beer, and a great book. Nikita lives in Copenhagen with his wife and three kids.

Ana Maria is a BI consultant and trainer as well as a Microsoft Data Platform MVP, Microsoft Partner in Power BI, and LinkedIn Learning Trainer. She has more than 25 years of industry experience ranging from the development of desktop solutions in FoxPro in the 1990s to data analysis and BI consulting and training. She has a degree in economic cybernetics from Moscow's State University of Management, in the former USSR, and also a master's degree in BI from the University of Alcala, in Spain. She is focused on the data world in Microsoft technology, modeling, and data analysis with SQL Server BI, Excel BI, Azure Machine Learning, R, and Power BI. You can find her at technical events and forums, as a speaker, organizer, or attendee.

Table of Contents

Preface

Section 1: Data Modeling in Power BI

1
Introduction to Data Modeling in Power BI

Understanding the Power BI layers 4
The data preparation layer (Power Query) 6
The data model layer 6
The data visualization layer 9
How data flows in Power BI 11

What data modeling means in Power BI 12
Semantic model 13
Building an efficient data model in Power BI 13
Star schema (dimensional modeling) and snowflaking 16

Power BI licensing considerations 25
Maximum size of individual dataset 26
Incremental data load 27
Calculation groups 27
Shared datasets 28
Power BI Dataflows 28

The iterative data modeling approach 29
Information gathering from the business 30
Data preparation based on the business logic 30
Data modeling 30
Testing the logic 31
Demonstrating the business logic in a basic data visualization 31
Thinking like a professional data modeler 32

Summary 32

2

Data Analysis eXpressions and Data Modeling

Understanding virtual tables 34

Creating a calculated table 34

Using virtual tables in a measure – Part 1 36

Using virtual tables in a measure – Part 2 39

Visually displaying the results of virtual tables 41

Relationships in virtual tables 44

Time intelligence and data modeling 56

Detecting valid dates in the date dimension 57

Period-over-period calculations 66

Generating the date dimension with DAX 76

Creating a time dimension with DAX 84

Summary 87

Section 2: Data Preparation in Query Editor

3

Data Preparation in Power Query Editor

Introduction to the Power Query M formula language in Power BI 92

Power Query is CaSe-SeNsItIvE 92

Queries 93

Expressions 94

Values 94

Types 100

Introduction to Power Query Editor 101

Queries pane 103

Query Settings pane 106

Data View pane 109

Status bar 113

Advanced Editor 113

Introduction to Power Query features for data modelers 115

Column quality 116

Column distribution 120

Column profile 123

Understanding query parameters 124

Understanding custom functions 132

Recursive functions 138

Summary 140

4

Getting Data from Various Sources

Getting data from common data sources 142
- Folder 142
- CSV/Text/TSV 149
- Excel 156
- Power BI datasets 164
- Power BI dataflows 169
- SQL Server 171
- SQL Server Analysis Services and Azure Analysis Services 174
- OData Feed 177

Understanding data source certification 180
- Bronze 180
- Silver 181
- Gold/Platinum 181

Working with connection modes 181
- Data Import 182
- DirectQuery 183
- Connect Live 184

Working with storage modes 185

Understanding dataset storage modes 187

Summary 188

5

Common Data Preparation Steps

Data type conversion 192

Splitting column by delimiter 201

Merging columns 204

Adding a custom column 206

Adding column from examples 209

Duplicating a column 211

Filtering rows 214

Working with Group By 218

Appending queries 221

Merging queries 224

Duplicating and referencing queries 228

Replacing values 229

Extracting numbers from text 233

Dealing with Date, DateTime, and DateTimeZone 235

Summary 239

6

Star Schema Preparation in Power Query Editor

Identifying dimensions and facts 242
- Number of tables in the data source 244
- The linkages between existing tables 244
- Finding the lowest required grain of Date and Time 246
- Defining dimensions and facts 248

Creating Dimensions tables 252
Geography 253
Sales order 257
Product 259
Currency 263
Customer 263
Sales Demographic 265
Date 269
Time 273
Creating Date and Time dimensions – Power Query versus DAX 276
Creating fact tables 277
Summary 286

7

Data Preparation Common Best Practices

General data preparation considerations 288
Consider loading a proportion of data while connected to the OData data source 288
Appreciating case sensitivity in Power Query saves you from dealing with issues in data modeling 292
Be mindful of query folding and its impact on data refresh 292
Organizing queries in Query Editor 300
datatype conversion 302
Data conversion can affect data modeling 302
Include the datatype conversion in a step when possible 309
Consider having only one datatype conversion step 311
Optimizing the size of queries 312
Removing unnecessary columns and rows 312
Summarization (Group by) 313
Disabling query load 314
Naming conventions 314
Summary 315

Section 3: Data Modeling

8

Data Modeling Components

Data modeling in Power BI Desktop 320
Understanding tables 320
Table properties 321
Featured tables 325
Calculated tables 326
Understanding fields 332
Data types 332
Custom formatting 334
Columns 335

Hierarchies 346
Measures 347

Using relationships 353

Primary keys/foreign keys 354
Handling composite keys 355
Filter propagation behavior 364
Bidirectional relationships 366

Summary 370

9

Star Schema and Data Modeling Common Best Practices

Dealing with many-to-many relationships 372

Many-to-many relationships using a bridge table 375
Hiding the bridge table 384

Being cautious with bidirectional relationships 385

Dealing with inactive relationships 388

Reachability via multiple filter paths 388
Multiple direct relationships between two tables 390

Using configuration tables 393

Segmentation 393
Dynamic conditional formatting with measures 396

Avoiding calculated columns when possible 403

Organizing the model 406

Hiding insignificant model objects 406
Creating measure tables 409
Using folders 413

Reducing model size by disabling auto date/time 417

Summary 421

Section 4: Advanced Data Modeling

10

Advanced Data Modeling Techniques

Using aggregations 426

Implementing aggregations for non-DirectQuery data sources 427
Using the Manage Aggregations feature 441

Incremental refresh 455

Configuring incremental refresh in Power BI Desktop 457
Testing the incremental refresh 463

Understanding Parent-Child hierarchies 466

Identifying the depth of the hierarchy 468
Creating hierarchy levels 470

Implementing roleplaying dimensions 475
Using calculation groups 479
Requirements 479
Terminology 480
Implementing calculation groups to handle time intelligence 481
Testing calculation groups 487
DAX functions for calculation groups 490
Summary 491

11

Row-Level Security

What RLS means in data modeling 494
What RLS is not 494
RLS terminologies 495
Assigning members to roles in the Power BI service 498
Assigning members to roles in Power BI Report Server 499
RLS implementation flow 501
Common RLS implementation approaches 502
Implementing static RLS 502
Implementing dynamic RLS 511
Summary 528

12

Extra Options and Features Available for Data Modeling

Dealing with SCDs 530
SCD type zero (SCD 0) 532
SCD type 1 (SCD 1) 532
SCD type 2 (SCD 2) 532
Introduction to OLS 537
Implementing OLS 537
Validating roles 540
Assigning members to roles in the Power BI service 542
Validating roles in the Power BI service 543
Introduction to dataflows 545
Scenarios for using dataflows 546
Dataflow terminologies 547
Creating dataflows 548
Introduction to composite models 565
New terminologies 566
Summary 575

Other Books You May Enjoy

Index

Preface

Microsoft Power BI is one of the most popular business intelligence tools available on the market for desktop and the cloud. This book will be your guide to understanding the ins and outs of data modeling and how to create data models using Power BI confidently. You'll learn how to connect data from multiple sources, understand data, define and manage the relationships between data, and shape data models.

In this book, you'll explore how to use data modeling and navigation techniques to define relationships and create a data model before defining new metrics and performing custom calculations using modeling features. As you advance through the chapters, the book will demonstrate how to create full-fledged data models, enabling you to create efficient data models and simple DAX code with new data modeling features. With the help of examples, you'll discover how you can solve business challenges by building optimal data models and changing your existing data models to meet evolving business requirements. Finally, you'll learn how to use some new and advanced modeling features to enhance your data models to carry out a wide variety of complex tasks. By the end of this Power BI book, you'll have gained the skills you need to structure data coming from multiple sources in different ways to create optimized data models that support reporting and data analytics.

Who this book is for

This Power BI book is for BI users, data analysts, and analysis developers who want to become well-versed in data modeling techniques to make the most of Power BI. Basic knowledge of Power BI and star schema will help you to understand the concepts covered in this book.

What this book covers

Chapter 1, *Introduction to Data Modeling in Power BI*, briefly describes different functionalities of Power BI and why data modeling is important. This chapter also reveals some important notes to be considered around Power BI licensing, which potentially could affect your data model. This chapter introduces an iterative data modeling approach, which guarantees an agile Power BI implementation.

Chapter 2, *Data Analysis eXpressions and Data Modeling*, does not discuss a lot of DAX as in parts 3 and 4 of this book DAX is heavily used to solve different data modeling challenges. Therefore, we'll only focus on the DAX functionalities that are harder to understand and are very relevant to data modeling. This chapter starts with a quick introduction to DAX, then we jump straight to virtual tables and time intelligence functionalities and their applications in real-world scenarios.

Chapter 3, *Data Preparation in Power Query Editor*, quickly explains the components of Power Query and their application. It expresses the emphasis of creating query parameters and user-defined functions along with real-world use cases and scenarios to demonstrate how powerful they are in building much more flexible and maintainable models.

Chapter 4, *Getting Data from Various Sources*, explains getting data from different data sources that are more commonly used in Power BI. Then, the importance of data source certification is explained, which helps you set your expectations on the type of data you're going to deal with. This is especially helpful in estimating data modeling efforts. Different connection modes are also explained in this chapter.

Chapter 5, *Common Data Preparation Steps*, explains common data preparation steps along with real-world hands-on scenarios. A combination of what you have learned so far in this book with the steps to be discussed in this chapter gives you a strong foundation to go on to the next chapters and build your data models more efficiently. By learning these functionalities, you can deal with a lot of different scenarios in implementing different data models.

Chapter 6, *Star Schema Preparation in Power Query Editor*, explains how to prepare your queries based on the star schema data modeling approach with real-life scenarios. The Power Query M language will be heavily used in this chapter, so you will learn how to deal with real-world challenges along the way. As you have already learned common data preparation steps in the previous chapter, the majority of Power Query scenarios explained in this chapter will be easier to implement. You'll also learn how to build dimension tables and fact tables, and how to denormalize your queries when needed.

Chapter 7, *Data Preparation Common Best Practices*, explains common best practices in data preparation. Following these practices will help you build more efficient data models that are easier to maintain and more flexible to make changes to. Following these practices, you can also avoid common mistakes, which can make your life much easier.

Chapter 8, *Data Modeling Components*, explains data modeling components from a Power BI perspective along with real file examples. In this chapter, we heavily use DAX when applicable so having a basic understanding of DAX is essential. We also have a complete star schema model in Power BI. The concept of config tables is covered, which unlocks a lot of possibilities in handling more complex business logic in the data model. The chapter ends with data modeling naming conventions.

Chapter 9, *Star Shema and Data Modeling Common Best Practices*, explains common data modeling best practices to help you make better decisions while building your data model to prevent facing some known issues down the road. For instance, dealing with data type issues in key columns that are used in relationships is somewhat time-consuming to identify, but it's very easy to prevent. So, knowing data modeling best practices helps you save a lot of maintenance time and consequently saves you money.

Chapter 10, *Advanced Data Modeling Techniques*, explains special modeling techniques that solve special business requirements. A good data modeler is one who is always open to new challenges. You may face some of the advanced business requirements discussed in this chapter or you may face something different but similar. The message we want to send in this chapter is to think freely when dealing with new business challenges and try to be innovative to get the best results.

Chapter 11, *Row-Level Security*, explains how to implement **row-level security** (**RLS**) in a Power BI data model. Dealing with RLS can be complex and knowing how to deal with different situations needs deep knowledge on data modeling and filter propagation concepts. Our aim in this chapter is to transfer that knowledge to you so you can design and implement high-performing and low-maintenance data models.

Chapter 12, *Extra Options and Features Available for Data Modeling*, introduces data modeling options such as **Slowly Changing Dimensions** (**SCD**), **Object-Level Security** (**OLS**), dataflows, and composite models, giving you broad exposure to all those topics.

To get the most out of this book

You will need to download and install the latest version of Power BI Desktop. All expressions have been tested in the March release of Power BI Desktop and will work in the later versions released on later dates. In addition to Power BI Desktop, you will need to install and use DAX Studio and Tabular Editor.

Software/Hardware covered in the book	OS Requirements
Power BI Desktop `https://powerbi.microsoft.com/en-us/downloads/`	• Windows 8.1 / Windows Server 2012 R2 or later • .NET 4.6.2 or later • Internet Explorer 11 or later • Memory (RAM): At least 2 GB available, 4 GB or more recommended. • Display: At least 1440x900 or 1600x900 (16:9) required. Lower resolutions such as 1024x768 or 1280x800 aren't supported, as certain controls (such as closing the startup screen) display beyond those resolutions. • Windows display settings: If you set your display settings to change the size of text, apps, and other items to more than 100%, you may not be able to see certain dialogs that you must interact with to continue using Power BI Desktop. If you encounter this issue, check your display settings in Windows by going to Settings > System > Display, and use the slider to return display settings to 100%. • CPU: 1 gigahertz (GHz) 64-bit (x64) processor or better recommended
DAX Studio `https://daxstudio.org/downloads/`	• Windows 7 or later (Windows 10 recommended) • .NET Framework 4.7.1 or later
Tabular Editor `https://github.com/otykier/TabularEditor/releases/tag/2.16.0`	• Windows Server 2019 • Windows Server 2016 • Windows 7 or later (Windows 10 recommended) • Azure Analysis Services Client Libraries "AMO" version 18.4.0.5 (or newer)

> *Note*
>
> Since January 31, 2021, Power BI Desktop is no longer supported on Windows 7.

In some chapters, you may need to have a Power BI Service account. You can sign up for a Power BI Service as an individual. Read more here: `https://docs.microsoft.com/en-us/power-bi/fundamentals/service-self-service-signup-for-power-bi?WT.mc_id=5003466`.

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

This book assumes that you are familiar with data warehousing and star schema terminology . However, the book tries to give a brief explanation of some terminologies when required.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Expert-Data-Modeling-with-Power-BI`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781800205697_ColorImages.pdf`

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The `Customertable` is wide and tall."

A block of code is set as follows:

```
Sequential Numbers =
SELECTCOLUMNS(
    GENERATESERIES(1, 20, 1)
    , "ID"
    , [Value]
    )
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Click **New table** from the **Modeling** tab."

> **Tips or important notes**
> Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

Section 1: Data Modeling in Power BI

In this section, we quickly introduce data modeling in Power BI from a general point of view. We assume you know what Power Query is, what DAX is, and that you know the basic concepts of the star schema. In this section, you will learn about virtual tables and time intelligence functionalities in DAX and how you can implement a powerful model with real-world scenarios.

This section comprises the following chapters:

- *Chapter 1, Introduction to Data Modeling in Power BI*
- *Chapter 2, Data Analysis eXpressions and Data Modeling*

1 Introduction to Data Modeling in Power BI

Power BI is not just a reporting tool that someone uses to build sophisticated reports; it is a platform supplying a wide range of features from data preparation to data modeling and data visualization. It is also a very well-designed ecosystem, giving a variety of users the ability to contribute to their organization's data analysis journey in many ways, from sharing datasets, reports, and dashboards to using their mobile phones to add some comments to a report, ask questions, and circulate it back to relevant people. All of this is only possible if we take the correct steps in building our Power BI ecosystem. A very eye-catching and beautiful report is worth nothing if it shows incorrect business figures or if the report is too slow to render so the user does not really have the appetite to use it.

One of the most important aspects of building a good Power BI ecosystem is getting the data right. In real-world scenarios, you normally get data from various data sources. Getting data from the data sources and mashing it up is just the beginning. Then you need to come up with a well-designed data model that guarantees you always represent the right figures supporting the business logic so the report performs well.

In this chapter, we'll start by learning about the different Power BI layers and how data flows between the different layers to be able to fix any potential issues more efficiently. Then, we'll study one of the most important aspects of Power BI implementation, that is, data modeling. You'll learn more about data modeling limitations and availabilities under different Power BI licensing plans. Finally, we'll discuss the iterative data modeling approach and its different phases.

In this chapter, we'll cover the following main sections:

- Power BI Desktop layers
- What data modeling means in Power BI
- Power BI licensing considerations for data modeling
- The iterative data modeling approach

Understanding the Power BI layers

As stated before, Power BI is not just a reporting tool. As the focus of this book is data modeling, we would rather not explain a lot about the tool itself, but there are some concepts that should be pointed out. When we talk about data modeling in Power BI, we are indeed referring to Power BI Desktop as our development tool. You can think of Power BI Desktop like Visual Studio when developing an **SQL Server Analysis Services** (**SSAS**) Tabular model. Power BI Desktop is a free tool offering from Microsoft that can be downloaded from `https://powerbi.microsoft.com/en-us/downloads/`. So, in this book, we're referring to Power BI Desktop when we say Power BI unless stated otherwise.

The following illustration shows a very simple process we normally go through while building a report in Power BI Desktop:

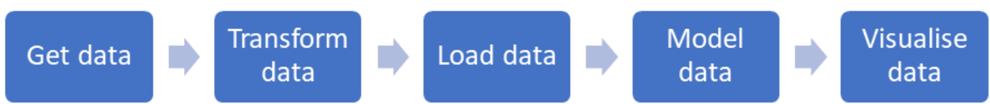


Figure 1.1 – Building a new report process in Power BI

To go through the preceding processes, we use different conceptual layers of Power BI. You can see those layers in Power BI Desktop as follows:

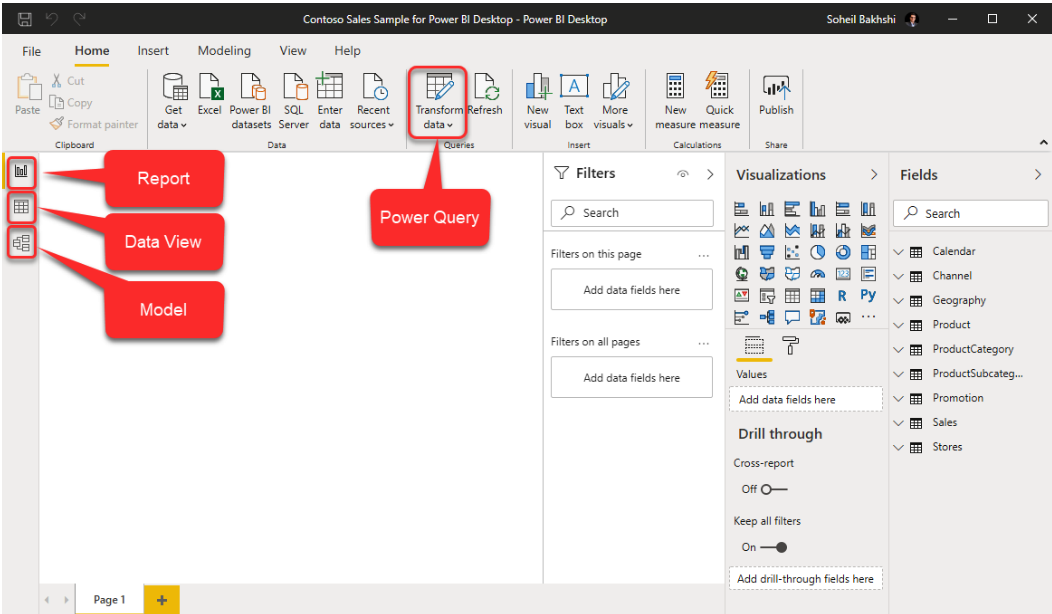


Figure 1.2 – Power BI layers

Download the Microsoft Contoso Sales sample for Power BI Desktop from `https://www.microsoft.com/en-us/download/confirmation.aspx?id=46801`.

Let's discuss each point in detail:

- The Power Query (data preparation) layer
- The data model layer
- The data visualization layer

The data preparation layer (Power Query)

In this layer, you get data from various data sources, transform and cleanse that data, and make it available for other layers. This is the very first layer that touches your data, so it is a very important part of your data journey in Power BI. In the Power Query layer, you decide which queries load data into your data model and which ones will take care of data transformation and data cleansing without loading the data into the data model:

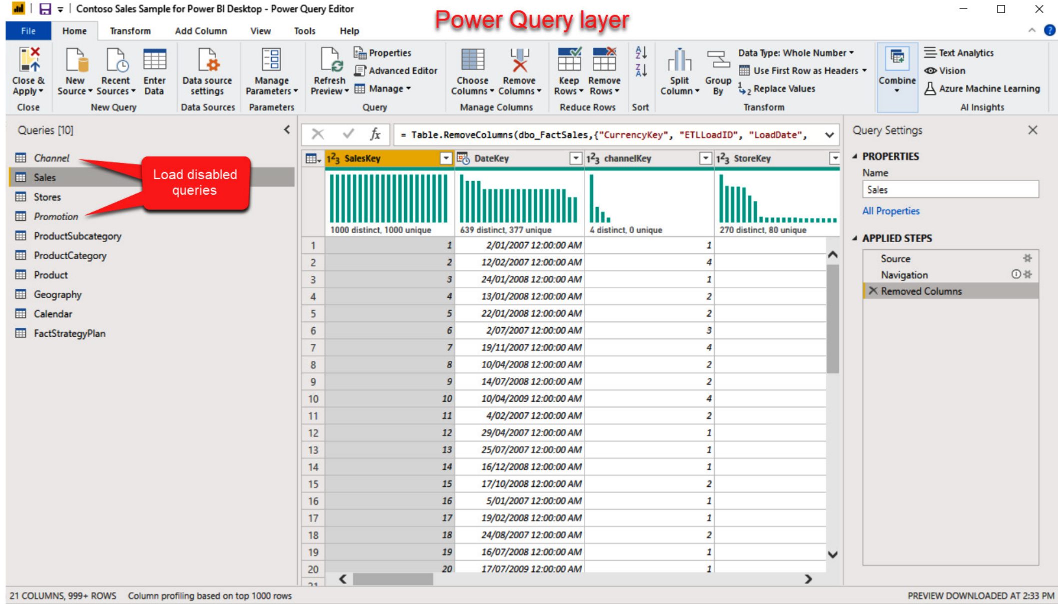


Figure 1.3 – Power Query

The data model layer

This layer has two views, the **Data view** and the **Model view**. In the Data view, you can see the data, and in the Model view, you can see the data models.

The Data view

After we are done with our data preparation in the Power Query layer, we load the data into the data model layer. Using the Data view, we can see the underlying data in our data model layer after it has been transformed in the data preparation layer. Depending on the connection mode, this view may or may not be accessible. While we can see the output of the data preparation, in this view we also take some other actions, such as creating analytical objects such as calculated tables, calculated columns, and measures, or copying data from tables.

> **Note**
>
> All objects we create in DAX are a part of our data model.

The following screenshot shows the Data view in Power BI Desktop when the storage mode of the table is set to **Import**:

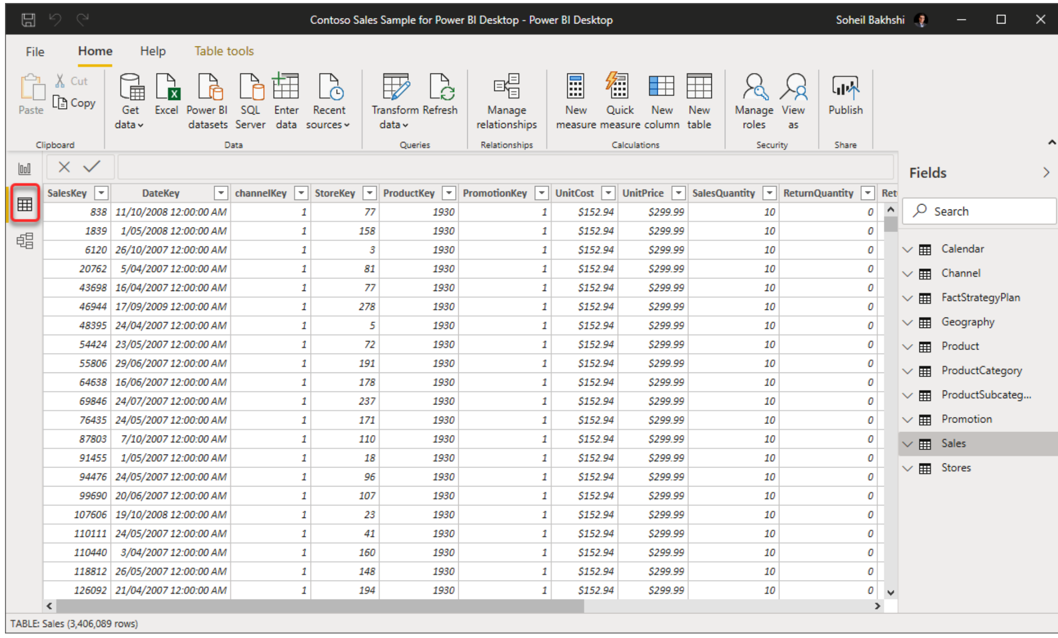


Figure 1.4 – Data view; storage mode: Import

The Data view tab does not show the underlying data if the table only shows the data when the storage mode is set to **Import**. If the storage mode is set to **DirectQuery**, the data will not be shown in the Data view:

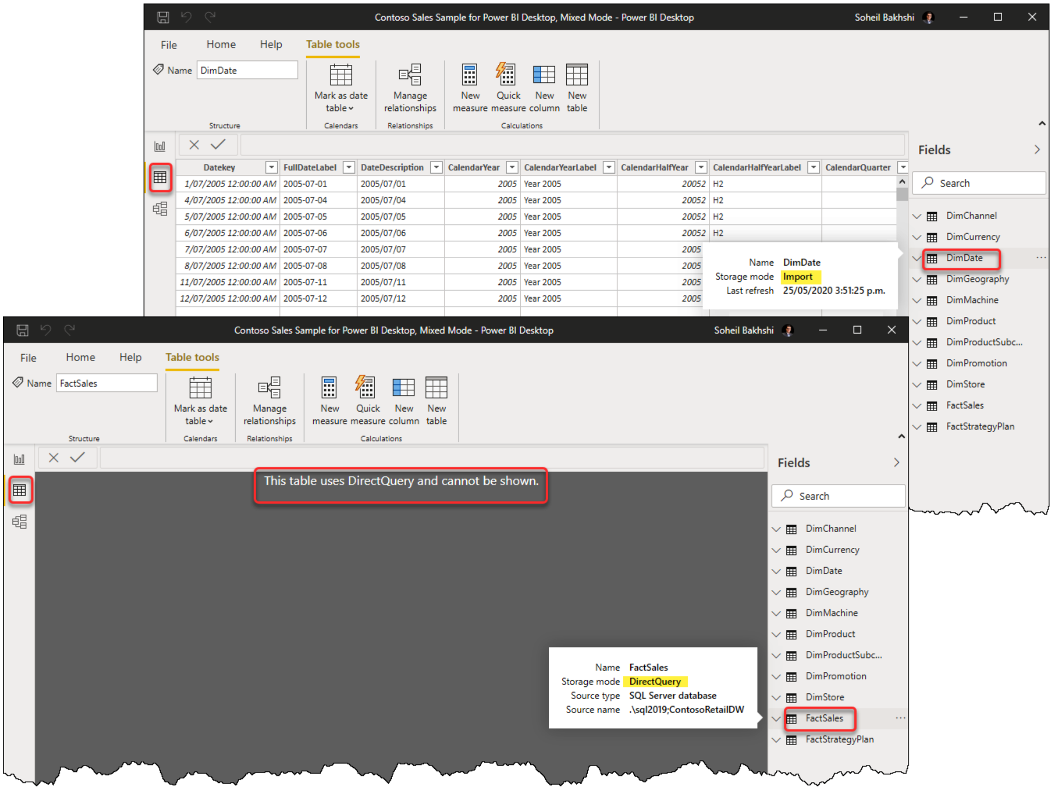


Figure 1.5 – Data view; storage mode: DirectQuery

The Model view

As its names implies, the *Model* view is where we stitch all the pieces together. Not only can we visually see how the tables are related in the model section, but also, we can create new relationships, format fields and synonyms, show/hide fields, and so on:

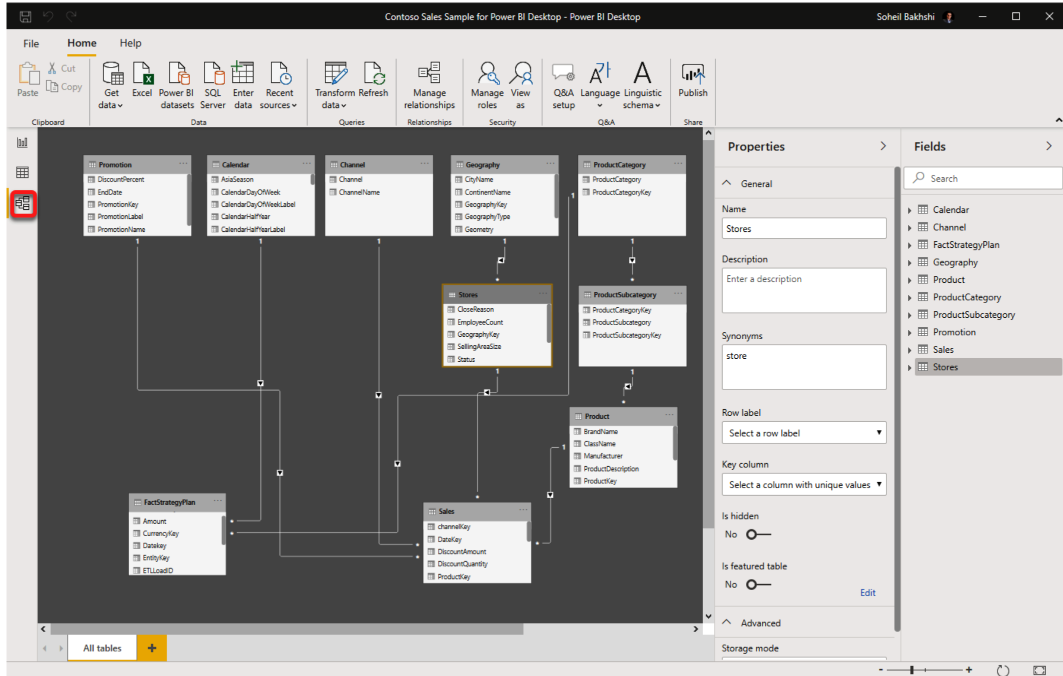


Figure 1.6 – Model view

The data visualization layer

In this layer, we bring the data to life by making meaningful and professional-looking data visualizations. This layer is accessible from the Report view, which is the default view in Power BI Desktop.

The Report view

In the Report view, we can build storytelling visualizations to help businesses make data-driven decisions on top of their data. For more convenience, we also create analytical calculations with DAX, such as calculated tables, calculated columns, and measures from the **Fields** pane in the Report view, but this doesn't mean those calculation objects are a part of the data visualization layer. Indeed, those calculations are a part of the data model layer:

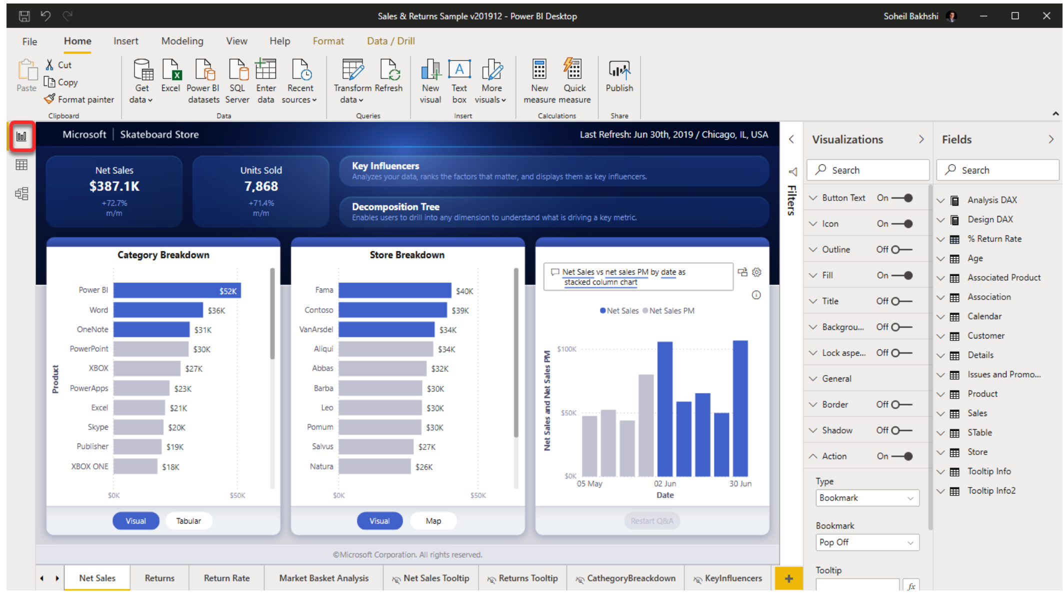


Figure 1.7 – The Report view

Download the Sales & Returns `sample.pbix` file from `https://docs.microsoft.com/en-us/power-bi/create-reports/sample-datasets#sales--returns-sample-pbix-file`.

How data flows in Power BI

Understanding how data flows during its journey in Power BI is important from a maintenance perspective. For instance, when you see an issue with some calculations in a report, you'll know how to do a root cause analysis and trace the issue back to an actionable point. So, if you find an issue with a figure in a line chart and that line chart is using a measure that is dependent on a calculated column, you quickly know that you won't find that calculated column in Power Query as the objects created in the data model are *not* accessible in Power Query. So, in that sense, you will never look for a measure in the Power Query layer or vice versa, as you do not expect to be able to use user-defined functions in the data model layer. We will discuss custom functions in *Chapter 3*, *Data Preparation in Power Query Editor*, *Custom Functions*:

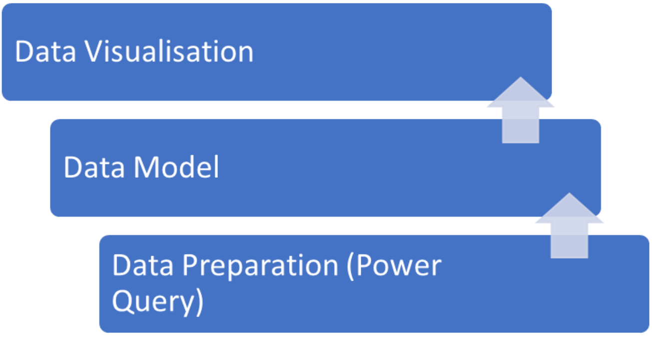


Figure 1.8 – The flow of data in Power BI

To understand this better, let's go through a scenario.

In a Power BI report, the developer has defined a **query parameter**. The parameter has a list of capital letters, `E`, `O`, and `P`. There is also a `Product` query in Power Query holding descriptive information about the product. The `Product Name` column is filtered by the parameters list. So, when the developer selects `E` from the parameter, the `Product` query filters the results showing only the products whose name starts with `E`.

You put a table visual on the report canvas with the `Product Name` column. Can you add a slicer to the report canvas showing the parameters' values so that the end user changes the values in the slicer and can see the changes in the table visual?

This is a real-world question you may get from time to time from Power BI developers. To answer the question, you need to think about Power BI layers. Let's do some analysis:

- Query parameters are defined in the data preparation layer in Power Query.
- Filtering a query is also a transformation step in Power Query, which changes the result sets of the query. Therefore, when we import the data into the data model, the result sets will *not* change unless we go back to Power Query and change the parameters' values, which consequently changes the result sets of the `Product` query and imports the new result sets to the data model.
- By default, query parameters' values are not loaded into the data model unless the developer sets **Enable load**. Setting **Enable load** only loads the selected values from the parameters list and not the whole list.
- A slicer is a visual. So, now we are talking about the data visualization layer. This means the slicer can only get values available in the data model.

So, the answer is *no*. After importing the result sets of a query to the data model, that data will be accessible to the data visualization layer.

What data modeling means in Power BI

Data modeling is undoubtedly one of the most important parts of Power BI development. The purpose of data modeling in Power BI is different from data models in transactional systems. In a transactional system, the goal is to have a model that is optimized for recording transactional data. Nevertheless, a well-designed data model in Power BI must be optimized for querying the data and reducing the dataset size by aggregating that data.

While not everyone has the luxury of having a prebuilt data warehouse, you end up creating a data model in Power BI in many cases. It is very tempting to get the whole data as is from various data sources and import it to Power BI. Answering business questions can quickly translate to complex queries that take a long time to process, which is not ideal. So, it is highly advised to resist the temptation to import everything from the data sources into Power BI and solve the problems later. Instead, it is wise to try to get your data model right so that it is capable of precisely answering business-driven questions in the most performant way. When modeling data in Power BI, you need to build a data model based on the business logic. Having said that, you may need to join different tables and aggregate your data to a certain level that answers all business-driven questions. It can get worse if you have data from various data sources of different grains representing the same logic.

Therefore, you need to transform and reshape that data before loading it to the data model in Power BI. Power Query is the right tool to take care of all that. After we cut all the noise from our data, we have a clean, easy-to-understand, and easy-to-work-with data model.

Semantic model

Power BI inherits its blood from Power Pivot and SSAS Tabular models. All of them use the **xVelocity engine**, an updated version of the VertiPaq engine designed for in-memory data analysis and consisting of semantic model objects such as tables, relationships, hierarchies, and measures, which are stored in memory, leveraging column store indexing. All of this means that you would expect to get tremendous performance gains over highly compressed data, right? Well, it depends. If you efficiently transformed the data to support business logic, then you can expect to have fast and responsive reports. After you import your data into the data model in Power BI, you have built a semantic model when it comes to the concepts. A semantic model is a unified data model that provides business contexts to data. The semantic model can be accessed from various tools to visualize data without needing it to be transformed again. In that sense, when you publish your reports to the Power BI service, you can analyze the dataset in Excel or use third-party tools such as Tableau to connect to a Power BI dataset backed by a Premium capacity and visualize your data.

Building an efficient data model in Power BI

An efficient data model can quickly answer all business questions that you are supposed to answer; it is *easy to understand* and *easy to maintain*. Let's analyze the preceding sentence. Your model must do the following:

- Perform well (quickly)
- Be business-driven
- Decrease the level of complexity (be easy to understand)
- Be maintainable with low costs

Let's look at the preceding points with a scenario.

You are tasked to create a report on top of three separate data sources, as follows:

- An OData data source with 15 tables. The tables have between 50 and 250 columns.
- An Excel file with 20 sheets that are interdependent with many formulas.

- A data warehouse hosted in SQL Server. You need to get data from five dimensions and two fact tables:

  - Of those five dimensions, one is a Date dimension and the other is a Time dimension. The grain of the Time dimension is hour, minute.
  - Each of the fact tables has between 50 and 200 million rows. The grain of both fact tables from a date and time perspective is day, hour, minute.
  - Your organization has a Power BI Pro license.

There are already a lot of important points in the preceding scenario that you must consider before starting to get the data from the data sources. There are also a lot of points that are *not* clear at this stage. I have pointed out some of them:

- **OData**: OData is an online data source, so it could be slow to load the data from the source system.

  The tables are very wide so it can potentially impact the performance.

  Our organization has a Power BI Pro license, so we are limited to 1 GB file size.

  The following questions should be answered by the business. Without knowing the answers, we may end up building a report that is unnecessarily large with poor performance. This can potentially lead to the customer's dissatisfaction:

  (a) Do we really need to import all the columns from those 15 tables?

  (b) Do we also need to import all data or is just a portion of the data enough? In other words, if there is 10-years worth of data kept in the source system, does the business need to analyze all the data, or does just 1 or 2 years of data fit the purpose?

- **Excel**: Generally, Excel workbooks with many formulas can be quite hard to maintain. So, we need to ask the following questions of the business:

  (a) How many of those 20 sheets of data are going to be needed by the business? You may be able to exclude some of those sheets.

  (b) How often are the formulas in the Excel file edited? This is a critical point as modifying the formulas can easily break your data processing in Power Query and generate errors. So, you would need to be prepared to replicate a lot of formulas in Power BI if needed.

- **Data warehouse in SQL Server**: It is beneficial to have a data warehouse as a source system as data warehouses normally have a much better structure from an analytical viewpoint. But in our scenario, the finest grain of both fact tables is down to a minute. This can potentially turn into an issue pretty quickly. Remember, we have a Power BI Pro license, so we are limited to a 1 GB file size only. Therefore, it is wise to clarify some questions with the business before we start building the model:

  (a) Does the business need to analyze all the metrics down to the minute or is day-level enough?

  (b) Do we need to load all the data into Power BI, or is a portion of the data enough?

  We now know the questions to ask, but what if the business needs to analyze the whole history of the data? In that case, we may consider using composite models with aggregations.

  Having said all that, there is another point to consider. We already have five dimensions in the data warehouse. Those dimensions can potentially be reused in our data model. So, it is wise to look at the other data sources and find commonalities in the data patterns.

You may come up with some more legitimate points and questions. However, you can quickly see in the previously mentioned points and questions that you need to *talk to the business* and *ask questions* before starting the job. It is a big mistake to start getting data from the source systems before framing your questions around business processes, requirements, and technology limitations. There are also some other points that you need to think about from a project management perspective that are beyond the scope of this book.

The initial points to take into account for building an efficient data model are as follows:

- We need to ask questions of the business to avoid any confusions and potential reworks in the future.
- We need to understand the technology limitations and come up with solutions.
- We have to have a good understanding of data modeling, so we can look for common data patterns to prevent overlaps.

At this point, you may think, "OK, but how we can get there?" My answer is that you have already taken the first step, which is reading this book. All the preceding points and more are covered in this book. The rest is all about you and how you apply your learning to your day-to-day challenges.

Star schema (dimensional modeling) and snowflaking

First things first, **star schema** and **dimensional modeling** are the same things. In Power BI data modeling, the term star schema is more commonly used. So, in this book, we will use the term star schema. The intention of this book is not to teach you about dimensional modeling. Nevertheless, we'll focus on how to model data using star schema data modeling techniques. We will remind you of some star schema concepts.

Transactional modeling versus star schema modeling

In transactional systems, the main goal is to improve the system's performance in creating new records and updating/deleting existing records. So, it is essential to go through the normalization process to decrease data redundancy and increase data entry performance when designing transactional systems. In a straightforward form of normalization, we break tables down into master-detail tables.

But the goal of a business analysis system is very different. In business analysis, we need a data model optimized for querying in the most performant way.

Let's continue with a scenario.

Say we have a transactional retail system for international retail stores. We have hundreds of transactions every second from different parts of the world. The company owner wants to see the total sales amount in the past 6 months.

This calculation sounds easy. It is just a simple `SUM` of sales. But wait, we have hundreds of transactions every second, right? If we have only 100 transactions per second, then we have 8,640,000 transactions a day. So, for 6 months of data, we have more than 1.5 billion rows. Therefore, a simple `SUM` of sales will take a reasonable amount of time to process.

The scenario continues: Now, a new query comes from the business. The company owner now wants to see the total sales amount in the past 6 months by country and city. He simply wants to know what the best-selling cities are.

We need to add another condition to our simple `SUM` calculation, which translates to a *join* to the geography table. For those coming from a relational database design background, it will be obvious that joins are relatively expensive operations. This scenario can go on and on. So, you can imagine how quickly you will face a severe issue.

In the star schema, however, we already joined all those tables based on business entities. We aggregated and loaded the data into denormalized tables. In the preceding scenario, the business is not interested in seeing every transaction at the second level. We can summarize the data at the *day* level, which decreases the number of rows from 1.5 billion to a couple of thousands of rows for the whole 6 months. Now you can imagine how fast the summation would be running over thousands of rows instead of 1.5 billion rows.

The idea of the star schema is to keep all numeric values in separate tables called fact tables and put all descriptive information into other tables called **dimension tables**. Usually, the fact tables are surrounded by dimensions that explain those facts. When you look at a data model with a fact table in the middle surrounded by dimensions, you see that it looks like a star. Therefore, it is called a star schema. In this book, we generally use the `Adventure Works DW` data, a renowned Microsoft sample dataset, unless stated otherwise. Adventure Works is an international bike shop that sells products both online and in their retail shops. The following figure shows **Internet Sales** in a star schema shape:

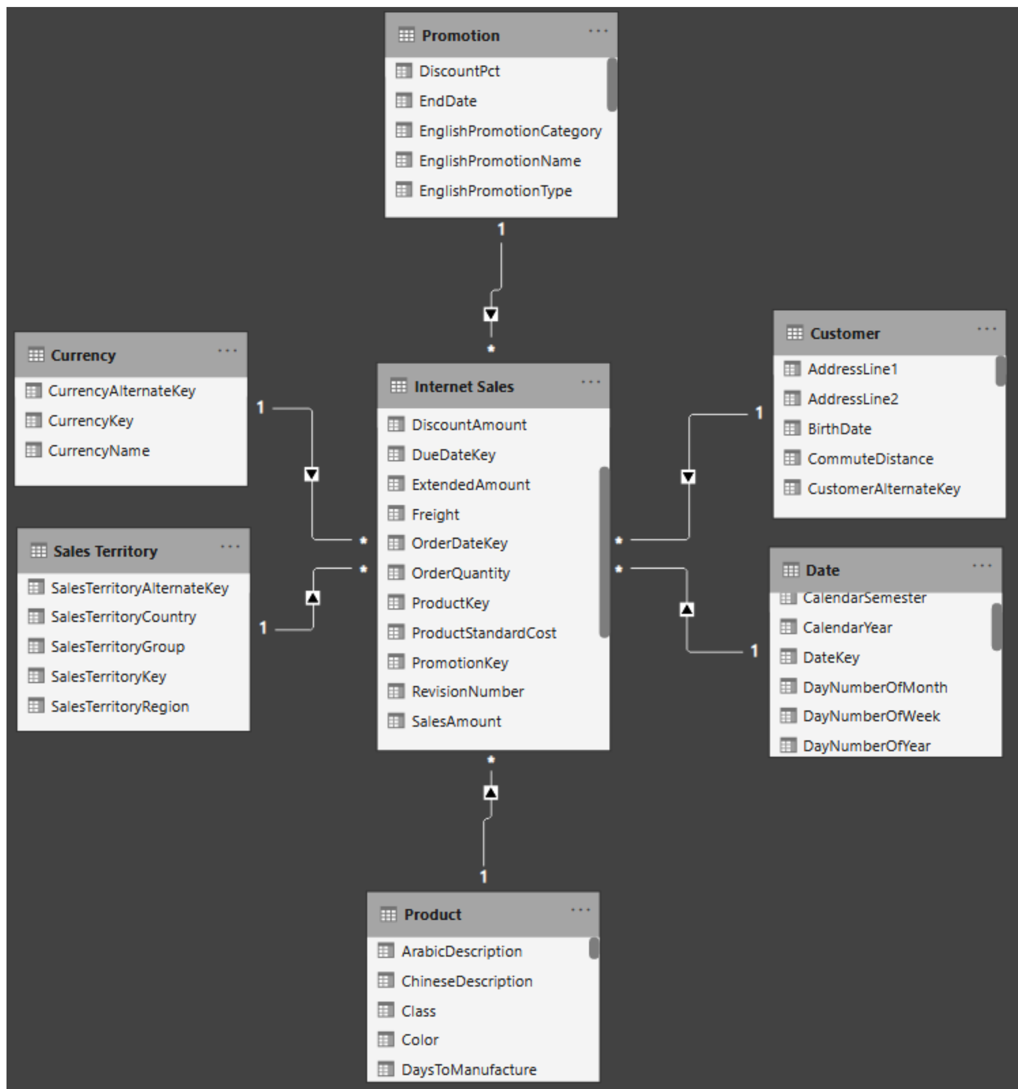


Figure 1.9 – Adventure Works DW, Internet Sales star schema

Snowflaking

Snowflaking is when you do not have a perfect star schema when dimension tables surround your fact tables. In some cases, you have some levels of descriptions stored in different tables. Therefore, some dimensions of your model are linked to some other tables that describe the dimensions in a greater level of detail. Snowflaking is normalizing the dimension tables. In some cases, snowflaking is inevitable; nevertheless, the general rule of thumb in data modeling in Power BI (when following a star schema) is to avoid snowflaking as much as possible. The following figure shows snowflaking in Adventure Works **Internet Sales**:

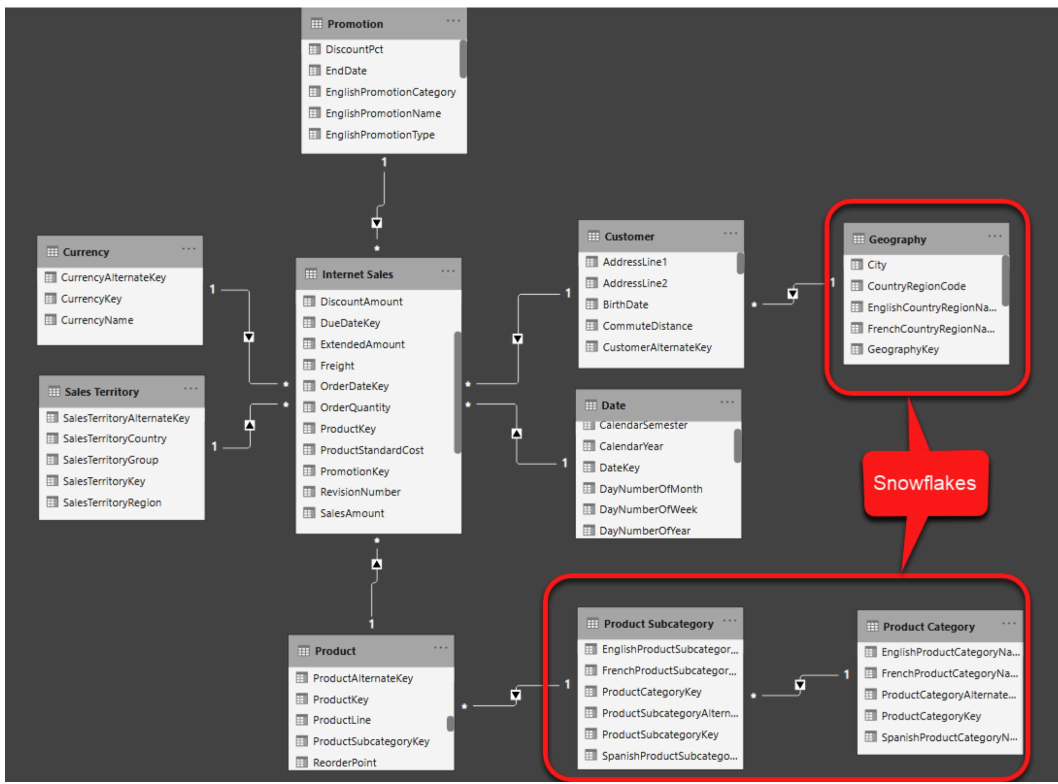


Figure 1.10 – Adventure Works, Internet Sales snowflakes

Understanding denormalization

In real-world scenarios, not everyone has the luxury of having a pre-built data warehouse designed in a star schema. In reality, snowflaking in data warehouse design is inevitable. So, you may build your data model on top of various data sources, including transactional database systems and non-transactional data sources such as Excel files and CSV files. So, you almost always need to denormalize your model to a certain degree. Depending on the business requirements, you may end up having some level of normalization along with some denormalization. The reality is that there is no specific rule for the level of normalization and denormalization. The general rule of thumb is to denormalize your model so that a dimension can describe all the details as much as possible.

In the preceding example from `Adventure Works DW`, we have snowflakes of `Product Category` and `Product Subcategory` that can be simply denormalized into the `Product` dimension.

Let's go through a hands-on exercise.

Go through the following steps to denormalize `Product Category` and `Product Subcategory` into the `Product` dimension.

> **Note**
>
> You can download the `Adventure Works, Internet Sales.pbix` file from here:
>
> `https://github.com/PacktPublishing/Expert-Data-Modeling-with-Power-BI/blob/master/Adventure%20Works%20DW.pbix`

Open the `Adventure Works, Internet Sales.pbix` file and follow these steps:

1. Click **Transform data** on the **Home** tab in the **Queries** section.
2. Click the **Product** Query.
3. Click **Merge Queries** on the **Home** tab in the **Combine** section.
4. Select **Product Subcategory** from the drop-down list.
5. Click **ProductSubcategoryKey** on the **Product** table.
6. Click **ProductSubcategoryKey** on the **Product Subcategory** table.

7. Select **Left Outer (all from first matching from the second)** from the **Join Kind** dropdown.
8. Click **OK**:

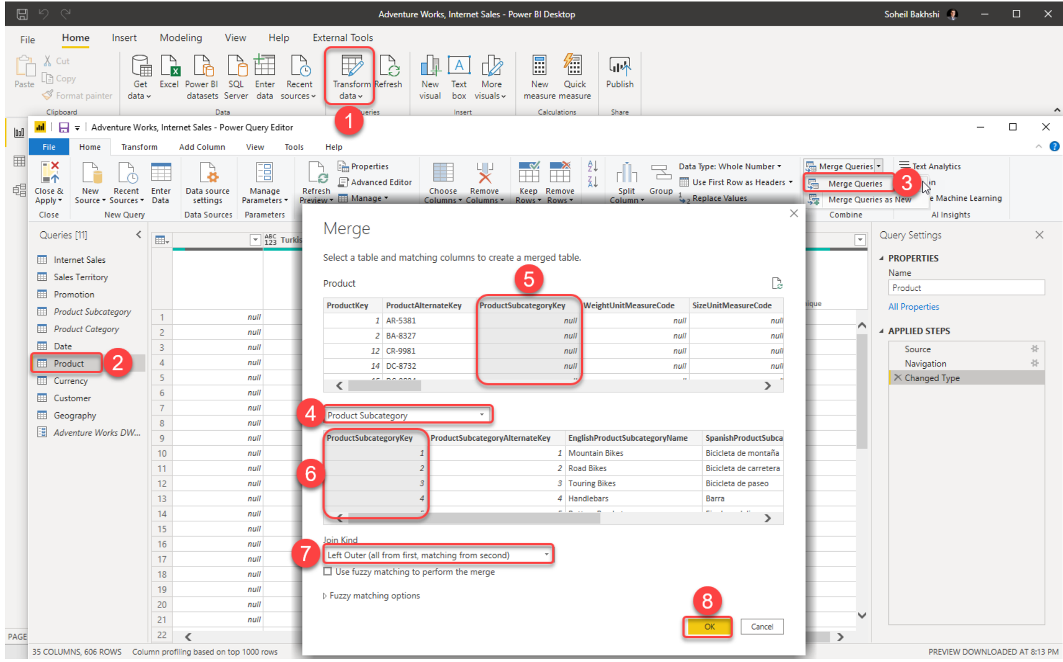


Figure 1.11 – Merging Product and Product Subcategory

This adds a new step named **Merged Queries**. As you see, the values of this column are all **Table**. This type of column is called **Structured Column**. The merging **Product** and **Product Subcategory** step creates a new structured column named **Product Subcategory**:

You will learn more about structured columns in *Chapter 3*, *Data Preparation in Power Query Editor*.

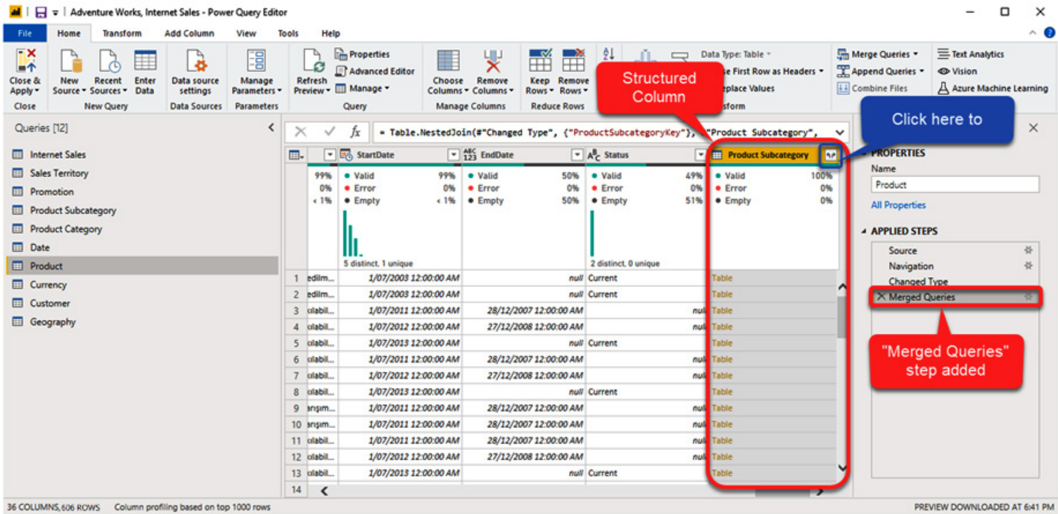


Figure 1.12 – Merging the Product and Product Subcategory tables

Now let's look at how to expand a structured column in the query editor:

1. Click the **Expand** button to expand the **Product Subcategory** column.
2. Select **ProductCategoryKey**.
3. Select the **EnglishProductSubcategoryName** columns and unselect the rest.
4. Unselect **Use original column names as prefix**.
5. Click **OK**:

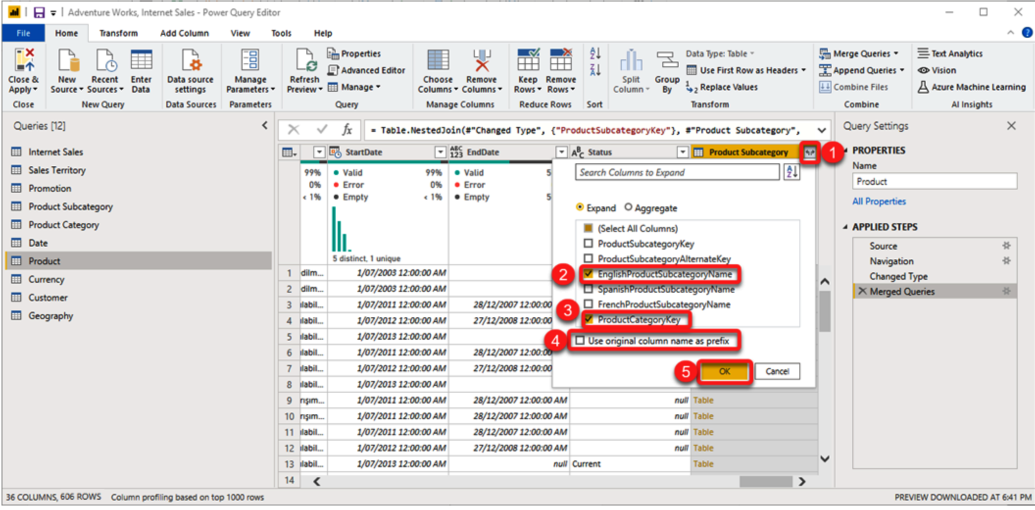


Figure 1.13 – Expanding Structured Column in the Query Editor

So far, we have added the `EnglishProductSubcategoryName` and `ProductCategoryKey` columns from the `Product Subcategory` query to the `Product` query. The next step is to add `EnglishProductCategoryName` from the `Product Category` query. To do so, we need to merge the `Product` query with `Product Category`:

1. Click **Merge Queries** again.
2. Select **Product Category** from the drop-down list.
3. Select **ProductCategoryKey** from the **Product** table.
4. Select **ProductCategoryKey** from the **Product Category** table.
5. Select **Left Outer (all from first matching from second)**.
6. Click **OK**:

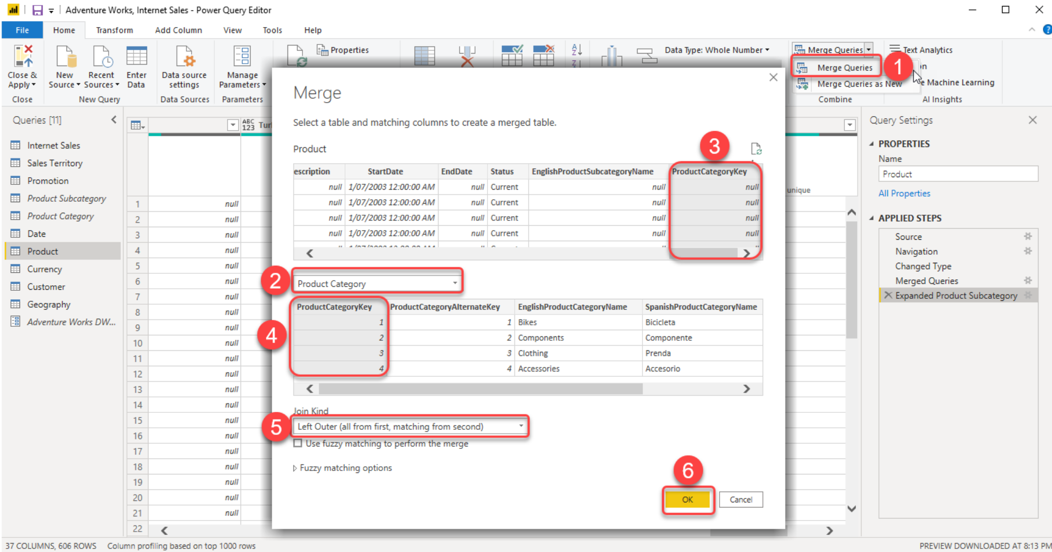


Figure 1.14 – Merging Product and Product Category

This adds another step and a new structured column named `Product Category`. We now need to do the following:

1. Expand the new column.
2. Pick **EnglishProductCategoryName** from the list.
3. Unselect **Use original column name as prefix**.
4. Click **OK**:

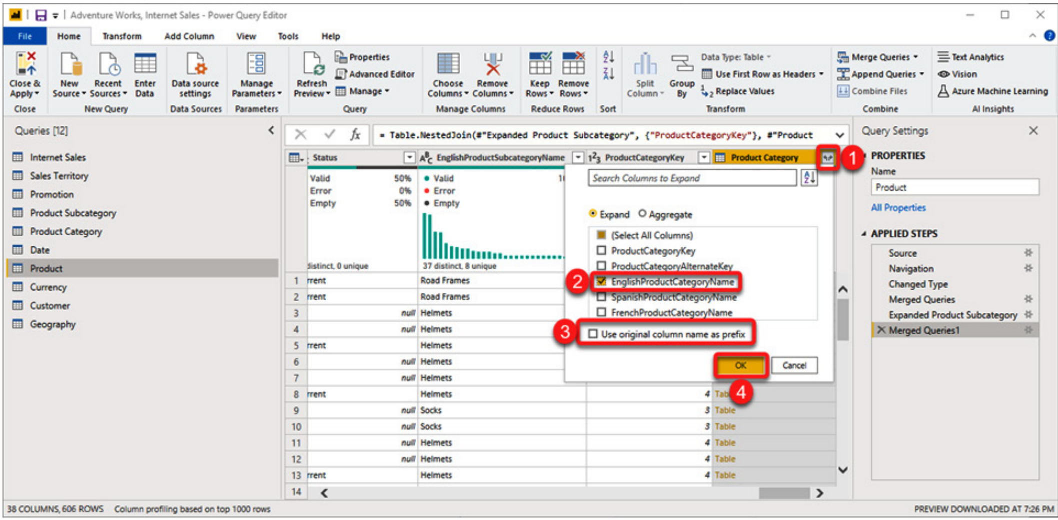


Figure 1.15 – Merging Product and Product Category

The next step is to remove the `ProductCategoryKey` column as we do not need it anymore. To do so, do the following:

1. Click on the **ProductCategoryKey** column.
2. Click the **Remove Columns** button in the **Managed Column** section of the **Home** tab:

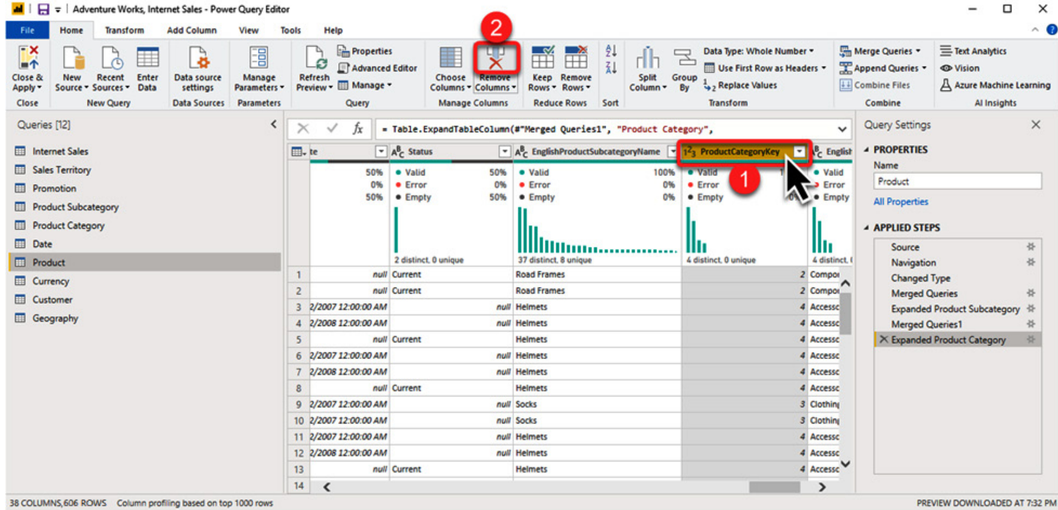


Figure 1.16 – Removing a column in the Query Editor

Now we have merged the `Product Category` and `Product Subcategory` snowflakes with the `Product` query. So, you have denormalized the snowflakes.

The very last step is to unload both the `Product Category` and `Product Subcategory` queries:

1. Right-click on each query.
2. Untick **Enable load** from the menu.
3. Click **Continue** on the **Possible Data Loss Warning** pop-up message:

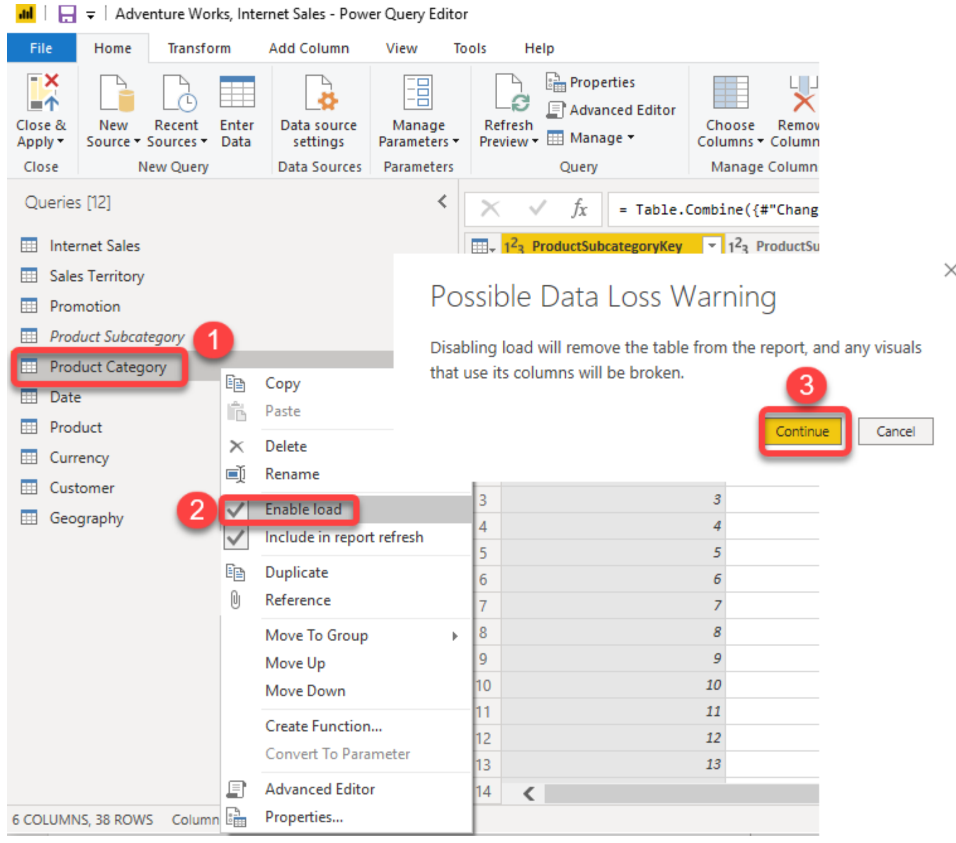


Figure 1.17 – Unloading queries in the Query Editor

Now we need to import the data into the data model by clicking **Close & Apply**:

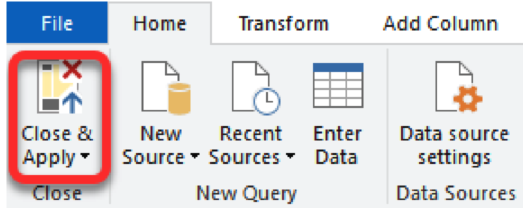


Figure 1.18 – Importing data into the data model

We have now achieved what we were after: we denormalized the `Product Category` and `Product Subcategory` tables, therefore rather than loading those two tables, we now have `EnglishProductCategoryName` and `EnglishProductSubcategoryName` represented as new columns in the `Product` table.

Job done!

Power BI licensing considerations

At this point, you may be wondering how Power BI licensing affects data modeling. It does, as each licensing tier comes with a set of features that can potentially affect the data modeling. Nevertheless, regardless of the licensing tier you are using, Power BI Desktop is free of charge. In this section, we'll quickly look at some licensing considerations related to data modeling.

The following table is a simplified version of the Power BI feature comparisons published on the Microsoft website separately based on different licenses:

Power BI License	Maximum Size of Individual Dataset	Incremental Data Load	Calculation Groups	Shared Datasets	Power BI Dataflows
Free	1	No	No	No	No
Professional	1	Yes	No	Yes	Yes
Power BI Report Server	2	Yes	No	N/A	Yes
EM1/A1	3	Yes	Yes	Yes	Yes
EM2/A2	5	Yes	Yes	Yes	Yes
EM3/A3	10	Yes	Yes	Yes	Yes
P1/A4	25	Yes	Yes	Yes	Yes
P2/A5	50	Yes	Yes	Yes	Yes
P3/A6	100	Yes	Yes	Yes	Yes
P4	200	Yes	Yes	Yes	Yes
P5	400	Yes	Yes	Yes	Yes

Figure 1.19 – A simplified version of Power BI feature comparisons

Maximum size of individual dataset

As the table illustrates, we are limited to 1 GB for each dataset published to the Power BI service under Free or Professional licensing. Therefore, managing the file size is quite important. There are several ways to keep the file size just below the limit, as follows:

- Import the necessary columns only.
- Import just a portion of data when possible. Explain the technology limitation to the business and ask whether you can filter out some data. For instance, the business may not need to analyze 10 years of data, so filter older data in Power Query.
- Use aggregations. In many cases, you may have the data stored in the source at a very low granularity. However, the business requires data analysis on a higher grain. Therefore, you can aggregate the data to a higher granularity, then import it into the data model. For instance, you may have data stored at a minute level. At the same time, the business only needs to analyze that data at the day level.

- Consider disabling auto date/time settings in Power BI Desktop.
- Consider optimizing data types.

We will cover all the preceding points in the upcoming chapters.

Incremental data load

One of the coolest features available in Power BI is the ability to set up an incremental data load. Incremental data loading in Power BI is inherited from SSAS to work with large models. When it is set up correctly, Power BI does not truncate the dataset and re-import all the data from scratch. Instead, it only imports the data that has been changed since the last data refresh. Therefore, incremental data load can significantly improve the data refresh performance and decrease the amount of processing load on your tenant. Incremental data load is available in both Professional and Premium licenses.

Calculation groups

Calculation groups are like *calculated members* in **MultiDimensional eXpressions** (**MDX**). Calculation groups were initially introduced in SSAS 2019 Tabular models. They are also available in Azure Analysis Services and all Power BI licensing tiers.

It is a common scenario that you create (or already have) some base measures in your Power BI model and then create many time intelligence measures on top of those base measures. In our sample file, we have three measures, as follows:

- Product cost: `SUM('Internet Sales'[TotalProductCost])`
- Order quantity: `SUM('Internet Sales'[OrderQuantity])`
- Internet sales: `SUM('Internet Sales'[SalesAmount])`

The business requires the following time intelligence calculations on top of all the preceding measures:

- Year to date
- Quarter to date
- Month to date
- Last year to date
- Last quarter to date
- Last month to date

- Year over year
- Quarter over quarter
- Month over month

We have nine calculations to be built on top of every single measure we have in our model. Hence, we end up having *9 x 3 = 27* measures to build in our model. You can imagine how quickly the number of measures can rise in the model, so you should not be surprised if someone tells you that they have hundreds of measures in their Power BI model.

Another common scenario is when we have multiple currencies. Without calculation groups, you need to convert the values into strings to show the figures and use a relevant currency symbol using the `FORMAT()` function in DAX. Now, if you think about the latter point, combined with time intelligence functions, you can see how the issue can get bigger and bigger.

Calculation groups solve those sorts of problems. We cover calculation groups in *Chapter 10*, *Advanced Data Modeling Techniques*.

Shared datasets

As the name implies, a shared dataset is a dataset used across various reports in a *modern workspace* (a new workspace experience) within the Power BI service. Therefore, it is only available in the Power BI Professional and Power BI Premium licensing plans. This feature is quite crucial to data modelers. It provides more flexibility in creating a more generic dataset, covering more business entities in a single dataset instead of having several datasets that may share many commonalities.

Power BI Dataflows

Dataflows, also referred to as Power Query Online, provide a centralized data preparation mechanism in the Power BI service that other people across the organization can take advantage of. Like using Power Query in Power BI Desktop for data preparation, we can prepare, clean, and transform the data in dataflows. Unlike Power Query queries, which are isolated within a dataset, when created in Power BI Desktop and then published to the Power BI service, you can share all data preparations, data cleansing, and data transformation processes across the organization with dataflows.

You can create Power BI dataflows inside a workspace, so it is only available to Professional and Premium users. We will also cover Power BI dataflows in future chapters.

The iterative data modeling approach

Like many other software development approaches, data modeling is an ongoing process. You start talking to the business, then apply the business logic to your model. You carry on with the rest of your Power BI development. In many cases, you build your data visualizations and then find out that you will get better results if you make some changes in your model. In many other cases, the business logic applied to the model is not what the business needs. This is a typical comment that many of us will get from the business after the first few iterations:

This looks really nice, but unfortunately, it is not what we want.

So, taking advantage of an agile approach would be genuinely beneficial for Power BI development. Here is the iterative approach you can follow in your Power BI development:

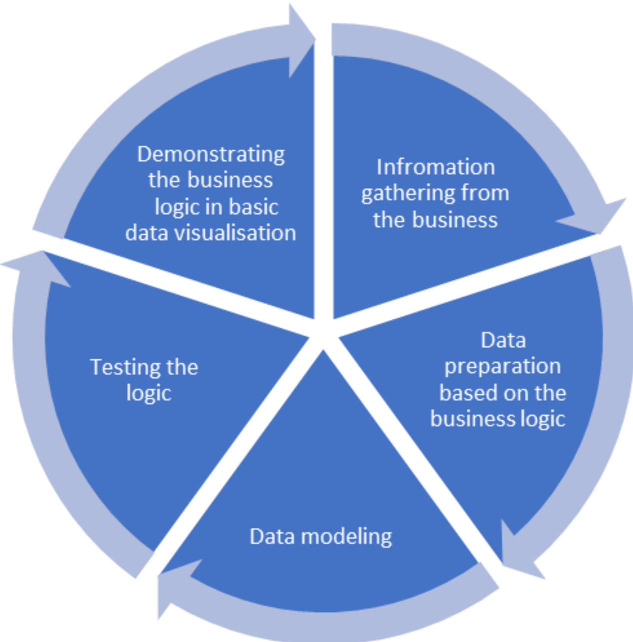


Figure 1.20 – The iterative data modeling approach

Information gathering from the business

Like all other software development processes, a Power BI development process starts with gathering information from the business to get a better understanding of the business requirements. A business analyst may take care of this step in the real world but wait, a lot of Power BI users are business analysts. Regardless of your role, whether you are a business analyst or a business analyst takes care of this step and you are a data modeler, you need to analyze the information you get from the business. You have to ask relevant questions and come up with a list of design possibilities. You have to identify potential risks and discuss them with the customer. You also need to be aware of technology limitations and discuss them with the customer as well. After you get answers to your questions and have a list of design possibilities, risks, and technology limitations, you can move on to the next step more confidently.

Data preparation based on the business logic

You now have a lot on your plate. You need to get the data from various data sources and go through the data preparation steps. Now that you know a lot about business logic, you can take the proper steps in your data preparation. For instance, if the business requires you to connect to an OData data source and get a list of the columns required by the business, you can prepare your data more efficiently with all the design risks and technology limitations in mind. After you have consciously prepared your data, you will go on to the next step, which is data modeling.

Data modeling

If you took the proper actions in the previous steps, your data model will be much tidier, so you can build your model more efficiently. Now you need to think about the analytical side of things. Simultaneously, you still have all the business requirements, design possibilities, risks, and technology limitations in mind. For instance, if the business cannot tolerate data latency longer than 5 minutes, you may need to think about using DirectQuery. Using DirectQuery comes with some limitations and performance risks. So, you need to think about the design approach that satisfies the business requirements the most. We cover DirectQuery in *Chapter 4, Getting Data from Various Sources* in the *Dataset storage modes* section.

Testing the logic

This is one of the most trivial and yet most important steps in data modeling: testing all the business logic you implement to meet the requirements. Not only do you need to test the figures to make sure the results are accurate, but you also need to test the solution from a performance and user experience perspective. Be prepared for tons of mixed feedback, and sometimes strong criticism from the end users, especially when you think everything is OK.

Demonstrating the business logic in a basic data visualization

As we are modeling the data, we do not need to be worried about the data visualization part. The fastest way to make sure all the business logic is right is to confirm with the business. The fastest way to do that is to demonstrate the logic in the simplest possible way, such as using table and matrix visuals and some slicers on the page. Remember, this is only to confirm the logic with the business, not the actual product delivery. There will be a lot of new information and surprises that come up during the demonstration in the real world, which means you'll then need to start the second iteration and gather more information from the business.

As you go through all the preceding steps several times, you'll gradually become a professional data modeler. In the next section, we'll quickly cover how professional data modelers think.

> **Note**
>
> This book also follows an iterative approach, so we'll go back and forth between different chapters to cover some scenarios.

Thinking like a professional data modeler

Back in the day, in the late 90s, I was working on transactional database systems. Back then, it was essential to know how to normalize your data model to at least the **third normal form**. In some cases, we were normalizing to the **Boyce-Codd normal form**. I carried out many projects facing a lot of different issues and I made many mistakes, but I learned from those mistakes. Gradually, I was experienced enough to visualize the data model to the second or sometimes even to the third normal form in my head while I was in a requirements gathering session with the customer. All data modeling approaches that I had a chance to work with, or read about, were based on relational models regardless of their usage, such as transactional models, star schema, Inmon, and data vault. They are all based on relational data modeling. Data modeling in Power BI is no different. Professional data modelers can visualize the data model in their minds from the first information-gathering sessions they have with the customer. But as mentioned, this capability comes with experience.

Once you have enough experience in data modeling, you'll be able to ask more relevant questions from the business. You already know of some common scenarios and pitfalls, so you can quickly recognize other similar situations. Therefore, you can avoid many future changes by asking more relevant questions. Moreover, you can also give your customer some new ideas to solve other problems down the road. In many cases, the customer's requirements will change during the project lifetime. So, you will not be surprised when those changes happen.

Summary

In this chapter, we discussed the different layers of Power BI and what is accessible in which layer. Therefore, when we face an issue, we know exactly where we should look to fix the problem. Then we discussed how when we build a data model, we are indeed making a semantic layer in Power BI. We also covered some star schema and snowflaking concepts, which are essential to model our data more efficiently. We then covered different Power BI licensing considerations and how they can potentially affect our data modeling. Lastly, we looked at the data modeling iterative approach to deliver a more precise and more reliable data model that solves many problems that the report writers may face down the road.

In the next chapter, we will look at DAX and data modeling. We will discuss a somewhat confusing topic, virtual tables, and we will walk you through some common time intelligence scenarios to help you with your future data modeling tasks.

2
Data Analysis eXpressions and Data Modeling

We covered in the previous chapter that Power BI has different layers: data preparation, data modeling, and data visualization. This chapter discusses **Data Analysis eXpressions** (**DAX**), and it relates to data modeling. Although data modeling and DAX are connected such that you cannot imagine one without the other, our goal in this book is not to focus only on DAX. Data modeling encompasses much broader concepts, while DAX is the expression language that developers must use to implement business logic in the data model. Our assumption in this book is that you have basic to intermediate knowledge of DAX; therefore, we will not cover basic DAX concepts. Instead, we will focus on more advanced DAX concepts with hands-on scenarios.

This chapter covers the following topics:

- Understanding virtual tables
- Relationships in virtual tables
- Time intelligence

Understanding virtual tables

The concept of **virtual tables** in DAX is somewhat confusing and misunderstood, and yet is one of the most powerful and important concepts of DAX. When we talk about virtual tables, we are referring to in-memory tables that we build using certain DAX functions or constructors. The data in a virtual table is either derived from the data within the data model or the data that we construct for specific purposes.

Remember, whenever we use a DAX function that results in a table of values, we are creating a virtual table.

At this point, you may ask, *so when I use a DAX function to create a calculated table, am I creating a virtual table?* The answer is *it depends*. If you simply use a set of DAX functions that generate data or selectively load data from other tables into a calculated table, the answer is *no*: you have not created any virtual tables. Nevertheless, suppose you generate or load the data from other tables. In that case, you do some table operations with the data and load the results into a calculated table. Most probably, you created a virtual table and populated a calculated table with the results. Virtual tables, as the name implies, are not physically stored in the model. Therefore, we cannot see them, but they exist in memory as we create them within our calculations. Hence, they are only accessible within that calculation and not from other calculations or any other parts of our data model. If you are coming from a SQL development background, then you can think of DAX virtual tables as subqueries in SQL.

Is it still confusing? Let's continue with some hands-on scenarios.

Creating a calculated table

We will create a calculated table and name it **Sequential Numbers** with a column named **ID**. The **ID** column values are sequential numbers between 1 and 20, increasing by one step in each row:

1. Open a new Power BI Desktop instance.
2. Click **New table** from the **Modeling** tab:

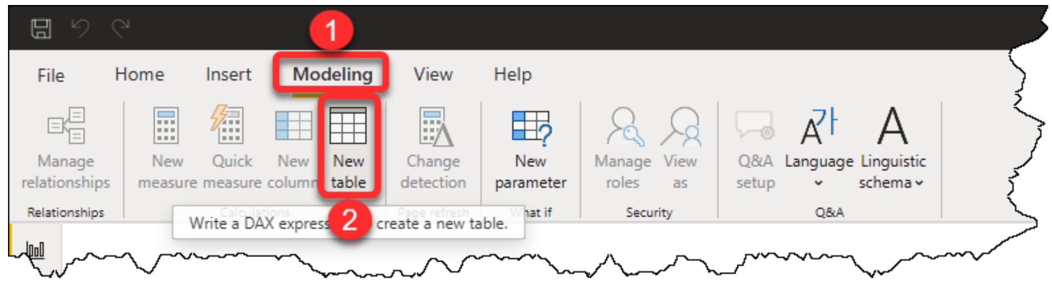


Figure 2.1 – Creating a calculated table in Power BI

3. Type the following DAX expression, then press *Enter*:

```
Sequential Numbers = GENERATESERIES(1, 20, 1)
```

This creates a calculated table named **Sequential Numbers**, as illustrated here, but the column name is **Value**, not **ID**:

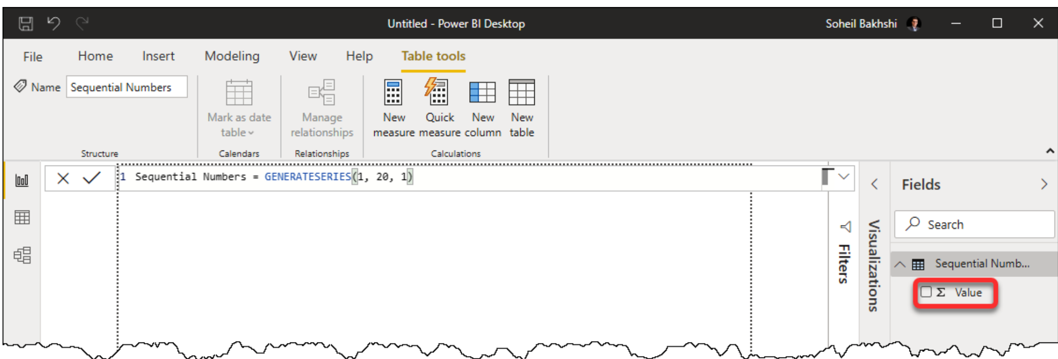


Figure 2.2 – Using the GENERATESERIES() function to create a calculated table

The `GENRATESERIES()` function generates values for us. The output is a desirable table, but we need to do one last operation to rename the `Values` column to `ID`.

4. Replace the previous expression with this expression, then press *Enter*:

```
Sequential Numbers =
SELECTCOLUMNS(
    GENERATESERIES(1, 20, 1)
    , "ID"
    , [Value]
    )
```

The following figure shows the results of the preceding calculation:

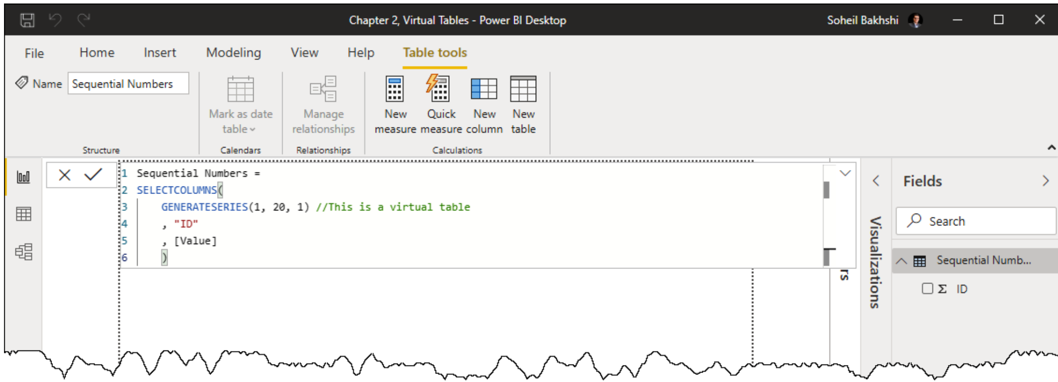


Figure 2.3 – Calculated table and virtual table

What we have done is that we first created a virtual table with a `Values` column. We renamed that column to `ID`, and finally, we populated a calculated table with the results.

In this scenario, we looked at the usage of virtual tables in calculated tables. In the following scenario, we demonstrate the usage of virtual tables in a measure.

Let's take a step further with a more complex scenario.

Using virtual tables in a measure – Part 1

In the `Adventure Works, Internet Sales.pbix` sample, create a measure in the `Internet Sales` table to calculate the ordered quantities for products when their list price is higher than $1,000. Name the measure `Orders with List Price Bigger than or Equal to $1,000`:

> **Good practice**
>
> Always create a new measure by right-clicking on the desired table from the **Fields** pane and clicking **New measure**. This way, you are always sure that you create the measure in a specific desired table. If you use the **New measure** button from the **Home** tab from the ribbon, the new measure will be created in a table that you previously focused on. Suppose there is no table selected (focused on). In that case, the new measure will be created in the first table available in your data model, which is not ideal.

1. Right-click on the `Internet Sales` table.
2. Click **New measure**.

3. Type the following DAX expression and press *Enter*:

```
Orders with List Price Bigger than or Equal to $1,000 =
CALCULATE(
    SUM('Internet Sales'[OrderQuantity])
        , FILTER('Product'                  //Virtual table start
            , 'Product'[List Price]>=1000
                )                           //Virtual table end
        )
```

The following figure shows the preceding expression in Power BI Desktop:

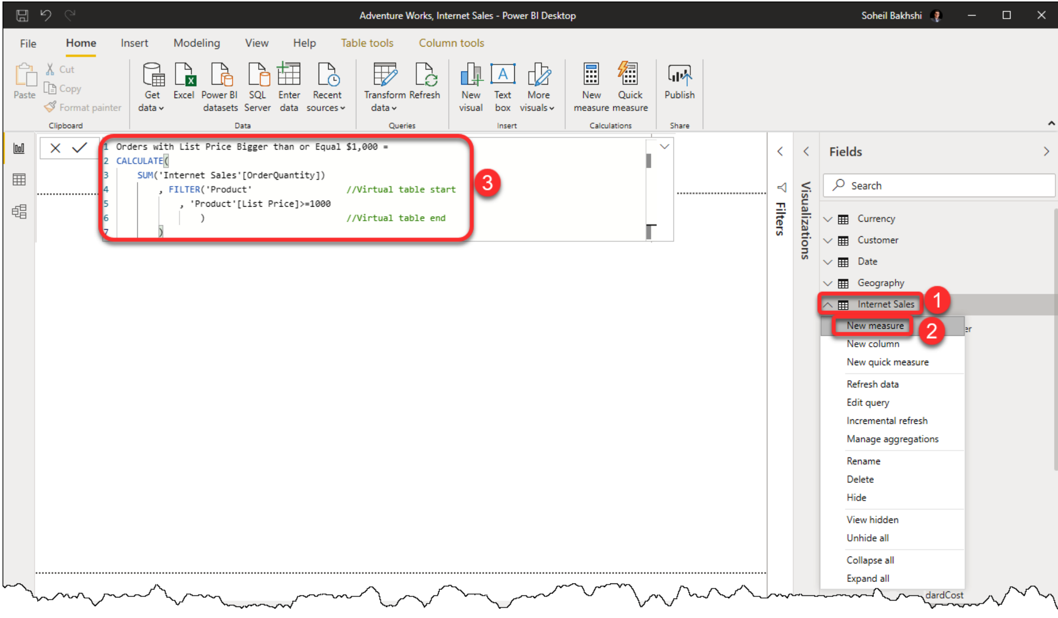


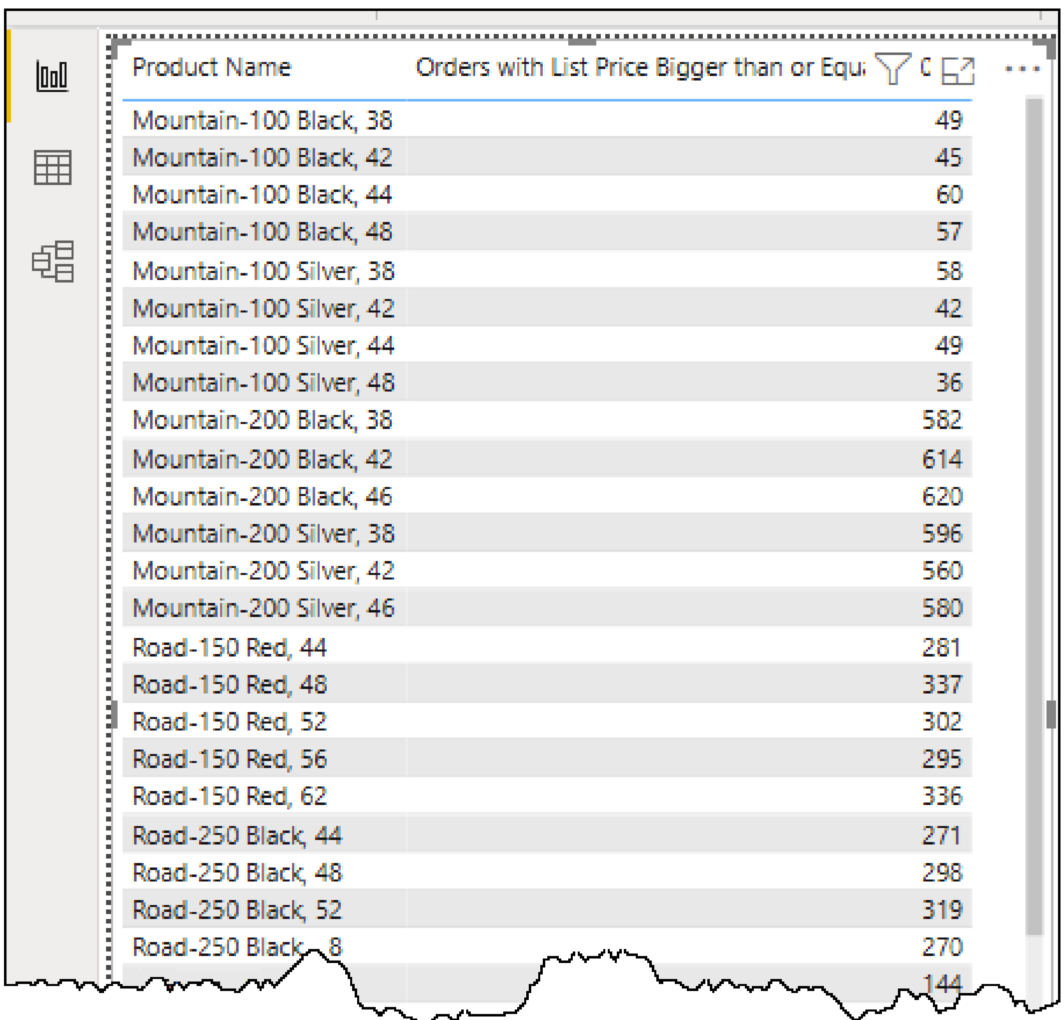
Figure 2.4 – Calculating orders with a list price bigger than or equal to $1,000

Let's analyze the preceding calculation:

- We created a virtual table using the `FILTER()` function on top of the `Product` table to get only the products with a `List Price` value bigger than or equal to $1,000. All columns from the `Product` tables are available in this virtual table, which lives in memory. It is only available within the `Orders with List Price Bigger than or Equal to $1,000` measure and nowhere else in the data model.

- The `SUM()` function then calculates the summation of the `OrderQuantity` from the `Internet Sales` table.

To use the preceding measure, put a table visual on a report page. Select `Product Name` from the `Product` table, then select the `Orders with List Price Bigger, or Equal to $1,000` measure. The result looks like the following figure:



Product Name	Orders with List Price Bigger than or Equ
Mountain-100 Black, 38	49
Mountain-100 Black, 42	45
Mountain-100 Black, 44	60
Mountain-100 Black, 48	57
Mountain-100 Silver, 38	58
Mountain-100 Silver, 42	42
Mountain-100 Silver, 44	49
Mountain-100 Silver, 48	36
Mountain-200 Black, 38	582
Mountain-200 Black, 42	614
Mountain-200 Black, 46	620
Mountain-200 Silver, 38	596
Mountain-200 Silver, 42	560
Mountain-200 Silver, 46	580
Road-150 Red, 44	281
Road-150 Red, 48	337
Road-150 Red, 52	302
Road-150 Red, 56	295
Road-150 Red, 62	336
Road-250 Black, 44	271
Road-250 Black, 48	298
Road-250 Black, 52	319
Road-250 Black, 8	270
[illegible]	144

Figure 2.5 – Orders with a list price bigger than or equal to $1,000 by product

Let's take another step forward and look at a more complex scenario.

Using virtual tables in a measure – Part 2

In the `Adventure Works, Internet Sales.pbix` sample, create a measure in the `Internet Sales` table to calculate quantities that the customers ordered more than 4 products with a list price bigger than $1,000. Name the measure `Order Qty for Customers Buying More than 4 Products with List Price Bigger Than $1,000`.

To solve this scenario, we need to create a virtual table of customers with more than 4 orders for products that cost more than $1,000. The following code will take care of that:

```
 Order Qty for Customers Buying More than 4 Product with List
Price Bigger Than $1,000 =
SUMX(
    FILTER(
        VALUES(Customer[CustomerKey]) //Virtual table
        , [Orders with List Price Bigger than or Equal $1,000]
> 4
        )
    , [Orders with List Price Bigger than or Equal $1,000]
    )
```

Analyzing the preceding calculation helps us to understand the power of virtual tables much better:

- The virtual table here has only one column, which is `CustomerKey`. Remember, this column is only accessible within the current calculation.
- Then, we use the `FILTER()` function to filter the results of `VALUES()` only to show the customer keys that have more than 4 orders for products with a list price of more than $1,000.
- Last, we sum all those quantities for the results of `FILTER()`.

We can now put a table visual on the report page, then select `First Name` and `Last Name` from the `Customer` table and the new measure we just created. The following figure shows the customers who ordered more than 4 items that cost more than $1,000:

First Name	Last Name	Order Qty for Customers Buying More than 4 Product with List Price Bigger Than $1,000
Adriana	Gonzalez	5
Brad	She	5
Brandi	Gill	5
Francisco	Sara	5
Janet	Munoz	5
Kaitlyn	Henderson	5
Margaret	He	5
Maurice	Shan	6
Nichole	Nara	5
Randall	Dominguez	5
Rosa	Hu	5
Total		**56**

Figure 2.6 – The Order Qty for Customers Buying More than 4 Product with List Price Bigger Than $1,000 by Customer table

As stated earlier, we can use all DAX functions that return a table value to create virtual tables. However, the functions in the following table are the most common ones:

ADDCOLUMNS()	CALENDARAUTO()	FILTERS()	SELECTCOLUMNS()	VALUES()
ADDMISSINGITEMS()	CALCULATETABLE()	GENERATESERIES()	SUMMARIZE()	Table Constructor {}
ALL()	CROSSJOIN()	INTERSECT()	SUMMARIZECOLUMNS()	
ALLEXCEPT()	DATATABLE()	NATURALINNERJOIN()	TOPN()	
ALLSELECTED()	DISTINCT()	NATURALLEFTOUTERJOIN()	TREATAS()	
CALENDAR()	EXCEPT()	RELATEDTABLE()	UNION()	

Figure 2.7 – Commonly used functions

The virtual tables are only available in memory. Moreover, they are not visible in the model view in Power BI Desktop. Therefore, they are a bit hard to understand. However, there are still some ways that you can test the virtual tables and visually see the results, which we will cover in the next section.

Visually displaying the results of virtual tables

In this section, we explain how you can see the results of **virtual tables**. It is essential to see the results of a virtual table rather than only internalizing the results in our minds. In this section, we look at two different ways to see a virtual table's results.

Creating calculated tables in Power BI Desktop

You can create calculated tables in Power BI Desktop with the function (or functions) you used to create the virtual tables.

We will use part 1 of the second scenario discussed in this chapter to see how this works:

1. In Power BI Desktop, click **New Table** from the **Modeling** tab.
2. Copy and paste the virtual table section, then press *Enter*:

```
Test Virtual Table =
FILTER('Product'                    //Virtual table start
            , 'Product'[List Price]>=1000
                )                              //Virtual
table end
```

3. Click the **Data** tab from the left pane.
4. Select **Test Virtual Table** from the **Fields** pane to see the results.

As you can see, all columns from the `Product` table are available. However, there are only 126 rows loaded into this table, which are the products with a `List Price` value bigger than or equal to 1,000. The original `Product` table has 397 rows:

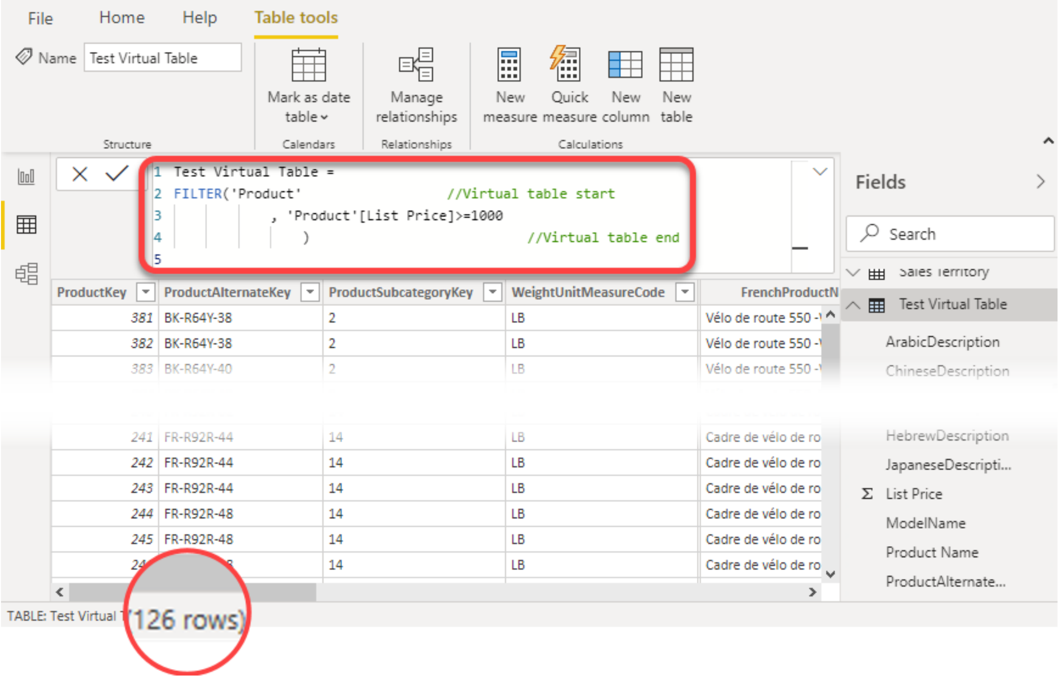


Figure 2.8 – Visually displaying virtual tables in a Power BI Desktop calculated table

Using DAX Studio

DAX Studio is one of the most popular third-party tools available to download for free. You can get it from the tool's official website (`https://daxstudio.org/`). You can easily use DAX Studio to see the results of your virtual tables. First of all, you need to open your Power BI file (`*.pbix`) before you can connect to it from DAX Studio. Open DAX Studio and follow these steps:

1. Click **PBI/SSTD Model**.
2. Select a desired Power BI Desktop instance.

3. Type in the `EVALUATE` statement, then copy and paste the `Virtual Table` part of the calculation:

```
FILTER('Product'                          //Virtual table start
            , 'Product'[List Price]>=1000
                ) 	                           //Virtual
table end
```

4. Press *F5* or click the **Run** button to run the query:

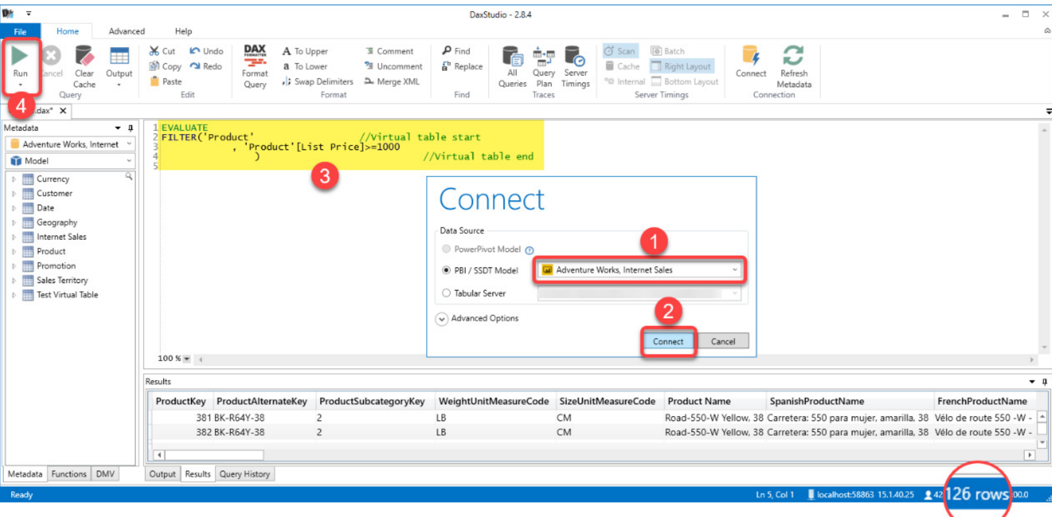


Figure 2.9 – Running virtual table expressions in DAX Studio

> **Note**
> In DAX Studio, you can run DAX queries, which must start with the `EVALUATE` statement.

Relationships in virtual tables

In relational systems, when we think about tables, we usually think about tables and their relationships. So far, we have learned about virtual tables. Now it is time to think about the relationships between virtual tables and other tables (either physical tables available in the data model or other virtual tables). As stated earlier, we create a virtual table by generating it, constructing it, or deriving from an existing table within the data model. Moreover, there are some cases where we can create more than one virtual table to calculate results. When it comes to virtual tables, there are two types of relationships:

- Suppose a virtual table is derived from a table that physically exists in the data model. In that case, there is a relationship between the virtual table and the original physical table, which is the so-called lineage.
- If we create more than one virtual table in a calculation, then we create relationships between those virtual tables programmatically

Either way, the relationship is not an actual physical relationship within the data model. It is created on the fly when a calculation is computed. Therefore, understanding virtual relationships is somewhat involved. So, let's go through a hands-on scenario.

A business requires you to calculate the average product standard cost by `Product Category`, `Product Subcategory`, and `Product`.

To go through this scenario, open the `Chapter 2, Virtual Tables.pbix` sample file. Look at the data model. It is pretty simple. It consists of only two tables, `Product` and `Internet Sales`. Looking more closely, you will see a `ProductStandardCost` column in the `Internet Sales` table to use in our measure. The following figure shows the data model:

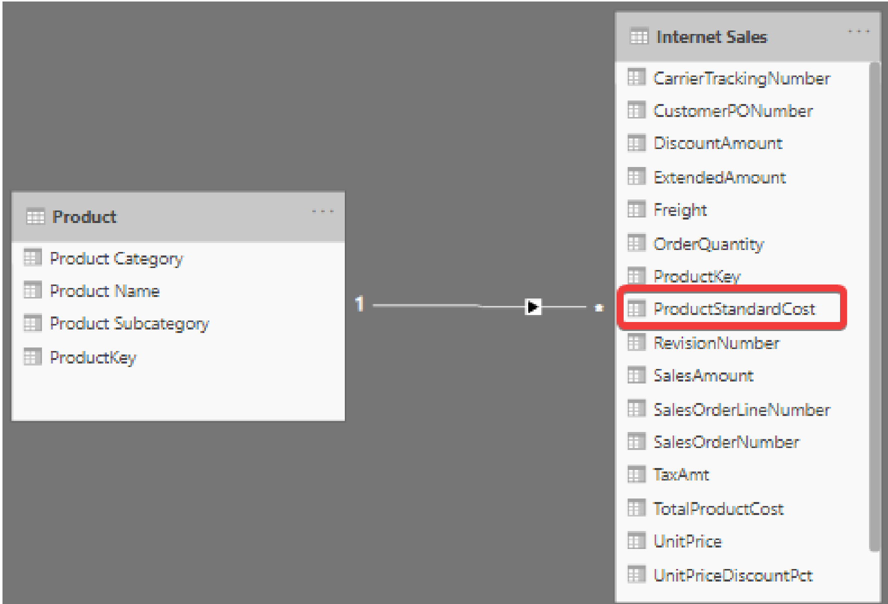


Figure 2.10 – Product and Internet Sales

The next step is to create a measure to calculate the average product standard cost:

```
Avg. Product Standard Costs =
AVERAGE('Internet Sales'[ProductStandardCost])
```

Now can test the measure we created with the following steps:

1. Put a matrix visual on the report canvas.
2. Put the `Product Category`, `Product Subcategory`, and `Product` columns from the `Product` table into the **Rows** section of the matrix visual.
3. Put the `Avg. Product Standard Costs` measure into the **Values** section of the matrix.

The following screenshot illustrates the steps:

Figure 2.11 – Visualizing Avg. Product Standard Costs by Product Category, Product Subcategory, and Product

It looks easy. But let's have a more thorough look at the underlying data and make sure the calculation is correct. Click **Data View** from the left pane in Power BI Desktop, then click the `Internet Sales` table.

As you can see in the following figure, there is a `ProductStandardCost` value for each transaction. In other words, all product costs from when the supplier supplies the item until the item comes off the shelf are considered:

OrderDateKey	SalesOrderNumber	SalesOrderLineNumber	RevisionNumber	OrderQuantity	UnitPrice	ExtendedAmount	ProductStandardCost	CarrierTrackingNumber
20130128	SO51900	1	1	1	4.99	4.99	1.8663	
20130129	SO51948	1	1	1	4.99	4.99	1.8663	
20130131	SO52043	1	1	1	4.99	4.99	1.8663	
20130131	SO52045	1	1	1	4.99	4.99	1.8663	
20130201	SO52094	1	1	1	4.99	4.99	1.8663	
20130203	SO52175	1	1	1	4.99	4.99	1.8663	
20130203	SO52190	1	1	1	4.99	4.99	1.8663	
20130204	SO52232	1	1	1	4.99	4.99	1.8663	
20130204	SO52234	1	1	1	4.99	4.99	1.8663	
20130204	SO52245	1	1	1	4.99	4.99	1.8663	
20130205	SO52301	1	1	1	4.99	4.99	1.8663	
20130205	SO52314	1	1	1	4.99	4.99	1.8663	
20130206	SO52342	1	1	1	4.99	4.99	1.8663	
20130207	SO52387	1	1	1	4.99	4.99	1.8663	
20130222	SO53154	1	1	1	4.99	4.99	1.8663	

Figure 2.12 – Each transaction has a value for ProductStandardCost

The product costs are usually variable, and they change over time. So, it is crucial to know if that is the case in our scenario. If that is the case, then the preceding calculation is correct.

Now let's make this scenario a bit more challenging. To confirm whether our understanding of the requirements is correct, we ask the business to confirm whether they calculate all costs for an item until it sells. We also need to confirm with the business whether we need to keep the product cost history.

The business' response is that their reporting requires always showing the current product standard cost for an item before it goes on the shelf. In other words, there is just one flat rate for the standard cost associated with each product. So, for this specific scenario, we do not need to keep a history of the costs.

That response means that the preceding calculation is incorrect. To fix the issue, we need to move the `ProductStandardCost` column into the `Product` table. In that case, there is only one product standard cost associated with each product. This is how data modeling can help to decrease the level of complexity of our DAX expressions. But, to demonstrate the virtual tables, we create another measure and analyze the results. Before creating the measure, let's review our requirements once again.

We need to get the average of the product standard cost by `Product Category`, `Product Subcategory`, and `Product`. Each product has only one current product standard cost. Therefore, we can create a virtual table with the `ProductKey` column and the `ProductStandardCost` column side by side. Note that the two columns come from different tables. However, since there is a physical relationship between the two tables already, we can easily have the two columns side by side in a single virtual table.

The following measure caters for the preceding requirements:

```
Avg. Product Standard Costs Correct =
AVERAGEX(
    SUMMARISE (
        'Internet Sales'
        , 'Product'[ProductKey]
        ,  'Internet Sales'[ProductStandardCost]
        )
        , 'Internet Sales'[ProductStandardCost]
)
```

We can now add the new measure to the matrix visual we previously put on the report canvas to see both measures side by side. This will quickly show the differences between the two calculations. The totals especially show a big difference between the two average calculations.

Let's analyze the preceding measure to see how it works.

`AVERAGEX()` is an iterator function that iterates through all the rows of its table argument to calculate its expression argument. Here is where we created the virtual table using the `SUMMARIZE()` function.

As you can see in the following figure, we created a virtual table on top of the `Internet Sales` table. We get the `ProductStandardCost` values grouped by `ProductKey` from the `Product` table:

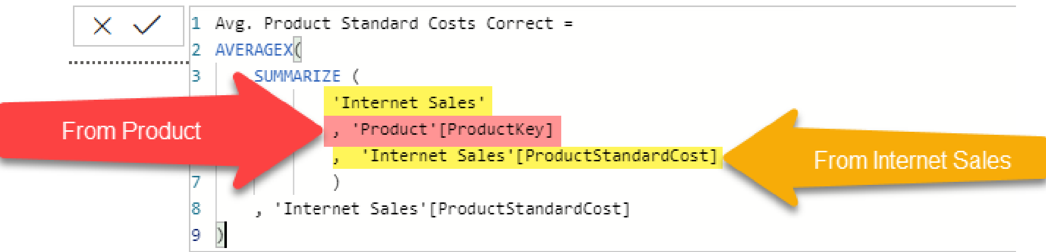


Figure 2.13 – A virtual table created on top of two tables

How is that possible? Here is how it works.

The `AVERAGEX()` function is an iterator function. It goes through each row of the `Internet Sales` table to get the `ProductStandardCost` column's values, grouped by related `ProductKey` from the `Product` table via the existing relationship between the two tables. When we run the `SUMMARIZE()` part of the preceding calculation in DAX Studio, we see that `SUMMARIZE()` retrieves only 158 rows, while the `Internet Sales` table has more than 60,000 rows. The reason is the existing relationship between the `Internet Sales` table and the `Product` table.

The following figure shows the results of running the `SUMMARIZE()` section of the preceding calculation in DAX Studio:

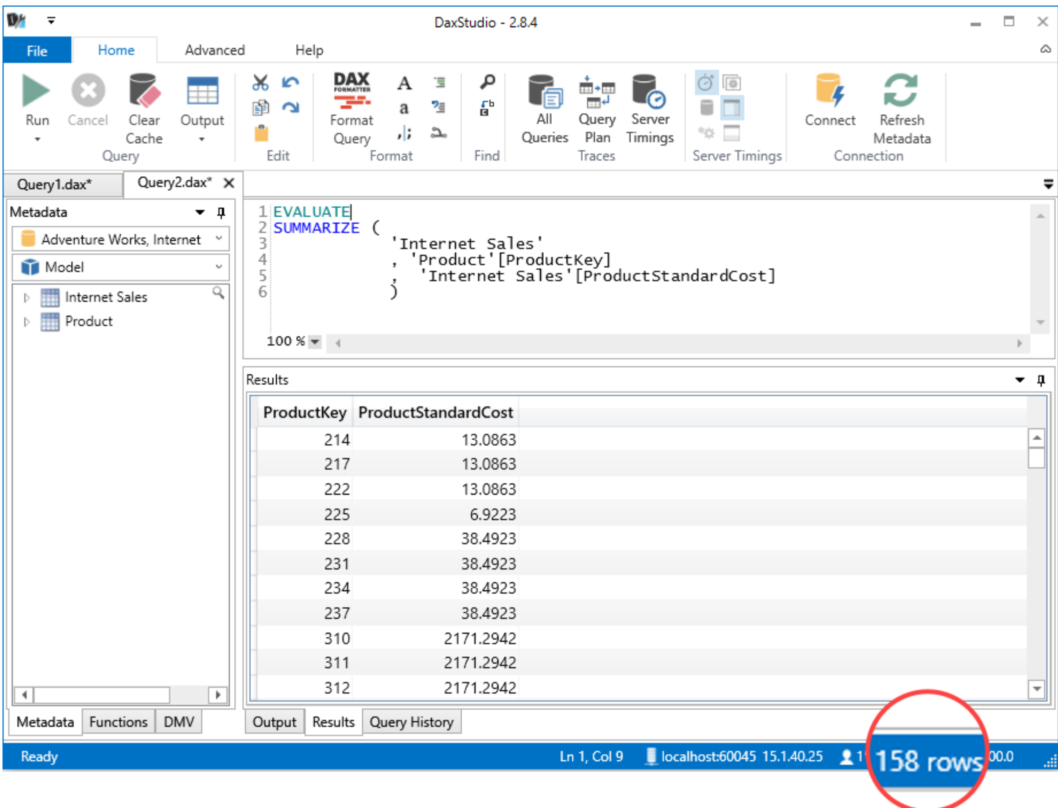


Figure 2.14 – Results of running the SUMMARIZE() part of the calculation in DAX Studio

So far, so good.

The other point to note is that we created a virtual table inside a measure. When we put the `Product Category`, `Product Subcategory`, and `Product` columns into the matrix visual, they show the correct values for the `Avg. Product Standard Costs Correct` measure. We did not physically create any relationships between `Internet Sales` and the virtual table. How come we get the correct results?

The answer is to do with the lineage between the derived virtual table and the original physical table. The virtual table inherits the relationship with the `Internet Sales` table from the `Product` table. In other words, the virtual table has a one-to-one virtual relationship with the `Product` table through `ProductKey`. Hence, when we put the `Product Category`, `Product Subcategory`, and `Product` columns into the matrix visual, the filters propagate to the `Product` table and then to the `Internet Sales` table. The following figure illustrates how the preceding virtual table relates to the `Product` table and the `Internet Sales` table:

> **Note**
> Neither the virtual table nor any virtual relationships are visible in the data model. The following figure is for illustration only.

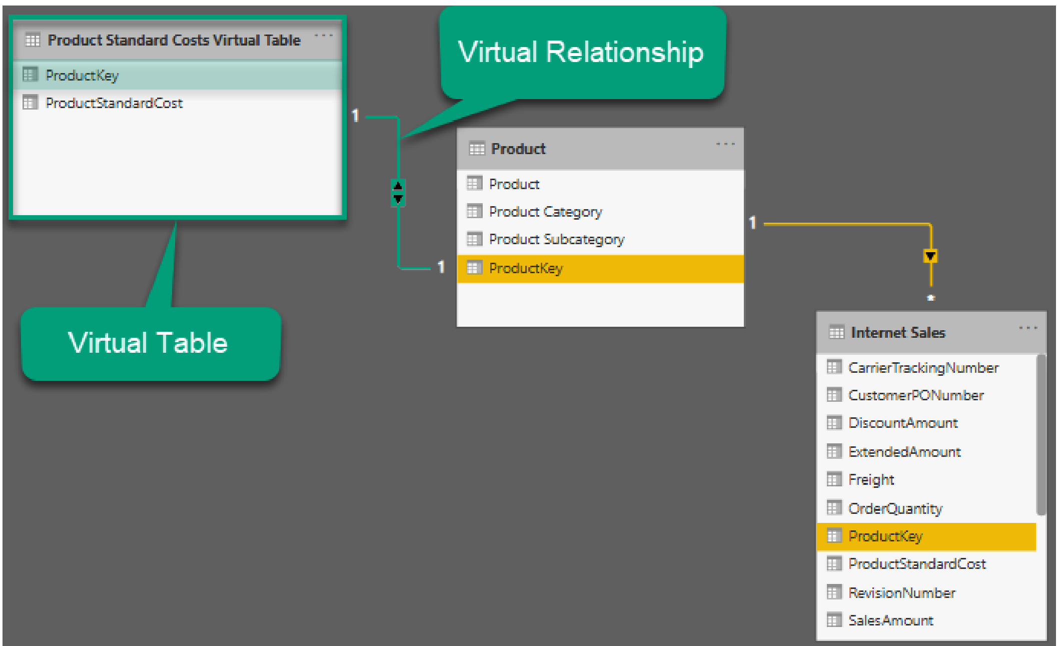


Figure 2.15 – The virtual table and its virtual relationship with the physical table

Using virtual tables is an effective technique that we can use on many different occasions. One such occasion would be when there is no relationship between two tables and we cannot create a physical relationship between the two as the relationship is only legitimate in some business cases.

Let's look at some more complex scenarios where we have multiple virtual tables with inter-virtual relationships.

The business needs to calculate `Internet Sales` in USD. At the same time, there are several values in `Internet Sales` with other currencies. Looking at the `Chapter 2, Virtual Tables and Relationships.pbix` file, we see an `Exchange Rates` table. While the base currency is USD, there is no USD data captured in the `Exchange Rates` table. The following figure shows the `Exchange Rates` data; as you can see, the exchange rate is captured each day. If we want to define a primary key for the `Exchange Rates` table, we can come up with a composite key of `CurrencyKey` and `Date`:

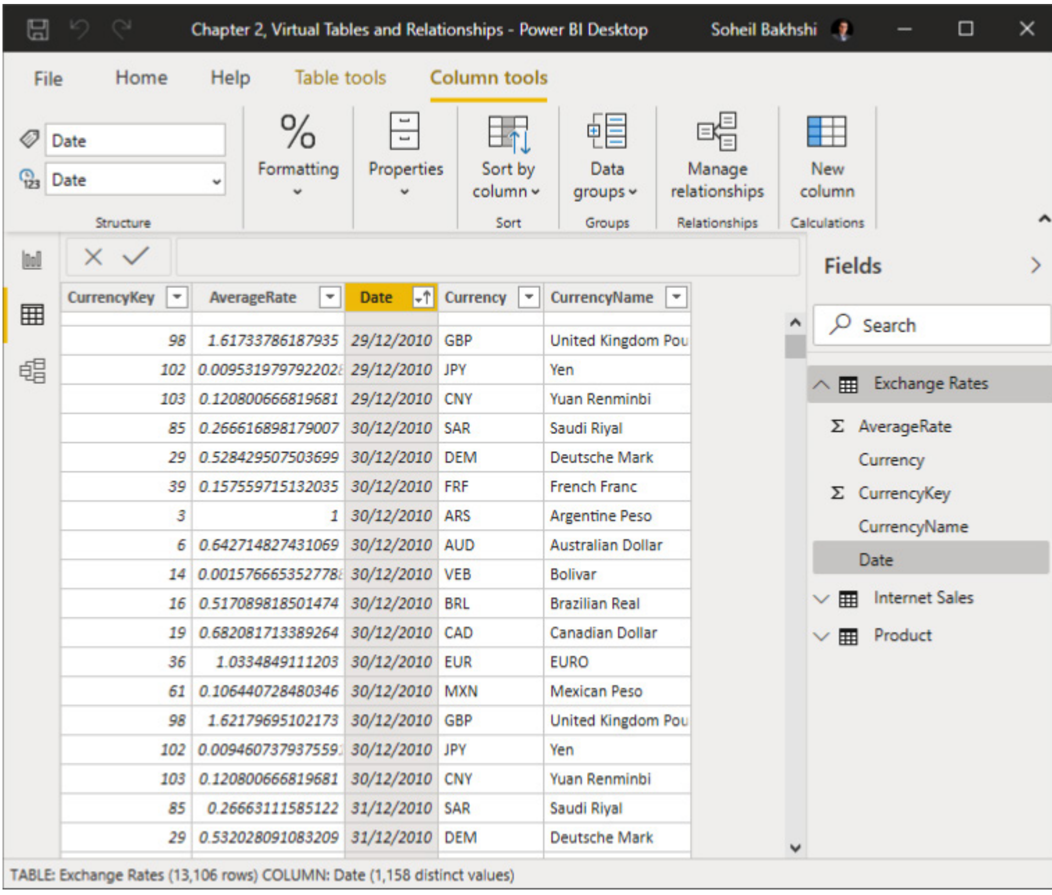


Figure 2.16 – Exchange Rates data

Looking more thoroughly at the `Exchange Rates` data shows that the base currency is USD. So, the `AverageRate` column shows the conversion rate from the value of the `Currency` column to USD, on a specific date, for each row.

To better understand the scenario, let's look at the `Internet Sales` and `Exchange Rates` data side by side, as in the following figure:

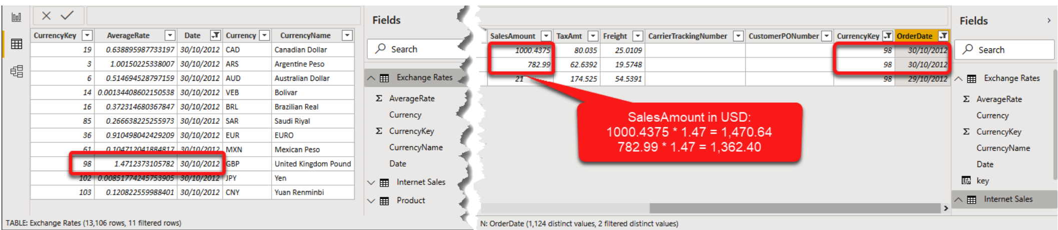


Figure 2.17 – Calculating Internet Sales in USD

To calculate the internet sales in USD on a specific date, we need to find the relevant `CurrencyKey` for that specific date in the `Exchange Rates` table, then multiply the value of `SalesAmount` from the `Internet Sales` table by the value of `AverageRate` from the `Exchange Rates` table.

The following measure caters for that:

```
Internet Sales USD =
SUMX(
    NATURALINNERJOIN (
            SELECTCOLUMNS(
                'Internet Sales'
, "CurrencyKeyJoin", 'Internet Sales'[CurrencyKey] * 1
                , "DateJoin", 'Internet Sales'[OrderDate] + 0
                , "ProductKey", 'Internet Sales'[ProductKey]
                , "SalesOrderLineNumber", 'Internet
Sales'[SalesOrderLineNumber]
                , "SalesOrderNumber", 'Internet
Sales'[SalesOrderNumber]
                , "SalesAmount", 'Internet Sales'[SalesAmount]
                )
            , SELECTCOLUMNS (
                'Exchange Rates'
                , "CurrencyKeyJoin", 'Exchange
Rates'[CurrencyKey] * 1
```

```
                    , "DateJoin", 'Exchange Rates'[Date] + 0
                    , "AverageRate", 'Exchange Rates'[AverageRate]
                )
            )
    , [AverageRate] * [SalesAmount]
)
```

The following figure shows how the preceding calculation works:

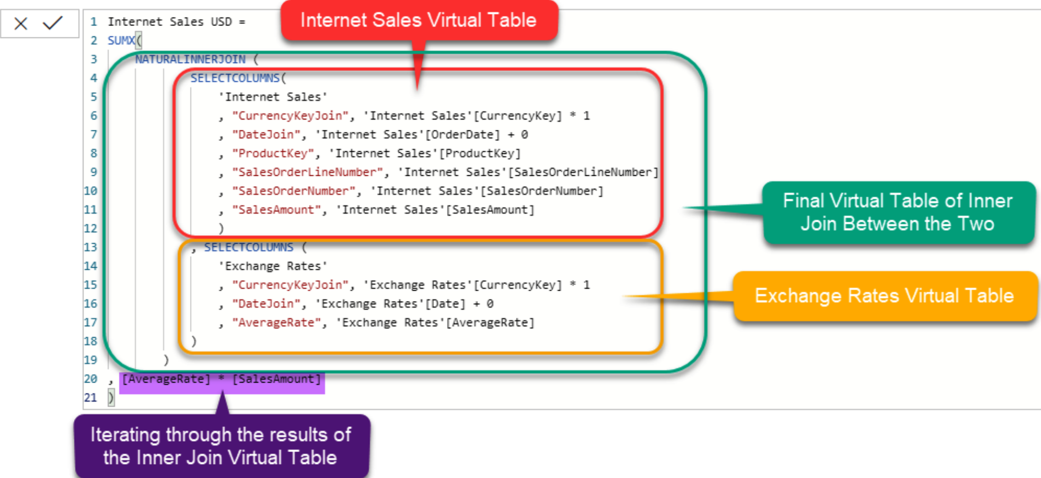


Figure 2.18 – How Internet Sales USD works

As the preceding figure shows, we first create two virtual tables. We join those two virtual tables using two columns, the `CurrencyKeyJoin` and `DateJoin` columns. If you look at the construction of the two columns, you will see the following:

- We added `0` days to `'Internet Sales'[OrderDate]` to construct `DateJoin` for the virtual table derived from the `Internet Sales` table. We did the same to `'Exchange Rates'[Date]` to construct `DateJoin` for the virtual table derived from the `Exchange Rates` table.
- We multiplied `CurrencyKey` by `1` to construct the `CurrencyKeyJoin` column in both virtual tables.

At this stage, you may ask why we need to do any of this. The reason is purely to make the `NATURALINNERJOIN()` function work. The `NATURALINNERJOIN()` function, as its name implies, performs an inner join of a table with another table using the same column names with the same data type and the same lineage.

We want to perform an inner join between the two virtual tables based on the following columns:

- `'Internet Sales'[OrderDate]` → `'Exchange Rates'[Date]`
- `'Internet Sales'[CurrencyKey]` → `'Exchange Rates'[CurrencyKey]`

The first requirement for the `NATURALINNERJOIN()` function is to have the same column names in both joining tables. To meet that requirement, we renamed both the `OrderDate` column from the `Internet Sales` table and the `Date` column from the `Exchange Rates` table to `DateJoin`.

The second requirement is that the columns participating in the join must have the same data type. We already meet this requirement.

The last requirement, which is the most confusing one yet, is that the join's columns must have the same lineage if there is a physical relationship between the two tables. If there is no physical relationship between the tables, the columns participating in the join must not have lineage to any physical columns within the data model. Therefore, we need to break the lineage of the join columns. Otherwise, the `NATURALINNERJOIN()` function will not work. To break the lineage, we need to use an expression rather than an actual column name. Therefore, we added `0` days to `'Internet Sales'[OrderDate]` and multiplied `'Exchange Rates'[CurrencyKey]` by `1` to break the lineage of those two columns.

Finally, we have the desired result set, so we can multiply `[SalesAmount]` by `[AverageRate]` to get the sales amount in USD.

The following figure shows the `Internet Sales` and `Internet Sales USD` measures side by side:

Product Category	Internet Sales	Internet Sales USD
Accessories	**700,759.96**	**$640,920.11**
All-Purpose Bike Stand	39,591.00	$35,628.69
Bike Wash - Dissolver	7,218.60	$6,352.51
Fender Set - Mountain	46,619.58	$41,974.10
Hitch Rack - 4-Bike	39,360.00	$35,934.55
HL Mountain Tire	48,860.00	$42,839.80
HL Road Tire	27,970.80	$24,531.24
Hydration Pack - 70 oz.	40,307.67	$35,307.84
LL Mountain Tire	21,541.38	$19,092.44
LL Road Tire	22,435.56	$21,492.01
ML Mountain Tire	34,818.39	$31,580.66
ML Road Tire	23,140.74	$20,554.35
Mountain Bottle Cage	20,229.75	$18,814.51
Mountain Tire Tube	15,444.05	$13,759.74
Patch Kit/8 Patches	7,307.39	$6,595.31
Road Bottle Cage	15,390.88	$13,953.77
Total	**29,358,677.22**	**$26,054,827.45**

Figure 2.19 – Visualizing Internet Sales and Internet Sales USD side by side in a matrix visual

> **Note**
>
> In virtual tables, we can join two tables using multiple columns.

We created the `Internet Sales USD` measure to demonstrate how virtual tables and virtual relationships work. In real-world scenarios, best practice is to move the `AverageRate` column to the `Internet Sales` table where possible. In our example, it is easily possible. We merge the `Internet Sales` table with the `Exchange Rates` table in Power Query Editor to get the `AverageRate` column into `Internet Sales`.

> **Notes**
>
> It is best practice to implement business logic in the source code when possible. If you cannot take care of the business logic in the source code, then try to solve the issue(s) in Power Query Editor. If the logic is too complex to implement in Power Query Editor, look at the data model and investigate the possibilities for creating physical relationships rather than virtual ones. For instance, in the `Exchange Rates` scenario, we can move the rates for each transaction to the `Internet Sales` table without changing the granularity of `Internet Sales`.
>
> Review the business logic and look for any conformities of using the same virtual tables. If you are likely to use the same virtual table(s) multiple times, try to create either physical tables or calculated tables that can be used across the data model.
>
> Virtual tables and virtual relationships are potent tools to have in your toolbelt, yet they are costly. When used on large amounts of data, you may not get good performance out of your virtual tables.

So far, we have looked at virtual tables, how they work through relationships, and how we can leverage their power in our model. In the next section, we look at **time intelligence** in data modeling.

Time intelligence and data modeling

Time intelligence is one of the most powerful and commonly used functionalities in Power BI. For those coming from a SQL development background, it will be pretty clear how hard it is to build time intelligence analysis in a relational database system such as SQL Server. These complex calculations are made easy in Power BI by just a handful of time intelligence functions. This section briefly looks at the common challenges of working with time intelligence functions in Power BI.

> **Note**
>
> Our aim in this book is not to teach you how to use DAX functions; we just cover some more relevant data modeling concepts.

There are currently 35 time intelligence functions available in Power BI. You can find the complete list of functions on the Microsoft website: `https://docs.microsoft.com/en-us/dax/time-intelligence-functions-dax`.

Detecting valid dates in the date dimension

When dealing with periodic calculations in time intelligence, it is often hard to get just valid dates to show in the visuals.

Let's have a look at how to do this with a scenario.

A business needs to see the following calculations:

- **Internet Sales Month to Date** (**MTD**)
- **Internet Sales Last Month to Date** (**LMTD**)
- **Internet Sales Year to Date** (**YTD**)
- **Internet Sales Last Year to Date** (**LYTD**)
- **Internet Sales Last Year Month to Date** (**LY MTD**)

Calculating all this is super easy as there are some DAX functions that are available for us to use. The calculations are as follows.

To calculate `MTD`, use the following DAX expressions:

```
Internet Sales MTD =
TOTALMTD(
     [Internet Sales]
     , 'Date'[Full Date]
 )
```

Use the following DAX expressions to calculate `LMTD`:

```

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

Use the following DAX expressions to calculate `YTD`:

```
Internet Sales YTD =
TOTALYTD(
    [Internet Sales]
    , 'Date'[Full Date]
    )
```

Use the following DAX expressions to calculate `LYTD`:

```
Internet Sales LYTD =
TOTALYTD (
    [Internet Sales]
    , DATEADD('Date'[Full Date], -1, YEAR)
)
```

Finally, use the following DAX expressions to calculate `LY MTD`:

```
Internet Sales LY MTD =
TOTALMTD(
    [Internet Sales]
    , SAMEPERIODLASTYEAR('Date'[Full Date])
)
```

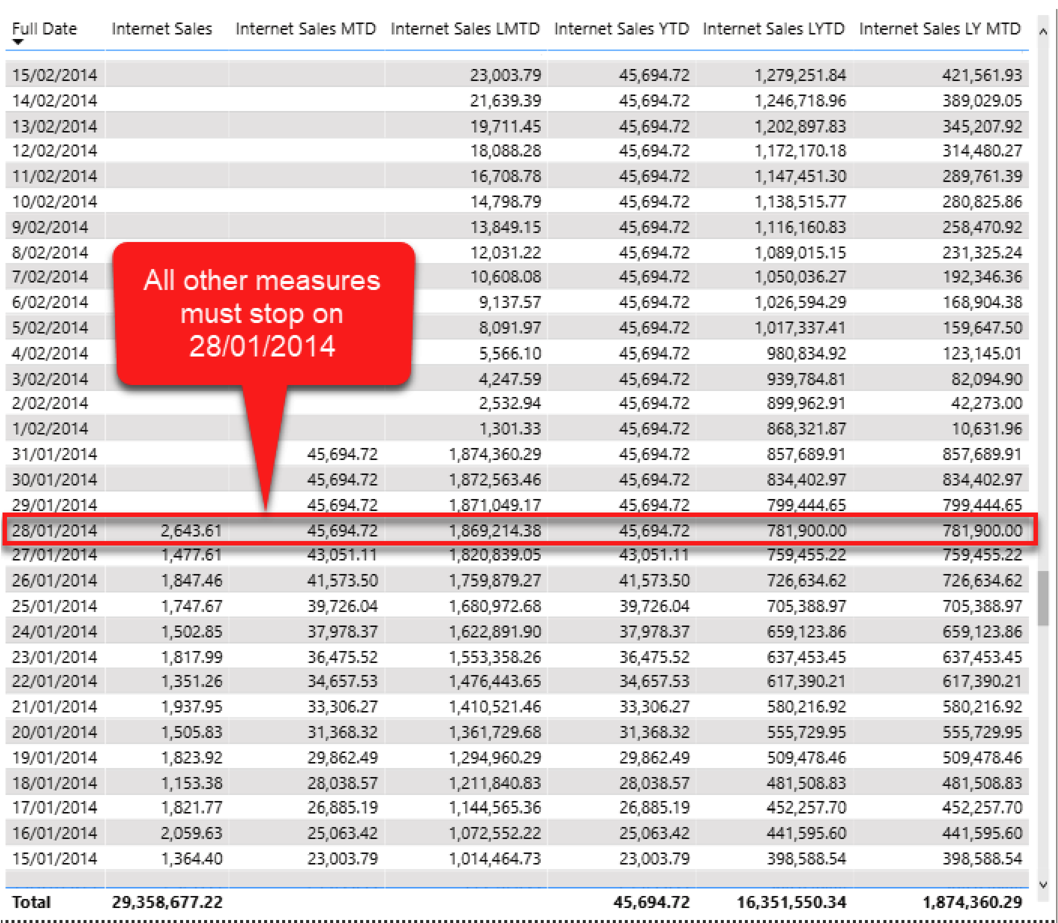
We can then put a table visual on the report canvas to show them side by side with `Full Date` from the `Date` table. Everything looks good unless we sort the `Full Date` column in descending order, which is when we realize that there is an issue. We get many null values for the `Internet Sales`, `Internet Sales MTD`, and `Internet Sales LMTD` measures for many dates. We are also getting a lot of duplicate values for the `Internet Sales YTD` measure.

The following figure illustrates the results:

Full Date	Internet Sales	Internet Sales MTD	Internet Sales LMTD	Internet Sales YTD	Internet Sales LYTD	Internet Sales LY MTD
31/12/2014				45,694.72	16,351,550.34	1,874,360.29
30/12/2014				45,694.72	16,349,753.51	1,872,563.46
29/12/2014				45,694.72	16,348,239.22	1,871,049.17
28/12/2014				45,694.72	16,346,404.43	1,869,214.38
27/12/2014				45,694.72	16,298,029.10	1,820,839.05
26/12/2014				45,694.72	16,237,069.32	1,759,879.27
25/12/2014				45,694.72	16,158,162.73	1,680,972.68
24/12/2014				45,694.72	16,100,081.95	1,622,891.90
23/12/2014				45,694.72	16,030,548.31	1,553,358.26
22/12/2014				45,694.72	15,953,633.70	1,476,443.65
21/12/2014				45,694.72	15,887,711.51	1,410,521.46
20/12/2014				45,694.72	15,838,919.73	1,361,729.68
19/12/2014				45,694.72	15,772,150.34	1,294,960.29
18/12/2014				45,694.72	15,689,030.88	1,211,840.83
17/12/2014				45,694.72	15,621,755.41	1,144,565.36
16/12/2014				45,694.72	15,549,742.27	1,072,552.22
15/12/2014				45,694.72	15,491,654.78	1,014,464.73
14/12/2014				45,694.72	15,420,654.38	943,464.33
13/12/2014				45,694.72	15,362,190.34	885,000.29
12/12/2014				45,694.72	15,296,625.96	819,435.91
11/12/2014				45,694.72	15,200,014.35	722,824.30
10/12/2014				45,694.72	15,142,292.49	665,102.44
9/12/2014				45,694.72	15,087,238.76	610,048.71
8/12/2014				45,694.72	15,007,394.58	530,204.53
7/12/2014				45,694.72	14,949,089.82	471,899.77
6/12/2014				45,694.72	14,885,605.61	408,415.56
5/12/2014				45,694.72	14,823,418.61	346,228.56
4/12/2014				45,694.72	14,734,321.19	257,131.14
3/12/2014				45,694.72	14,651,472.92	174,282.87
2/12/2014				45,694.72	14,585,185.26	107,995.21
1/12/2014				45,694.72	14,535,718.65	58,528.60
30/11/2014				45,694.72	14,477,190.05	1,780,920.06
29/11/2014				45,694.72	14,441,694.02	1,745,424.03
Total	**29,358,677.22**			**45,694.72**	**16,351,550.34**	**1,874,360.29**

Figure 2.20 – Future dates-related issue in the periodic time intelligence calculations

This is indeed an expected behavior. We must cover all dates from 1st January up to the 31st December of all years in the date dimension. So, it is expected that we will get null values for future dates for the `Internet Sales`, `Internet Sales MTD`, and `Internet Sales LMTD` measures and duplicate values for the `Internet Sales YTD` measure. The following figure shows the results when we scroll down a bit in the table visual. We can see that the last date with a valid transaction is **28/01/2014**, which means all the preceding measures must finish the calculations by that date:



Full Date	Internet Sales	Internet Sales MTD	Internet Sales LMTD	Internet Sales YTD	Internet Sales LYTD	Internet Sales LY MTD
15/02/2014			23,003.79	45,694.72	1,279,251.84	421,561.93
14/02/2014			21,639.39	45,694.72	1,246,718.96	389,029.05
13/02/2014			19,711.45	45,694.72	1,202,897.83	345,207.92
12/02/2014			18,088.28	45,694.72	1,172,170.18	314,480.27
11/02/2014			16,708.78	45,694.72	1,147,451.30	289,761.39
10/02/2014			14,798.79	45,694.72	1,138,515.77	280,825.86
9/02/2014			13,849.15	45,694.72	1,116,160.83	258,470.92
8/02/2014			12,031.22	45,694.72	1,089,015.15	231,325.24
7/02/2014			10,608.08	45,694.72	1,050,036.27	192,346.36
6/02/2014			9,137.57	45,694.72	1,026,594.29	168,904.38
5/02/2014			8,091.97	45,694.72	1,017,337.41	159,647.50
4/02/2014			5,566.10	45,694.72	980,834.92	123,145.01
3/02/2014			4,247.59	45,694.72	939,784.81	82,094.90
2/02/2014			2,532.94	45,694.72	899,962.91	42,273.00
1/02/2014			1,301.33	45,694.72	868,321.87	10,631.96
31/01/2014		45,694.72	1,874,360.29	45,694.72	857,689.91	857,689.91
30/01/2014		45,694.72	1,872,563.46	45,694.72	834,402.97	834,402.97
29/01/2014		45,694.72	1,871,049.17	45,694.72	799,444.65	799,444.65
28/01/2014	2,643.61	45,694.72	1,869,214.38	45,694.72	781,900.00	781,900.00
27/01/2014	1,477.61	43,051.11	1,820,839.05	43,051.11	759,455.22	759,455.22
26/01/2014	1,847.46	41,573.50	1,759,879.27	41,573.50	726,634.62	726,634.62
25/01/2014	1,747.67	39,726.04	1,680,972.68	39,726.04	705,388.97	705,388.97
24/01/2014	1,502.85	37,978.37	1,622,891.90	37,978.37	659,123.86	659,123.86
23/01/2014	1,817.99	36,475.52	1,553,358.26	36,475.52	637,453.45	637,453.45
22/01/2014	1,351.26	34,657.53	1,476,443.65	34,657.53	617,390.21	617,390.21
21/01/2014	1,937.95	33,306.27	1,410,521.46	33,306.27	580,216.92	580,216.92
20/01/2014	1,505.83	31,368.32	1,361,729.68	31,368.32	555,729.95	555,729.95
19/01/2014	1,823.92	29,862.49	1,294,960.29	29,862.49	509,478.46	509,478.46
18/01/2014	1,153.38	28,038.57	1,211,840.83	28,038.57	481,508.83	481,508.83
17/01/2014	1,821.77	26,885.19	1,144,565.36	26,885.19	452,257.70	452,257.70
16/01/2014	2,059.63	25,063.42	1,072,552.22	25,063.42	441,595.60	441,595.60
15/01/2014	1,364.40	23,003.79	1,014,464.73	23,003.79	398,588.54	398,588.54
Total	**29,358,677.22**			**45,694.72**	**16,351,550.34**	**1,874,360.29**

Figure 2.21 – The last valid date is 28/01/2014

One way that some may think of to solve the issue is to return BLANK() for the invalid dates. The following calculation shows the Internet Sales MTD Blanking Invalid Dates measure that returns null if there is no transaction for a particular date:

```
Internet Sales MTD Blanking Invalid Dates =
VAR lastorderDate = MAX('Internet Sales'[OrderDateKey])
RETURN
IF(
    MAX('Date'[DateKey]) <= lastoderDate
    , TOTALMTD(
            [Internet Sales]
            , 'Date'[Full Date]
            )
    )
```

What we are doing in the preceding measure is simple. We get the maximum of OrderDateKey from the Internet Sales table, and then we add a condition that if an OrderDateKey value does not exist, we return blank; otherwise, we return the TOTALMTD() calculation.

The following figure shows the results of the **Internet Sales MTD Blanking Invalid Dates** measure:

Full Date	Internet Sales	Internet Sales MTD Blanking Invalid Dates
28/01/2014	2,643.61	45,694.72
27/01/2014	1,477.61	43,051.11
26/01/2014	1,847.46	41,573.50
25/01/2014	1,747.67	39,726.04
24/01/2014	1,502.85	37,978.37
23/01/2014	1,817.99	36,475.52
22/01/2014	1,351.26	34,657.53
21/01/2014	1,937.95	33,306.27
20/01/2014	1,505.83	31,368.32
19/01/2014	1,823.92	29,862.49
18/01/2014	1,153.38	28,038.57
Total	**29,358,677.22**	

Figure 2.22 – Internet Sales MTD Blanking Invalid Dates

While this calculation may work for some scenarios, it is not a correct calculation in some other scenarios, such as ours. Looking more precisely at the results reveals an issue. The following figure reveals the problem:

13/02/2011 is missing

Full Date	Internet Sales	Internet Sales MTD Blanking Invalid Dates
7/02/2011	21,088.06	119,199.78
8/02/2011	17,891.35	137,091.13
9/02/2011	21,469.62	158,560.75
10/02/2011	11,929.73	170,490.48
11/02/2011	14,313.08	184,803.56
12/02/2011	21,113.06	205,916.62
14/02/2011	3,578.27	209,494.89
15/02/2011	21,266.34	230,761.23
16/02/2011	17,109.47	247,870.70
17/02/2011	21,266.34	269,137.04
18/02/2011	28,016.32	297,153.36
Total	**29,358,677.22**	

Figure 2.23 – Missing dates as a result of blanking invalid dates

The other way to overcome the issue is to create a `flag` column in the `Date` table to validate each date value in the `Date` table. It shows `TRUE` when the `DateKey` value is between the minimum and maximum `OrderDateKey` values. We can create the new column either in Power Query Editor or as a calculated column using DAX. From a performance and data compression viewpoint, creating a calculated column with DAX is ideal in terms of performance. The column's data type is `True/False`; therefore, its cardinality is low. Hence, the xVelocity engine can perfectly compress it.

The following calculation creates a new calculated column:

```
IsValidDate =
    AND('Date'[DateKey] >= MIN('Internet Sales'[OrderDateKey])
         , 'Date'[DateKey] <= MAX('Internet
Sales'[OrderDateKey])
    )
```

Now we have two options:

- We can simply use the `IsValidDate` calculated column as a visual filter. In this case, we don't need to change the DAX expressions of the original measures. The following figure shows the results in a table visual:

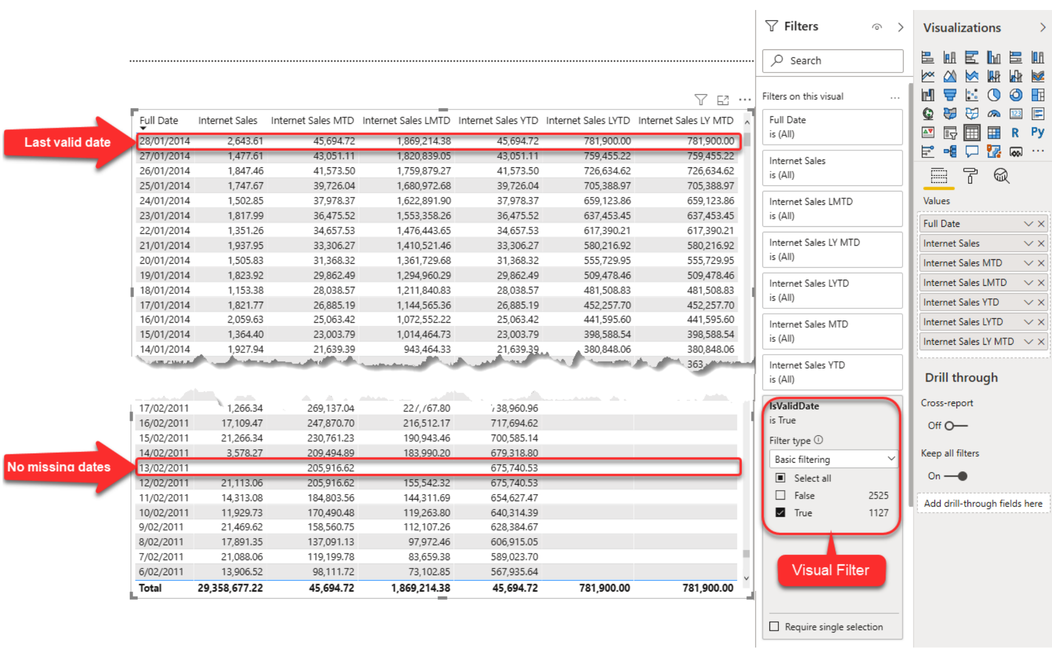


Figure 2.24 – Using the IsValidDate calculated column under Filters

- We can add the `IsValidDate` column to all measures. In this case, we need to filter the results of the periodic functions by `IsValidDate` when `IsValidDate` returns `TRUE()`.

Use the following DAX expressions to calculate `Internet Sales MTD with Valid Dates`:

```
Internet Sales MTD with Valid Dates =
TOTALMTD(
    [Internet Sales]
    , CALCULATETABLE(
        VALUES('Date'[Full Date])
        , 'Date'[IsValidDate] = TRUE()
    )
)
```

Use the following DAX expressions to calculate `Internet Sales LMTD with Valid Dates`:

```
Internet Sales LMTD with Valid Dates =
TOTALMTD(
    [Internet Sales]
    , DATEADD(
          CALCULATETABLE(
                  VALUES('Date'[Full Date])
                  , 'Date'[IsValidDate] = TRUE()
                  )
        , -1
        , MONTH
        )
    )
```

Use the following DAX expressions to calculate `Internet Sales YTD with Valid Dates`:

```
Internet Sales YTD with Valid Dates =
CALCULATE(
    TOTALYTD(
        [Internet Sales]
        , CALCULATETABLE(
                  VALUES('Date'[Full Date])
                  , 'Date'[IsValidDate] = TRUE()
                  )
            ) )
```

Use the following DAX expressions to calculate `Internet Sales LYTD with Valid Dates`:

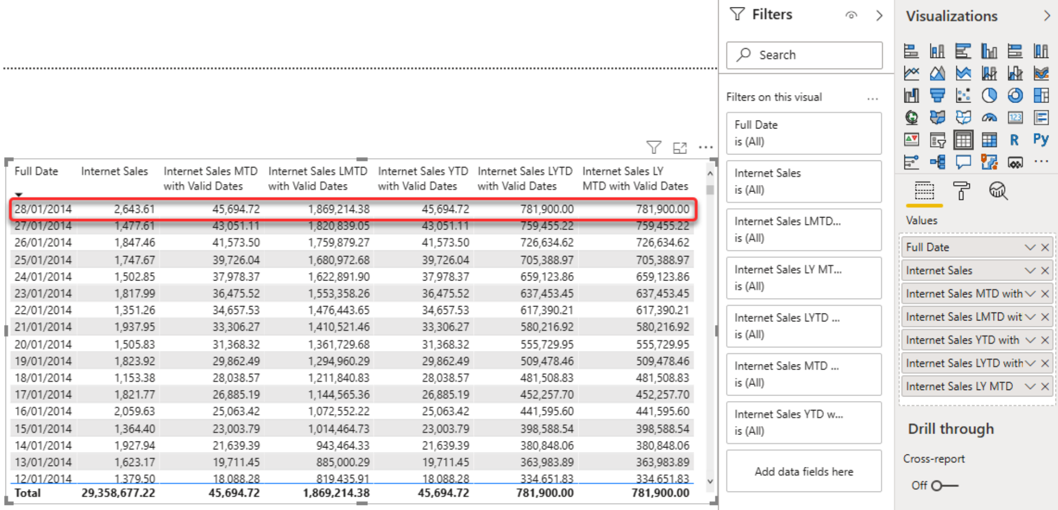
```
Internet Sales LYTD with Valid Dates =
TOTALYTD (
    [Internet Sales]
    , DATEADD(
          CALCULATETABLE(
                  VALUES('Date'[Full Date])
```

```
                        , 'Date'[IsValidDate] = true()
                    )
                , -1
                , YEAR
                )
            )
```

And finally, use the following DAX expressions to calculate `Internet Sales LY MTD with Valid Dates`:

```
Internet Sales LY MTD with Valid Dates =
TOTALMTD(
    [Internet Sales]
    , SAMEPERIODLASTYEAR(
            CALCULATETABLE(
                VALUES('Date'[Full Date])
                , 'Date'[IsValidDate] = TRUE()
                    )
                )
            )
```

The following figure shows the results of putting the preceding measures in a table visual without filtering the visual with the `IsValidDate` column:



Full Date	Internet Sales	Internet Sales MTD with Valid Dates	Internet Sales LMTD with Valid Dates	Internet Sales YTD with Valid Dates	Internet Sales LYTD with Valid Dates	Internet Sales LY MTD with Valid Dates
28/01/2014	2,643.61	45,694.72	1,869,214.38	45,694.72	781,900.00	781,900.00
27/01/2014	1,477.61	43,051.11	1,820,839.05	43,051.11	759,455.22	759,455.22
26/01/2014	1,847.46	41,573.50	1,759,879.27	41,573.50	726,634.62	726,634.62
25/01/2014	1,747.67	39,726.04	1,680,972.68	39,726.04	705,388.97	705,388.97
24/01/2014	1,502.85	37,978.37	1,622,891.90	37,978.37	659,123.86	659,123.86
23/01/2014	1,817.99	36,475.52	1,553,358.26	36,475.52	637,453.45	637,453.45
22/01/2014	1,351.26	34,657.53	1,476,443.65	34,657.53	617,390.21	617,390.21
21/01/2014	1,937.95	33,306.27	1,410,521.46	33,306.27	580,216.92	580,216.92
20/01/2014	1,505.83	31,368.32	1,361,729.68	31,368.32	555,729.95	555,729.95
19/01/2014	1,823.92	29,862.49	1,294,960.29	29,862.49	509,478.46	509,478.46
18/01/2014	1,153.38	28,038.57	1,211,840.83	28,038.57	481,508.83	481,508.83
17/01/2014	1,821.77	26,885.19	1,144,565.36	26,885.19	452,257.70	452,257.70
16/01/2014	2,059.63	25,063.42	1,072,552.22	25,063.42	441,595.60	441,595.60
15/01/2014	1,364.40	23,003.79	1,014,464.73	23,003.79	398,588.54	398,588.54
14/01/2014	1,927.94	21,639.39	943,464.33	21,639.39	380,848.06	380,848.06
13/01/2014	1,623.17	19,711.45	885,000.29	19,711.45	363,983.89	363,983.89
12/01/2014	1,379.50	18,088.28	819,435.91	18,088.28	334,651.83	334,651.83
Total	**29,358,677.22**	**45,694.72**	**1,869,214.38**	**45,694.72**	**781,900.00**	**781,900.00**

Figure 2.25 – Periodic calculations with improved measures

This method quickly resolves another issue when we use the `Full Date` column in a slicer visual. It is much nicer if we only show the valid dates in the slicer. The following figure illustrates a slicer visual showing full date ranges from the `Date` table, side by side with another slicer visual showing only valid dates:

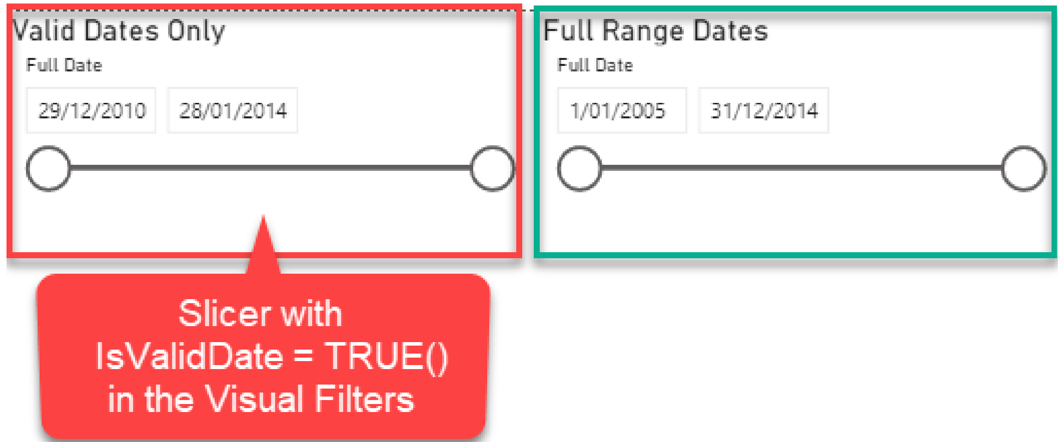


Figure 2.26 – The date slicer can be filtered by IsValidDate

In the preceding image, we used the **IsValidDate** flag column to filter the right slicer's values.

Period-over-period calculations

In many cases, businesses need to perform period-over-period calculations, such as year-over-year and month-over-month calculations. Period-over-period calculations can vary from business to business, but the principles remain the same. Let's move forward with a scenario.

A business needs to be able to dynamically select a *Period-over-Period Variance* calculation in a single area chart based on the following conditions:

- There is just one area chart on the report canvas.
- The user can choose between year-over-year or month-over-month calculations. The year-over-year calculation compares `Internet Sales` values by `Internet Sales Last Year` (**LY**). Month-over-month compares `Internet Sales` values by `Internet Sales Last Month` (**LM**).

- The calculations of *Period-over-Period Variance* are as follows:

```
Internet Sales MoM Variance = Internet Sales – Internet
Sales LM
```

```
Internet Sales YoY Variance = Internet Sales – Internet
Sales LY
```

This may sound like a data visualization-related assignment. However, in reality, we have to take care of almost 99% of it in the data model by traversing all the necessary calculations and making all the bits and pieces available for the report writers to drag and drop visuals and use fields to create reports.

The first step is to analyze the scenario:

- The business needs to calculate the **Internet Sales Month-over-Month (MoM) variance** and **Internet Sales Year-over-Year (YoY) variance**. The `Internet Sales` measure already exists, but we need to create new measures to calculate `Internet Sales LM` and `Internet Sales LY`.
- We need another measure to consolidate the preceding measures based on the requirements to show on the line chart. We can do this in only one measure, but for reusability, we prefer to create separate measures, then reference those measures in a third measure.
- To satisfy the dynamic measure selection in the line chart, we need to create a new table in Power BI with only one column containing the period-over-period names. We use this column in the consolidating measure (the third measure).

  The calculations are as follows:

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

Internet Sales LY =
CALCULATE(
        [Internet Sales]
        , SAMEPERIODLASTYEAR('Date'[Full Date])
)
```

Before implementing the scenario, let's look at the `Internet Sales` and `Internet Sales LM` measures side by side to understand how to implement `Internet Sales MoM Variance`. The following figure shows the two measures side by side in a table visual:

Full Date	Internet Sales	Internet Sales LM
29/12/2010	14,477.34	
30/12/2010	13,931.52	
31/12/2010	15,012.18	
1/01/2011	7,156.54	
2/01/2011	15,012.18	
3/01/2011	14,313.08	
4/01/2011	7,855.64	
5/01/2011	7,855.64	
6/01/2011	20,909.78	
7/01/2011	10,556.53	
8/01/2011	14,313.08	
9/01/2011	14,134.80	
10/01/2011	7,156.54	
11/01/2011	25,047.89	
12/01/2011	11,230.63	
13/01/2011	14,313.08	
14/01/2011	14,134.80	
15/01/2011	6,953.26	
16/01/2011	25,568.71	
17/01/2011	11,255.63	
18/01/2011	14,313.08	
19/01/2011	38,241.29	
20/01/2011	15,012.18	
21/01/2011	10,734.81	
22/01/2011	11,433.91	
23/01/2011	17,534.79	
24/01/2011	28,041.32	
25/01/2011	19,785.36	
26/01/2011	17,688.07	
27/01/2011	14,402.34	
28/01/2011	15,012.18	
29/01/2011	17,891.35	14,477.34
30/01/2011	10,734.81	13,931.52
Total	**29,358,677.22**	**29,358,677.22**

Figure 2.27 – There are blank values for the last month

The preceding figure shows the issue immediately. `Internet Sales LM` returns null values for the first 28 rows as the first transaction started on **29/12/2010**. Now we create a measure based on the scenario's calculation and see what can go wrong:

```
Internet Sales MoM Variance = [Internet Sales] - [Internet
Sales LM]
```

With that, put the new measure on the table visual. The following figure shows the results:

Full Date	Internet Sales	Internet Sales LM	Internet Sales MoM Variance
29/12/2010	14,477.34		14,477.34
30/12/2010	13,931.52		13,931.52
31/12/2010	15,012.18		15,012.18
1/01/2011	7,156.54		7,156.54
2/01/2011	15,012.18		15,012.18
3/01/2011	14,313.08		14,313.08
4/01/2011	7,855.64		7,855.64
5/01/2011	7,855.64		7,855.64
6/01/2011	20,909.78		20,909.78
7/01/2011	10,556.53		10,556.53
8/01/2011	14,313.08		14,313.08
9/01/2011	14,134.80		14,134.80
10/01/2011	7,156.54		7,156.54
11/01/2011	25,047.89		25,047.89
12/01/2011	11,230.63		11,230.63
13/01/2011	14,313.08		14,313.08
14/01/2011	14,134.80		14,134.80
15/01/2011	6,953.26		6,953.26
16/01/2011	25,568.71		25,568.71
17/01/2011	11,255.63		11,255.63
18/01/2011	14,313.08		14,313.08
19/01/2011	38,241.29		38,241.29
20/01/2011	15,012.18		15,012.18
21/01/2011	10,734.81		10,734.81
22/01/2011	11,433.91		11,433.91
23/01/2011	17,534.79		17,534.79
24/01/2011	28,041.32		28,041.32
25/01/2011	19,785.36		19,785.36
26/01/2011	17,688.07		17,688.07
27/01/2011	14,402.34		14,402.34
28/01/2011	15,012.18		15,012.18
29/01/2011	17,891.35	14,477.34	3,414.01
30/01/2011	10,734.81	13,931.52	-3,196.71
Total	**29,358,677.22**	**29,358,677.22**	**0.00**

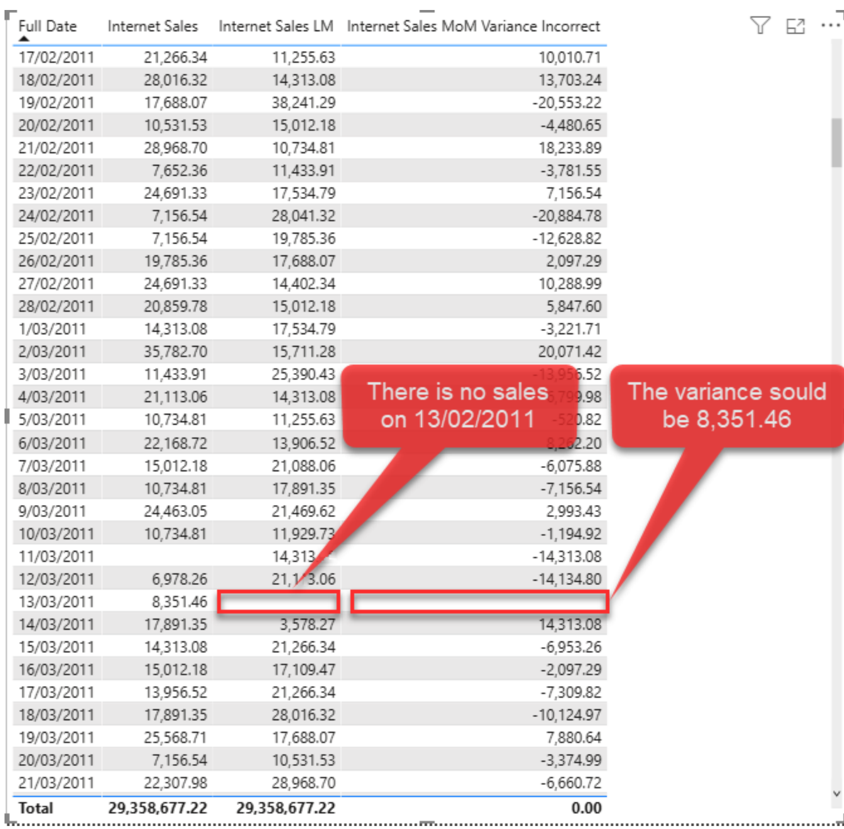
Figure 2.28 – The variance values for the first 28 rows do not make sense

As you see, the variance values start to make sense only after **28/01/2011**. The reason is trivial. There were no sales in the past 28 days, so we have to cut off those values. One of the first solutions that may come to mind is eliminating null values for `Internet Sales LM`. The measure looks as follows:

```
Internet Sales MoM Variance Incorrect =
IF(
    NOT(ISBLANK([Internet Sales LM]))
    , [Internet Sales] - [Internet Sales LM]
    )
```

In the preceding calculation, we say that if `Internet Sales LM` is not blank, get the variance; otherwise, it is blank.

But there is a problem with the calculation. The following figure shows the issue:



Full Date	Internet Sales	Internet Sales LM	Internet Sales MoM Variance Incorrect
17/02/2011	21,266.34	11,255.63	10,010.71
18/02/2011	28,016.32	14,313.08	13,703.24
19/02/2011	17,688.07	38,241.29	-20,553.22
20/02/2011	10,531.53	15,012.18	-4,480.65
21/02/2011	28,968.70	10,734.81	18,233.89
22/02/2011	7,652.36	11,433.91	-3,781.55
23/02/2011	24,691.33	17,534.79	7,156.54
24/02/2011	7,156.54	28,041.32	-20,884.78
25/02/2011	7,156.54	19,785.36	-12,628.82
26/02/2011	19,785.36	17,688.07	2,097.29
27/02/2011	24,691.33	14,402.34	10,288.99
28/02/2011	20,859.78	15,012.18	5,847.60
1/03/2011	14,313.08	17,534.79	-3,221.71
2/03/2011	35,782.70	15,711.28	20,071.42
3/03/2011	11,433.91	25,390.43	-13,956.52
4/03/2011	21,113.06	14,313.08	-6,799.98
5/03/2011	10,734.81	11,255.63	-520.82
6/03/2011	22,168.72	13,906.52	8,262.20
7/03/2011	15,012.18	21,088.06	-6,075.88
8/03/2011	10,734.81	17,891.35	-7,156.54
9/03/2011	24,463.05	21,469.62	2,993.43
10/03/2011	10,734.81	11,929.73	-1,194.92
11/03/2011		14,313[illegible]	-14,313.08
12/03/2011	6,978.26	21,1[illegible]3.06	-14,134.80
13/03/2011	8,351.46		
14/03/2011	17,891.35	3,578.27	14,313.08
15/03/2011	14,313.08	21,266.34	-6,953.26
16/03/2011	15,012.18	17,109.47	-2,097.29
17/03/2011	13,956.52	21,266.34	-7,309.82
18/03/2011	17,891.35	28,016.32	-10,124.97
19/03/2011	25,568.71	17,688.07	7,880.64
20/03/2011	7,156.54	10,531.53	-3,374.99
21/03/2011	22,307.98	28,968.70	-6,660.72
Total	**29,358,677.22**	**29,358,677.22**	**0.00**

Figure 2.29 – The calculation results in incorrect variance values when there are no sales in a day last month

There is also another issue with the preceding calculation. The calculation shows the last month's sales for future dates, which is incorrect. We can see the issue when we sort the results by `Full Date` in descending order. *Figure 2.30* illustrates the problem:

Full Date	Internet Sales	Internet Sales LM	Internet Sales MoM Variance Incorrect
28/02/2014		2,643.61	-2,643.61
27/02/2014		1,477.61	-1,477.61
26/02/2014		1,847.46	-1,847.46
25/02/2014		1,747.67	-1,747.67
24/02/2014		1,502.85	-1,502.85
23/02/2014		1,817.99	-1,817.99
22/02/2014		1,351.26	-1,351.26
21/02/2014		1,937.95	-1,937.95
20/02/2014		1,505.83	-1,505.83
19/02/2014		1,823.92	-1,823.92
18/02/2014		1,153.38	-1,153.38
17/02/2014		1,821.77	-1,821.77
16/02/2014		2,059.63	-2,059.63
15/02/2014		1,364.40	-1,364.40
14/02/2014		1,927.94	-1,927.94
13/02/2014		1,623.17	-1,623.17
12/02/2014		1,379.50	-1,379.50
11/02/2014		1,909.99	-1,909.99
10/02/2014		949.64	-949.64
9/02/2014		1,817.93	-1,817.93
8/02/2014		1,423.14	-1,423.14
7/02/2014		1,470.51	-1,470.51
6/02/2014		1,045.60	-1,045.60
5/02/2014		2,525.87	-2,525.87
4/02/2014		1,318.51	-1,318.51
3/02/2014		1,714.65	-1,714.65
2/02/2014		1,231.61	-1,231.61
1/02/2014		1,301.33	-1,301.33
31/01/2014		1,796.83	-1,796.83
30/01/2014		1,514.29	-1,514.29
29/01/2014		1,834.79	-1,834.79
28/01/2014	2,643.61	48,375.33	-45,731.72
27/01/2014	1,477.61	60,959.78	-59,482.17
Total	**29,358,677.22**	**29,358,677.22**	**0.00**

Figure 2.30 – Future dates show up for Internet Sales LM.
The calculation must stop at the last valid date with Internet Sales

Two failures in a calculation are enough to prompt us to come up with a better solution. The following calculation resolves the issues:

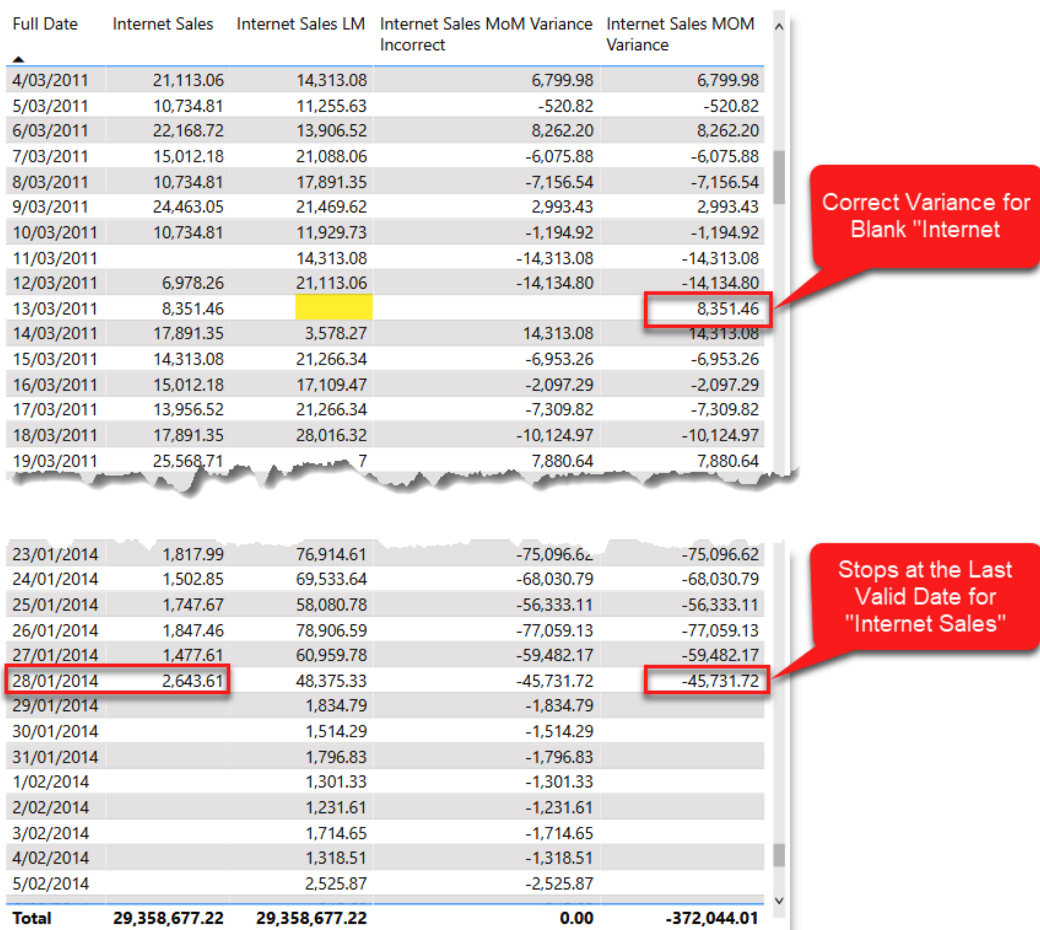
```
Internet Sales MOM Variance =
VAR firstValidDateWithSalesLM = FIRSTNONBLANK(ALL('Date'[Full
Date]), [Internet Sales LM])
VAR lastValidDateWithSalesLM = LASTNONBLANK(ALL('Date'[Full
Date]), [Internet Sales])
RETURN
    SUMX(
        FILTER(
            VALUES('Date'[Full Date])
            , 'Date'[Full Date] >= firstValidDateWithSalesLM
                && 'Date'[Full Date] <=
lastValidDateWithSalesLM
            )
        , [Internet Sales] - [Internet Sales LM]
        )
```

Let's see how the preceding calculation works.

- The `firstValidDateWithSalesLM` variable calculates the first non-blank date from the `Date` table that has `Internet Sales LM` associated with it.
- The `lastValidDateWithSalesLM` variable calculates the last non-blank date from the `Date` table that has `Internet Sales` associated with it.
- In `FILTER()`, we are generating a virtual table, getting the valid dates that fall between the `firstValidDateWithSalesLM` and `LastValidDateWithSalesLM` variables.
- `SUMX()` then iterates through the rows of the virtual table, calculating the variance.

The preceding logic guarantees that we only get valid values for the valid date range. The valid date range starts from the first date with a transaction for `Internet Sales LM` and goes up to the last date with a transaction for `Internet Sales`.

The following figure shows the results:



Full Date	Internet Sales	Internet Sales LM	Internet Sales MoM Variance Incorrect	Internet Sales MOM Variance
4/03/2011	21,113.06	14,313.08	6,799.98	6,799.98
5/03/2011	10,734.81	11,255.63	-520.82	-520.82
6/03/2011	22,168.72	13,906.52	8,262.20	8,262.20
7/03/2011	15,012.18	21,088.06	-6,075.88	-6,075.88
8/03/2011	10,734.81	17,891.35	-7,156.54	-7,156.54
9/03/2011	24,463.05	21,469.62	2,993.43	2,993.43
10/03/2011	10,734.81	11,929.73	-1,194.92	-1,194.92
11/03/2011		14,313.08	-14,313.08	-14,313.08
12/03/2011	6,978.26	21,113.06	-14,134.80	-14,134.80
13/03/2011	8,351.46			8,351.46
14/03/2011	17,891.35	3,578.27	14,313.08	14,313.08
15/03/2011	14,313.08	21,266.34	-6,953.26	-6,953.26
16/03/2011	15,012.18	17,109.47	-2,097.29	-2,097.29
17/03/2011	13,956.52	21,266.34	-7,309.82	-7,309.82
18/03/2011	17,891.35	28,016.32	-10,124.97	-10,124.97
19/03/2011	25,568.71	7	7,880.64	7,880.64
23/01/2014	1,817.99	76,914.61	-75,096.62	-75,096.62
24/01/2014	1,502.85	69,533.64	-68,030.79	-68,030.79
25/01/2014	1,747.67	58,080.78	-56,333.11	-56,333.11
26/01/2014	1,847.46	78,906.59	-77,059.13	-77,059.13
27/01/2014	1,477.61	60,959.78	-59,482.17	-59,482.17
28/01/2014	2,643.61	48,375.33	-45,731.72	-45,731.72
29/01/2014		1,834.79	-1,834.79	
30/01/2014		1,514.29	-1,514.29	
31/01/2014		1,796.83	-1,796.83	
1/02/2014		1,301.33	-1,301.33	
2/02/2014		1,231.61	-1,231.61	
3/02/2014		1,714.65	-1,714.65	
4/02/2014		1,318.51	-1,318.51	
5/02/2014		2,525.87	-2,525.87	
Total	**29,358,677.22**	**29,358,677.22**	**0.00**	**-372,044.01**

Figure 2.31 – The correct calculation shows the correct results when Internet Sales LM is blank and stops the calculation at the last date with an Internet Sales value

Now that we know the logic, we can create the other measure, `Internet Sales LY`, as follows:

```
Internet Sales YoY Variance =
VAR firstValidDateWithSalesLY = FIRSTNONBLANK(ALL('Date'[Full
Date]), [Internet Sales LY])
VAR lastValidDateWithSalesLY = LASTNONBLANK(ALL('Date'[Full
Date]), [Internet Sales])
RETURN
    SUMX(
        FILTER(
```

```
                VALUES('Date'[Full Date])
                , 'Date'[Full Date] >= firstValidDateWithSalesLY
                    && 'Date'[Full Date] <=
lastValidDateWithSalesLY
                )
            , [Internet Sales] - [Internet Sales LY])
```

So far, we have sorted out the initial measures; we can now implement the rest of the scenario.

As *Figure 2.32* illustrates, the next step is to create a new table using the **Enter data** functionality in Power BI and type in the period-over-period values. We can type in `MoM Variance` and `YoY Variance` and click **Load**:

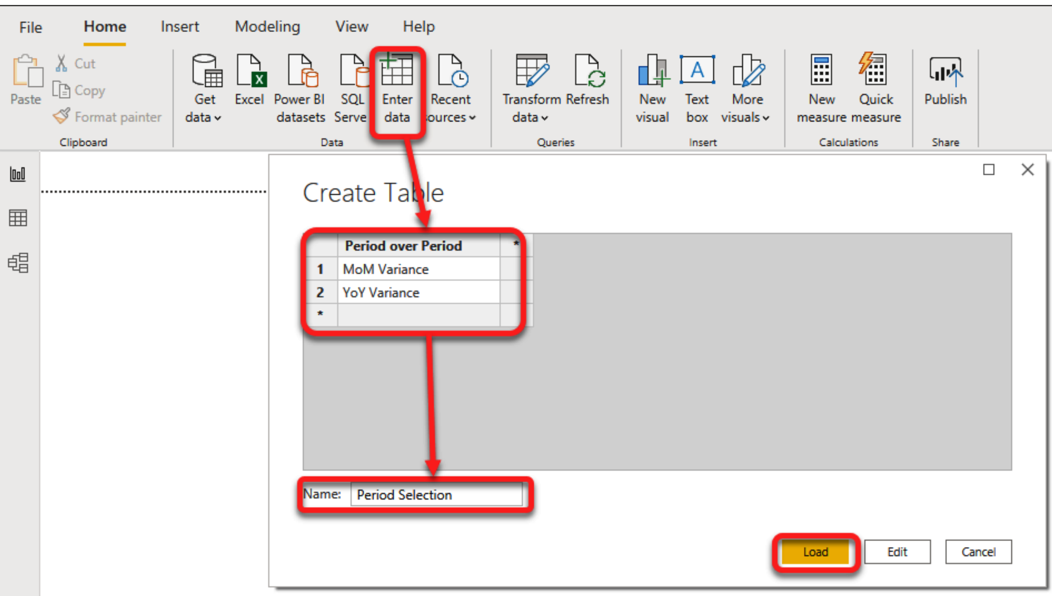


Figure 2.32 – Creating a new table using the Enter data functionality in Power BI

The next step is to put a slicer on the report canvas and use the `Period over Period` column from the `Period Selection` table.

The next step is creating another measure to dynamically switch between the measures based on what the end user selects in the slicer. The following calculation caters for that:

```
Internet Sales PoP Variance =
SWITCH(
        TRUE()
```

```
        , SELECTEDVALUE('Period Selection'[Period over Period])
 = "MoM Variance"
        , [Internet Sales MoM Variance]
        , [Internet Sales YoY Variance]
 )
```

Using the preceding calculation, if the end user selects `MoM Variance` from the slicer, it calls the `Internet Sales MoM Variance` measure; otherwise, it calls the `Internet Sales YoY Variance` measure.

We now put an area chart on the report canvas to test the results. The following figure shows the results when the user selects `MoM Variance`. Note that the starting date and the ending date are correct:

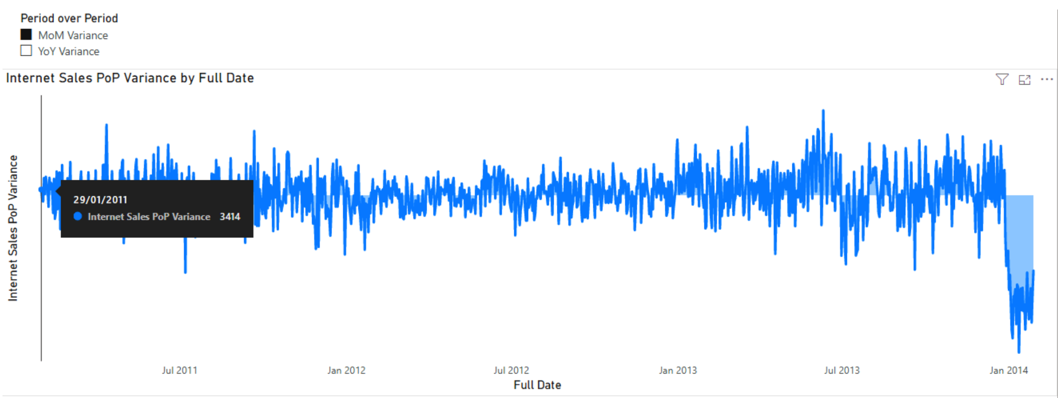


Figure 2.33 – Area chart showing MoM Variance when the user selects MoM Variance from the slicer

The following figure shows the results when the user selects `YoY Variance`:

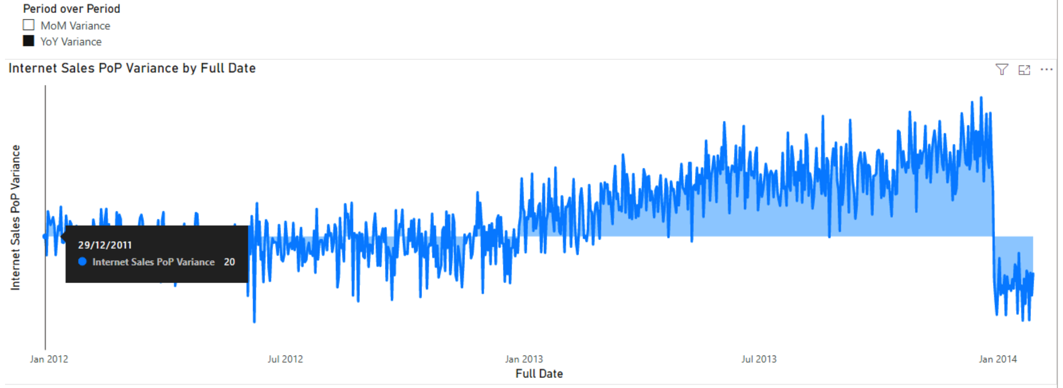


Figure 2.34 – Area chart showing YoY Variance when the user selects YoY Variance from the slicer

With that, we have finished implementing the scenario.

Generating the date dimension with DAX

Thinking about a data model without date values is unrealistic. Sometimes, there can even be more than just one date to describe a single fact. Moreover, in many cases, we need to analyze facts by other date elements, such as year, quarter, month, financial year, public holidays, and so on. Besides, the time intelligence functions in DAX perform the best with a `Date` table. Therefore, having a `Date` table in any model is inevitable. In this section, we learn how to create a `Date` table in Power BI using DAX.

> **Note**
>
> If there is already a `Date` table available in the source system, we do not need to create another using DAX. We have seen some developers with multiple `Date` tables in their model for no particular reason.

When creating the `Date` table, consider the following notes:

- The `Date` table must have at least one column with the `Date` or `DateTime` data type.
- The column containing the date values must be at day granularity (not year/month).
- The `Date` column must start from 1st January of the starting year and go up to 31st December of the ending year.
- The date range contained in the `Date` table must be continuous, so there are no gaps between the dates.

It is good practice to ask the business to provide the date range they would like to cover. We can find the start date from the data model by looking at the fact tables to find the minimum date. But the end date is not usually that simple. Different businesses' requirements are different. Some prefer the end date to always be the 31st December in the current year, while others require a broader date range.

The following two DAX functions can help us to identify the date range:

- `CALENDARAUTO()`: Searches across the data model, among all columns with the `Date` or `DateTime` data type, and finds the minimum date and maximum date. Finally, it generates one column named `Date`, which starts from 1st January of the first date and finishes on 31st December of the last date. This is quite handy, but you need to be careful. It also considers things such as dates of birth or deceased dates, which may result in irrelevant dates being in the model. Besides, in some cases, the data contains a date in the past and a date in the future to demonstrate the unknown dates, such as 01/01/1900 for unknown dates in the past and 31/12/9999 for unknown dates in the future. The `CALENDARAUTO()` function also considers those dates, resulting in having an unnecessarily huge `Date` table. Therefore, we have to tailor the results.
- `CALENDAR()`: Accepts a start date and an end date. Both the start date and end date must be in date format. Like `CALENDARAUTO()`, the `CALENDAR()` function also generates a `Date` column containing date values. But unlike `CALENDARAUTO()`, it does not automatically start from 1st January of the start date, and it does not finish by 31st December of the end date. Remember, the date dimension needs to start from 1st January of the starting year and finish by 31st December of the ending year. Therefore, we need to adjust the results.

As mentioned earlier, if, for any reason, we cannot get the start year and end year from the business, then we need to use one of the preceding DAX functions. If we use the `CALENDAR()` function and have many `Date` or `DateTime` columns across the model, this can be laborious work. Therefore, we can always use the `CALENDARAUTO()` function, which automatically generates the date based on all `Date` or `DateTime` columns across the model. However, we need to review the results. While using the `CALENDARAUTO()` or `CALENDAR()` functions works, there are some other points to consider before creating a `Date` table using DAX. Many Power BI developers create a new calculated table with either of the preceding functions and then add calculated columns to it. While this method works, it is not ideal. The reason for this is that we are creating a few calculated columns. The calculated columns are generated on the fly. Whether we use them in the visuals or not, the data gets loaded into memory at creation time. To release the allocated memory, we need to close the Power BI file.

On the other hand, calculated tables interact differently with memory. The calculated tables are also created on the fly. However, the data is not loaded into the memory unless we use the columns in a visual. So, it is best practice to add the required columns in the `Date` table within the DAX expression to create the calculated table, rather than creating the calculated table and adding those columns as calculated columns later.

While this is more of a performance-tuning strategy, it is good to be conscious of it while creating a `Date` table. The following DAX expressions generate a basic `Date` table using `CALENDARAUTO()`:

```
Date =
VAR firstOderDate = MIN('Internet Sales'[OrderDate])
VAR lastOderDate = MAX('Internet Sales'[OrderDate])
RETURN
ADDCOLUMNS(
    SELECTCOLUMNS(
            CALENDARAUTO()
            , "Full Date"
            , [Date]
            )
    , "DateKey", VALUE(FORMAT([Full Date], "yyyyMMdd"))
    , "Quarter", CONCATENATE("Q ", QUARTER([Full Date]))
    , "Month", FORMAT([Full Date], "MMMM")
    , "Month Short", FORMAT([Full Date], "MMM")
    , "MonthOrder", MONTH([Full Date])
    , "Week", CONCATENATE("Wk ", WEEKNUM([Full Date]))
    , "Day", FORMAT([Full Date], "dddd")
    , "Day Short", FORMAT([Full Date], "ddd")
    , "Day of Month", DAY([Full Date])
    , "DayOrder", WEEKDAY([Full Date], 2) //First day is Monday
    , "Year Month", FORMAT([Full Date], "yyyy-MM")
    , "IsValidDate", AND([Full Date] >= firstOderDate, [Full
Date] <= lastOderDate)
)
```

The following figure shows the results of running the preceding code:

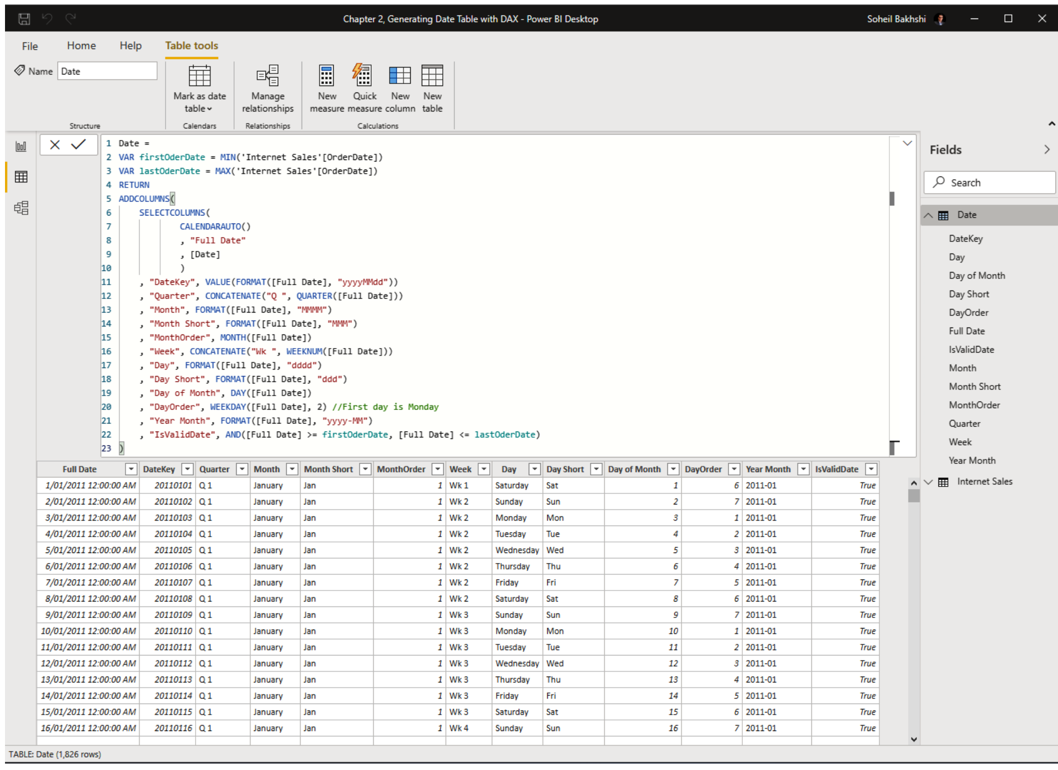


Figure 2.35 – Creating a Date table with DAX

Marking a Date table as a date table

So far, we have discussed the importance of having a `Date` table in our model, and we also looked at some related scenarios. In this section, we discuss an essential aspect of the `date` dimension. As discussed in the previous section, time intelligence functions work best with a `Date` table. In many cases, we only keep the date keys in the fact tables used in the relationship between the fact table and the `Date` dimension. To ensure that the time intelligence functions work correctly, we need to set a unique identifier column with either the `Date` or `DateTime` data type. To specify that unique identifier, we need to set **Mark as Date Table**. Setting this up is super easy. Let's see in a scenario what happens if we forget to set this up.

For this scenario, we will use the `Chapter 2, Mark Date as Date Table Before.pbix` and `Chapter 2, Mark Date as Date Table After.pbix` sample files.

In this scenario, the business needs to analyze the following measures over a **calendar hierarchy**:

- `Internet Sales MTD`
- `Internet Sales YTD`
- `Internet Sales LMTD`
- `Internet Sales LYTD`

The calendar hierarchy holds the following levels:

- Year
- Month
- Full date

> **Note**
>
> The relationship between `Internet Sales` and the `Date` table is created between `OrderDateKey` from `Internet Sales` and `DateKey` from the `Date` table. Both `OrderDateKey` and `DateKey` are of the `Number` data type (integer).

The DAX expressions for the preceding measures are as follows:

Use the following DAX expressions to calculate `Internet Sales MTD`:

```
Internet Sales MTD =
TOTALMTD(
        [Internet Sales]
        , 'Date'[Full Date]
        )
```

Use the following DAX expressions to calculate `Internet Sales YTD`:

```
Internet Sales YTD =
TOTALYTD(
        [Internet Sales]
        , 'Date'[Full Date])
```

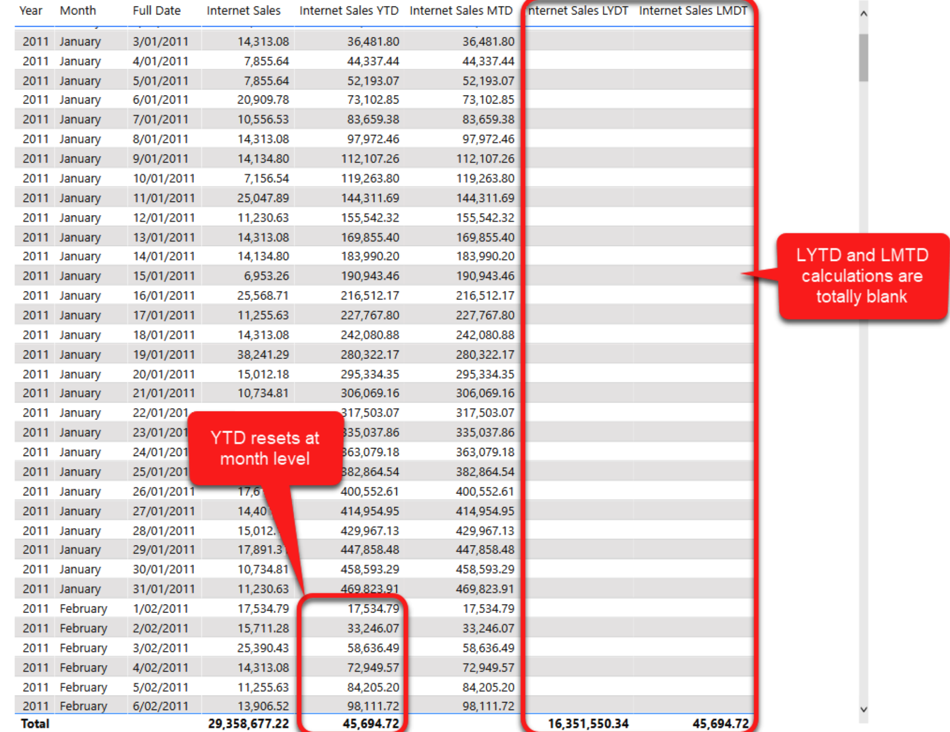
Use the following DAX expressions to calculate `Internet Sales LMTD`:

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

Use the following DAX expressions to calculate `Internet Sales LYTD`:

```
Internet Sales LYTD =
TOTALYTD(
          [Internet Sales]
          , SAMEPERIODLASTYEAR('Date'[Full Date])
)
```

We put a table visual on the report canvas and use the preceding measures adjacent to the `Internet Sales` measure. The following figure shows the results:



Year	Month	Full Date	Internet Sales	Internet Sales YTD	Internet Sales MTD	nternet Sales LYDT	Internet Sales LMDT
2011	January	3/01/2011	14,313.08	36,481.80	36,481.80		
2011	January	4/01/2011	7,855.64	44,337.44	44,337.44		
2011	January	5/01/2011	7,855.64	52,193.07	52,193.07		
2011	January	6/01/2011	20,909.78	73,102.85	73,102.85		
2011	January	7/01/2011	10,556.53	83,659.38	83,659.38		
2011	January	8/01/2011	14,313.08	97,972.46	97,972.46		
2011	January	9/01/2011	14,134.80	112,107.26	112,107.26		
2011	January	10/01/2011	7,156.54	119,263.80	119,263.80		
2011	January	11/01/2011	25,047.89	144,311.69	144,311.69		
2011	January	12/01/2011	11,230.63	155,542.32	155,542.32		
2011	January	13/01/2011	14,313.08	169,855.40	169,855.40		
2011	January	14/01/2011	14,134.80	183,990.20	183,990.20		
2011	January	15/01/2011	6,953.26	190,943.46	190,943.46		
2011	January	16/01/2011	25,568.71	216,512.17	216,512.17		
2011	January	17/01/2011	11,255.63	227,767.80	227,767.80		
2011	January	18/01/2011	14,313.08	242,080.88	242,080.88		
2011	January	19/01/2011	38,241.29	280,322.17	280,322.17		
2011	January	20/01/2011	15,012.18	295,334.35	295,334.35		
2011	January	21/01/2011	10,734.81	306,069.16	306,069.16		
2011	January	22/01/201	[illegible]	317,503.07	317,503.07		
2011	January	23/01/201	[illegible]	335,037.86	335,037.86		
2011	January	24/01/201	[illegible]	363,079.18	363,079.18		
2011	January	25/01/201	[illegible]	382,864.54	382,864.54		
2011	January	26/01/2011	17,6	400,552.61	400,552.61		
2011	January	27/01/2011	14,40	414,954.95	414,954.95		
2011	January	28/01/2011	15,012.	429,967.13	429,967.13		
2011	January	29/01/2011	17,891.3	447,858.48	447,858.48		
2011	January	30/01/2011	10,734.81	458,593.29	458,593.29		
2011	January	31/01/2011	11,230.63	469,823.91	469,823.91		
2011	February	1/02/2011	17,534.79	17,534.79	17,534.79		
2011	February	2/02/2011	15,711.28	33,246.07	33,246.07		
2011	February	3/02/2011	25,390.43	58,636.49	58,636.49		
2011	February	4/02/2011	14,313.08	72,949.57	72,949.57		
2011	February	5/02/2011	11,255.63	84,205.20	84,205.20		
2011	February	6/02/2011	13,906.52	98,111.72	98,111.72		
Total			**29,358,677.22**	**45,694.72**		**16,351,550.34**	**45,694.72**

Figure 2.36 – Our time intelligence functions do not work correctly

The relationship between the `Internet Sales` table and `Date` is created between the `OrderDateKey` and `Date` columns with the `Number` (integer) data type. As *Figure 2.36* shows, the time intelligence functions do not work correctly. One way to fix the issue is to add `ALL('Date')` to all calculations, as follows, and everything works perfectly.

Use the following DAX expressions to calculate `Internet Sales` `MTD`:

```
Internet Sales MTD =
TOTALMTD(
        [Internet Sales]
        , 'Date'[Full Date]
        , ALL('Date')
        )
```

Use the following DAX expressions to calculate `Internet Sales` `YTD`:

```
Internet Sales YTD =
TOTALYTD(
        [Internet Sales]
        , 'Date'[Full Date]
        , ALL('Date')
        )
```

Use the following DAX expressions to calculate `Internet Sales` `LMTD`:

```
Internet Sales LMTD =
TOTALYTD(
        [Internet Sales]
        , DATEADD('Date'[Full Date], -1, MONTH)
        , ALL('Date')
)
```

Use the following DAX expressions to calculate `Internet Sales` `LYTD`:

```
Internet Sales LYTD =
TOTALYTD(
        [Internet Sales]
        , SAMEPERIODLASTYEAR('Date'[Full Date])
        , ALL('Date'))
```

The following figure shows the new results:

Year	Month	Full Date	Internet Sales	Internet Sales YTD	Internet Sales MTD	Internet Sales LYDT	Internet Sales LMDT
2011	January	3/01/2011	14,313.08	36,481.80	36,481.80		
2011	January	4/01/2011	7,855.64	44,337.44	44,337.44		
2011	January	5/01/2011	7,855.64	52,193.07	52,193.07		
2011	January	6/01/2011	20,909.78	73,102.85	73,102.85		
2011	January	7/01/2011	10,556.53	83,659.38	83,659.38		
2011	January	8/01/2011	14,313.08	97,972.46	97,972.46		
2011	January	9/01/2011	14,134.80	112,107.26	112,107.26		
2011	January	10/01/2011	7,156.54	119,263.80	119,263.80		
2011	January	11/01/2011	25,047.89	144,311.69	144,311.69		
2011	January	12/01/2011	11,230.63	155,542.32	155,542.32		
2011	January	13/01/2011	14,313.08	169,855.40	169,855.40		
2011	January	14/01/2011	14,134.80	183,990.20	183,990.20		
2011	January	15/01/2011	6,953.26	190,943.46	190,943.46		
2011	January	16/01/2011	25,568.71	216,512.17	216,512.17		
2011	January	17/01/2011	11,255.63	227,767.80	227,767.80		
2011	January	18/01/2011	14,313.08	242,080.88	242,080.88		
2011	January	19/01/2011	38,241.29	280,322.17	280,322.17		
2011	January	20/01/2011	15,012.18	295,334.35	295,334.35		
2011	January	21/01/2011	10,734.81	306,069.16	306,069.16		
2011	January	22/01/2011	11,433.91	317,503.07	317,503.07		
2011	January	23/01/2011	17,534.79	335,037.86	335,037.86		
2011	January	24/01/2011	28,041.32	363,079.18	363,079.18		
2011	January	25/01/2011	19,785.36	382,864.54	382,864.54		
2011	January	26/01/2011	17,688.07	400,552.61	400,552.61		
2011	January	27/01/2011	14,402.34	414,954.95	414,954.95		
2011	January	28/01/2011	15,012.18	429,967.13	429,967.13		
2011	January	29/01/2011	17,891.35	447,858.48	447,858.48		14,477.34
2011	January	30/01/2011	10,734.81	458,593.29	458,593.29		28,408.86
2011	January	31/01/2011	11,230.63	469,823.91	469,823.91		43,421.04
2011	February	1/02/2011	17,534.79	487,358.70	17,534.79		7,156.54
2011	February	2/02/2011	15,711.28	503,069.98	33,246.07		22,168.72
2011	February	3/02/2011	25,390.43	528,460.41	58,636.49		36,481.80
2011	February	4/02/2011	14,313.08	542,773.49	72,949.57		44,337.44
2011	February	5/02/2011	11,255.63	554,029.12	84,205.20		52,193.07
2011	February	6/02/2011	13,906.52	567,935.64	98,111.72		73,102.85
Total			**29,358,677.22**	**45,694.72**		**16,351,550.34**	**45,694.72**

Figure 2.37 – Resolving the incorrect time intelligence calculations by adding ALL('Date') to all expressions

A better way, indeed the best way, to resolve the issue is to mark the `Date` table as a `date` table and specify `Full Date` as the unique date identifier. There are several ways to do so; here, we show one of them:

1. Right-click the `Date` table from the `Fields` pane.
2. Click **Mark as date table** ➡ **Mark as date table**.

3. Select a column of either the `Date` or `DateTime` data type.
4. Click **OK**:

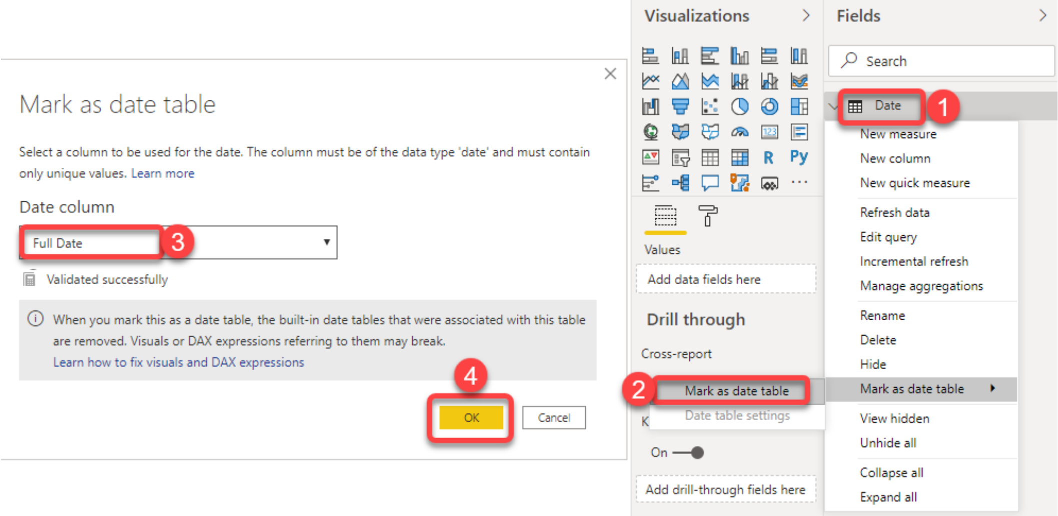


Figure 2.38 – Marking Date as the date table

After setting the `Date` table as the `date` table, we do not need to add `ALL('Date')` to the time intelligence calculations. All the time intelligence functions now work as expected.

Creating a time dimension with DAX

So far, we have discussed the importance of having a `Date` table in our data model. But what if we need to analyze our data at the time level, such as when a business needs to analyze their data at the minute level? This means that the granularity of the fact table would be at the minute level. So, suppose we store the data in our transactional database at the second level. In that case, we need to aggregate that data to the minute level. It is crucial to think about the fact table's granularity in the first steps of the data modeling process.

In most cases, if not all cases, it is better to have a separate `Time` table. We also need to have a `TimeKey` or `Time` column in our fact table to create a relationship between the `Time` table and the fact table. In this section, we discuss a simple way to create a `Time` table using DAX.

The sample file we use in this section is `Chapter 2, Time Table.pbix`, which comes from `FactInternetSales in Time Level.xlsx`.

Let's discuss all this further with a scenario.

A business stores all `Internet Sales` transactions at the second level. The business needs to analyze the business metrics in different time bands, specifically, `5 Min`, `15 Min`, `30 Min`, `45 Min`, and `60 Min`. Resolve this challenge with DAX only.

Looking at the sample data, we see an `OrderDateTime` column of the `DateTime` data type in the `Internet Sales` table. As mentioned in the scenario, the granularity is down to the second level. To solve this challenge purely in DAX, we have to add a calculated column to `Internet Sales` to take the `Time` part of `OrderDateTime`. We use the following DAX expression to create the new calculated column:

```
Order Time = TIMEVALUE(FORMAT([OrderDateTime], "hh:mm:ss"))
```

This column participates in the relationship between the `Time` table and the `Internet Sales` table.

We also need to create a calculated table with DAX that has a `Time` column at second granularity. The following expressions create the `Time` table, including the time bands and the `Time` column at second granularity, that is required by the business:

```
Time =
SELECTCOLUMNS(
    GENERATESERIES(1/86400, 1, TIME(0, 0, 1))
    , "Time", [Value]
    , "Hour", HOUR ( [Value] )
    , "Minute", MINUTE ( [Value] )
    , "5 Min Band",  TIME(HOUR([Value]),
FLOOR(MINUTE([Value])/5, 1) * 5, 0) + TIME(0, 5, 0)
    , "15 Min Band", TIME(HOUR([Value]),
FLOOR(MINUTE([Value])/15, 1) * 15, 0) + TIME(0, 15, 0)
    , "30 Min Band", TIME(HOUR([Value]),
FLOOR(MINUTE([Value])/30, 1) * 30, 0) + TIME(0, 30, 0)
    , "45 Min Band", TIME(HOUR([Value]),
FLOOR(MINUTE([Value])/45, 1) * 45, 0) + TIME(0, 45, 0)
    , "60 Min Band", TIME(HOUR([Value]),
FLOOR(MINUTE([Value])/60, 1) * 60, 0) + TIME(0, 60, 0)
 )
```

The next step is to format all `DateTime` columns to `Time`. The following figure shows the results of running the preceding DAX expressions:

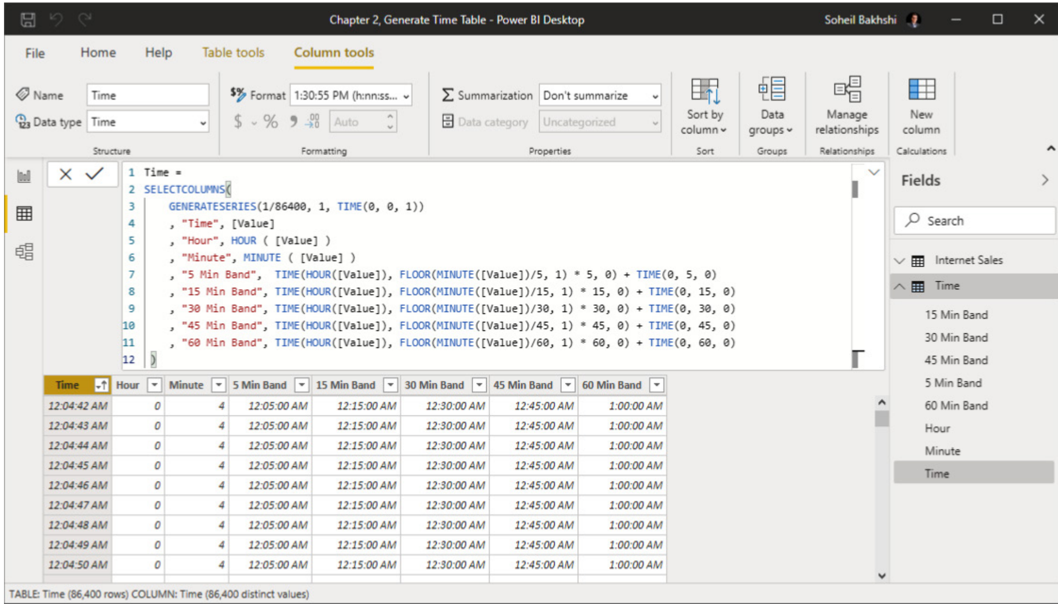


Figure 2.39 – Generating a Time table with DAX

> **Note**
>
> The results shown in the preceding figure are sorted by the `Time` column in ascending order.

The next step is to create a relationship between the `Order Time` column from the `Internet Sales` table and from the `Time` table.

The following figure shows how we can add some area charts to the report canvas and visualize `Internet Sales` by different time bands:

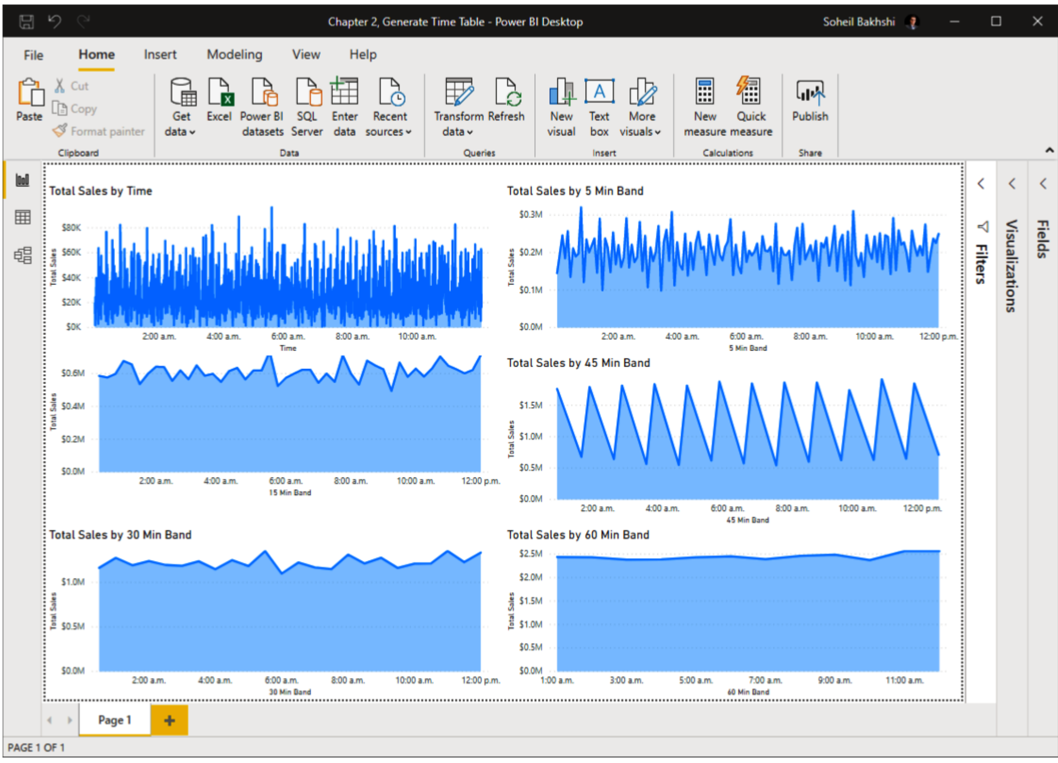


Figure 2.40 – Visualizing Internet Sales by different time bands

> **Note**
>
> It is best practice to move all data transformations, such as generating `Date` or `Time` tables, to the source system as much as possible. If it is not an option for any reason, then it is best to take care of the transformations in Power Query Editor.

Summary

In this chapter, we discussed different aspects of DAX and how it can affect our data modeling. We looked at some real-world scenarios and challenges and how to solve them with DAX.

In the next chapter, we will look at the Power Query (M) expression language in more detail. We will also go through some hands-on scenarios, and we will prepare a star schema in Power Query step by step.

Section 2: Data Preparation in Query Editor

In this section, you will learn how to prepare efficient data models in Query Editor. This chapter is all about transitioning from theory to reality. This chapter explains one of the most important aspects of data modeling, which is data preparation. Not everyone has the luxury of having a pre-built data warehouse; therefore, it is important to know how to build dimensions and facts in Query Editor. Power BI leverages the power of Power Query in Query Editor, so this chapter quickly introduces the Power Query language from a data modeling perspective then explains how to make all components needed in the star schema available to then be used in the data model layer in Power BI. We explain different techniques in data modeling along with real-world hands-on scenarios. We also discuss common pitfalls that can easily turn building a simple report into a nightmare and ways to avoid falling into those traps.

This section comprises the following chapters:

- *Chapter 3, Data Preparation in Power Query Editor*
- *Chapter 4, Getting Data from Various Sources*
- *Chapter 5, Common Data Preparation Steps*

- *Chapter 6, Star Schema Preparation in Power Query Editor*
- *Chapter 7, Data Preparation Common Best Practices*

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:

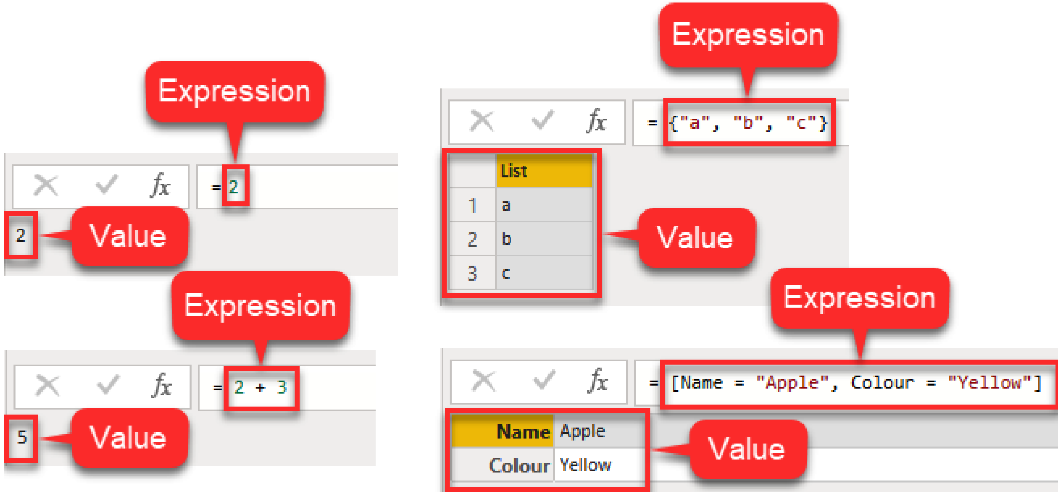


Figure 3.1 – Expressions and their values

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:

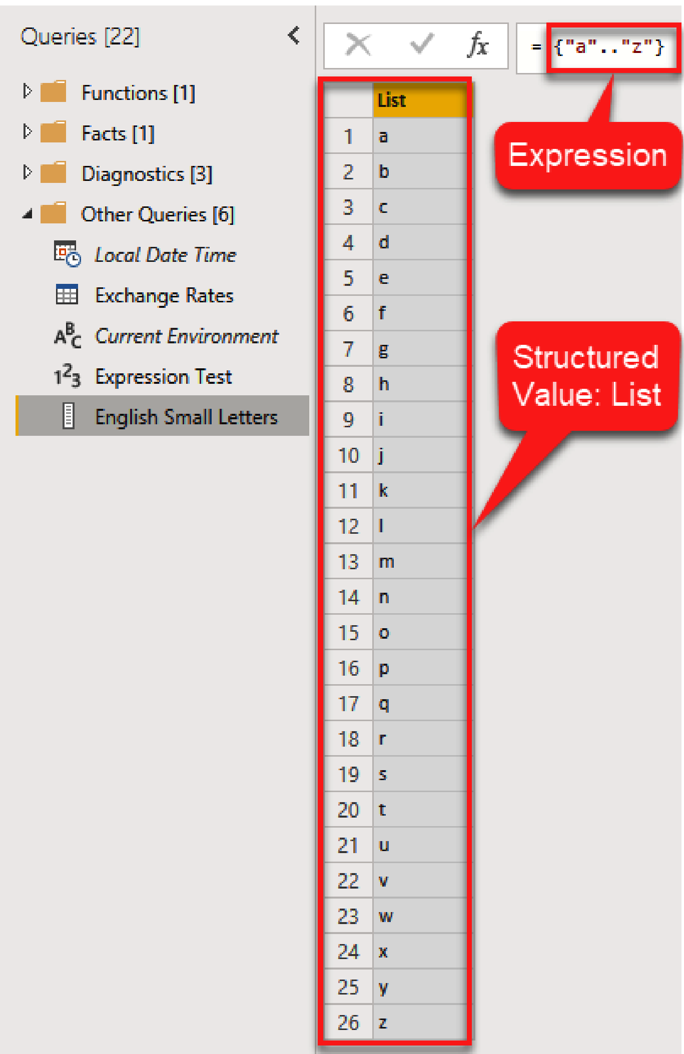


Figure 3.2 – Defining a list of small English letters

- **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:

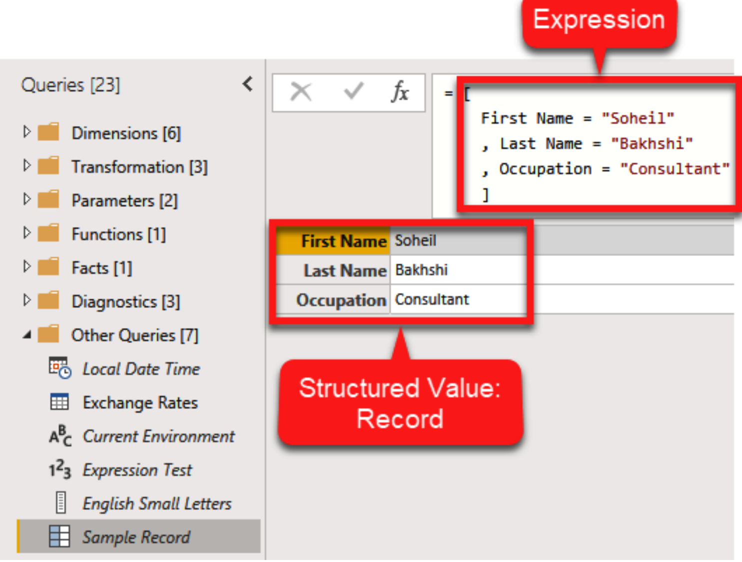


Figure 3.3 – Defining a record in Power Query

> **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):

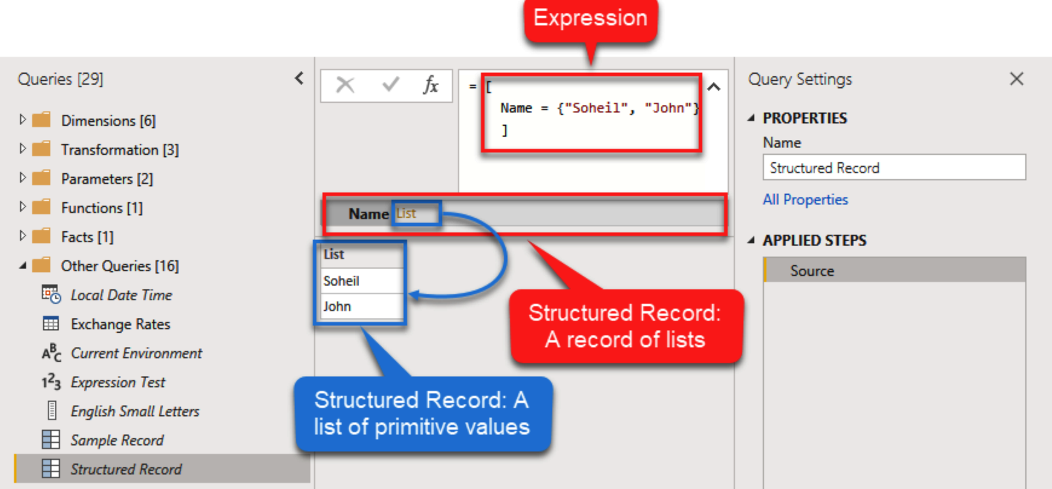


Figure 3.4 – A structured value containing other structures' values

- **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:

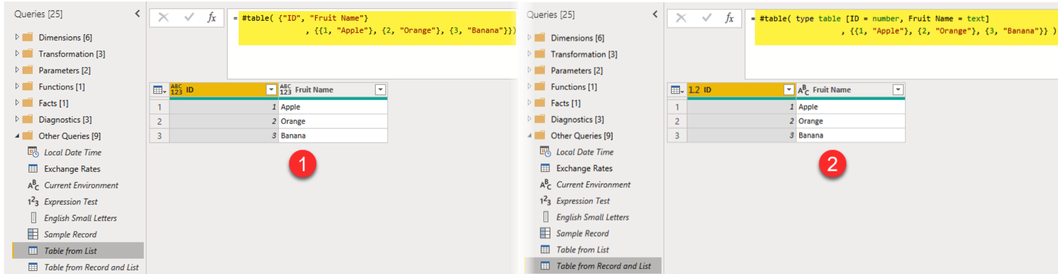


Figure 3.5 – Constructing a table in Power Query

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:

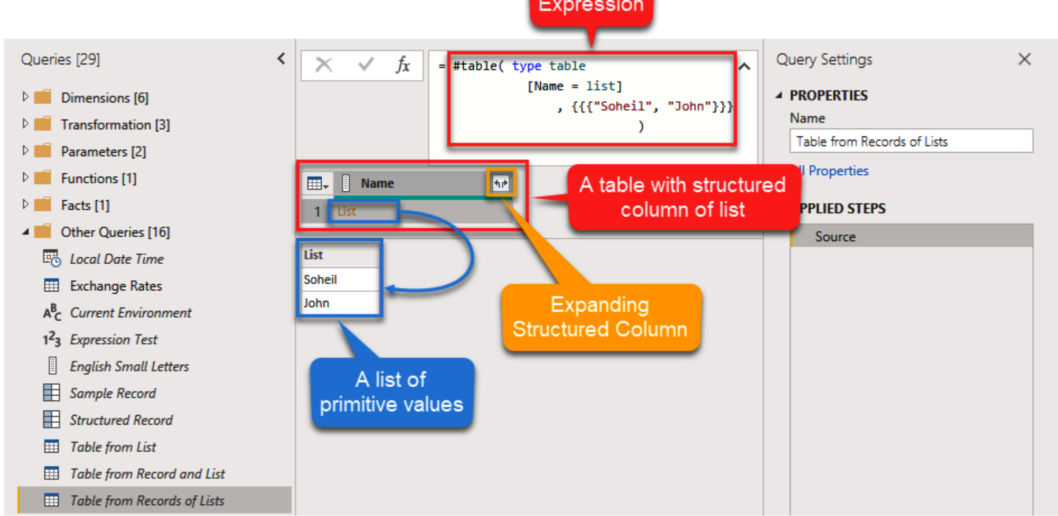


Figure 3.6 – Table with a structured column

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

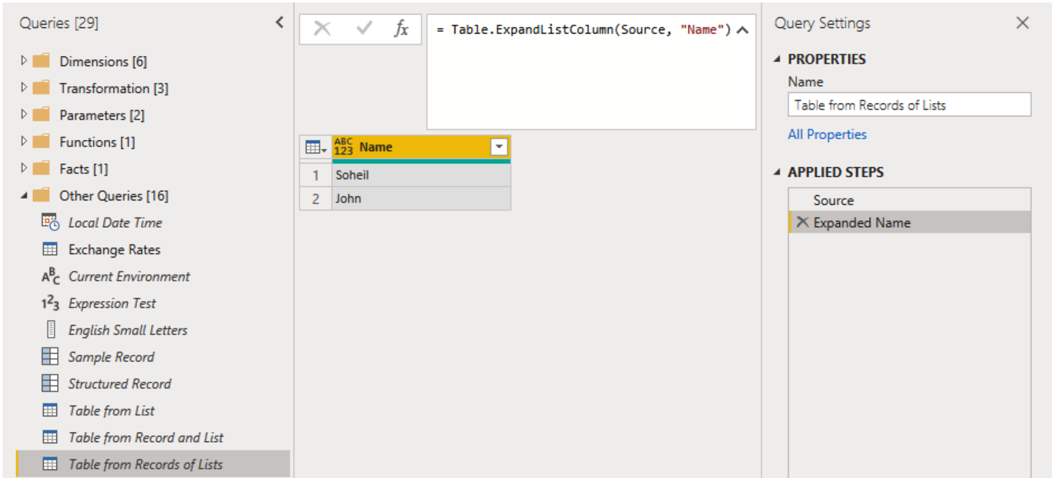


Figure 3.7 – Table with an expanded structured column

- **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):

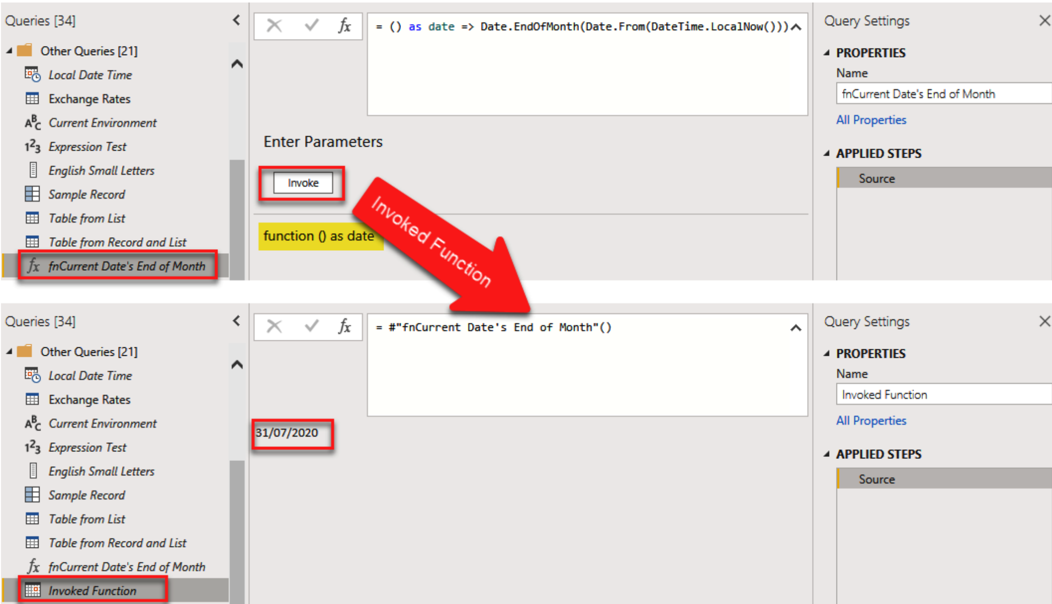


Figure 3.8 – Invoking a custom function

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:

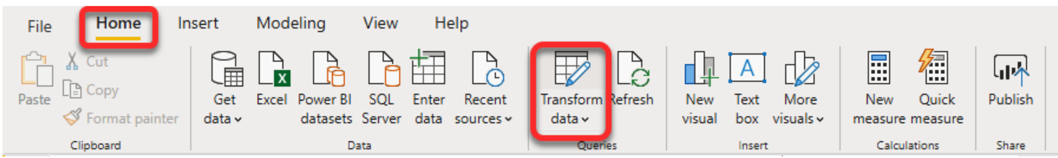


Figure 3.9 – Opening Power Query Editor from the ribbon in Power BI

- 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:

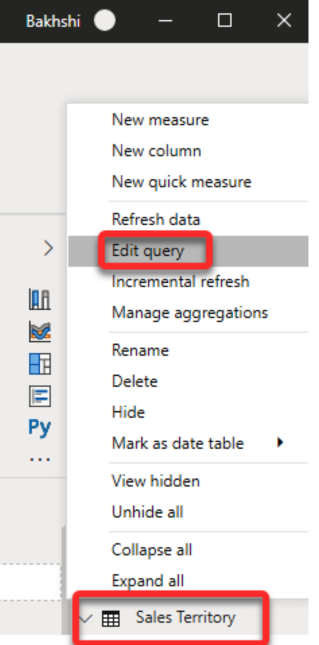


Figure 3.10 – Navigating directly to a specific underlying query in Power Query Editor

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 *fx*.

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:

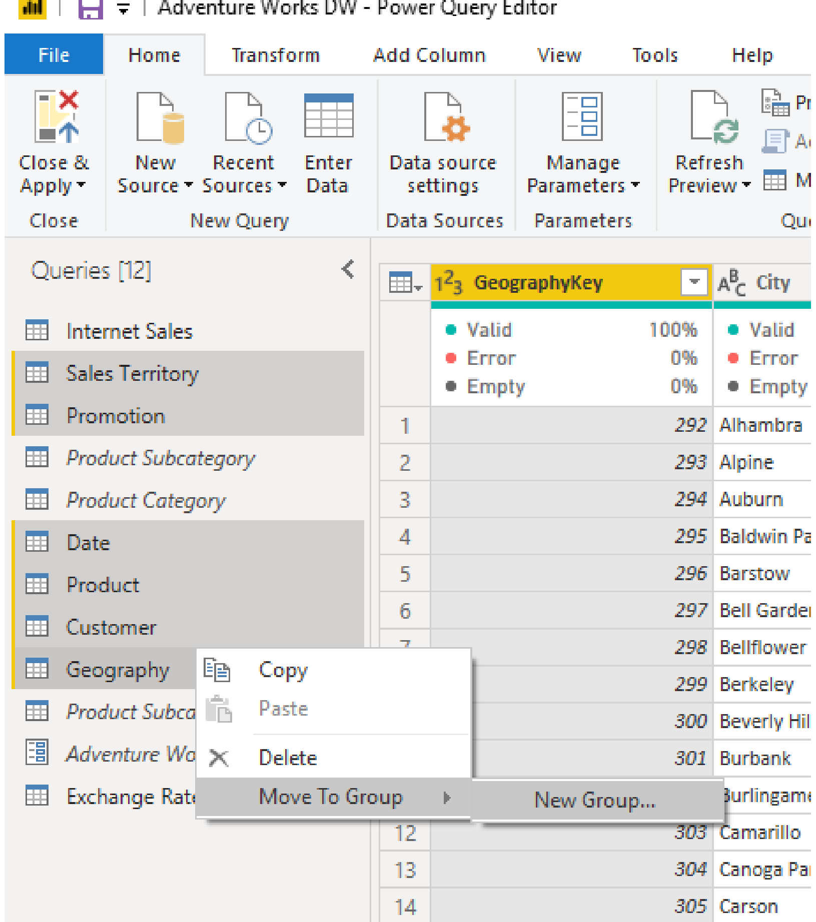


Figure 3.12 – Organizing queries in Power Query Editor

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:

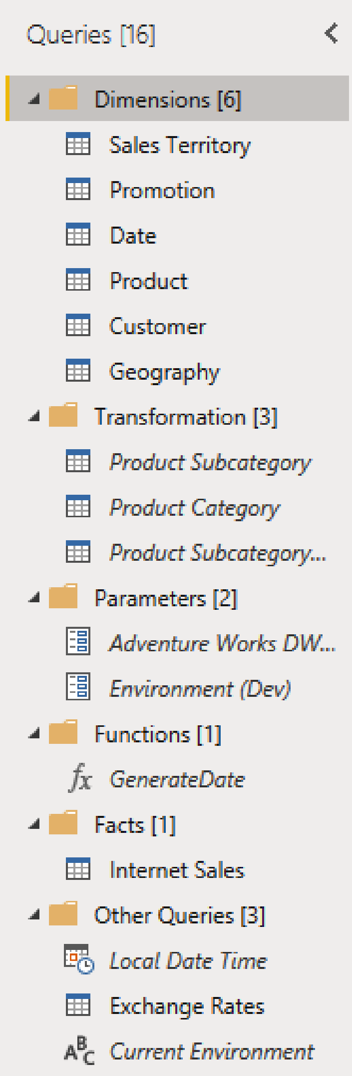


Figure 3.13 – Organized queries in Power Query Editor

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:

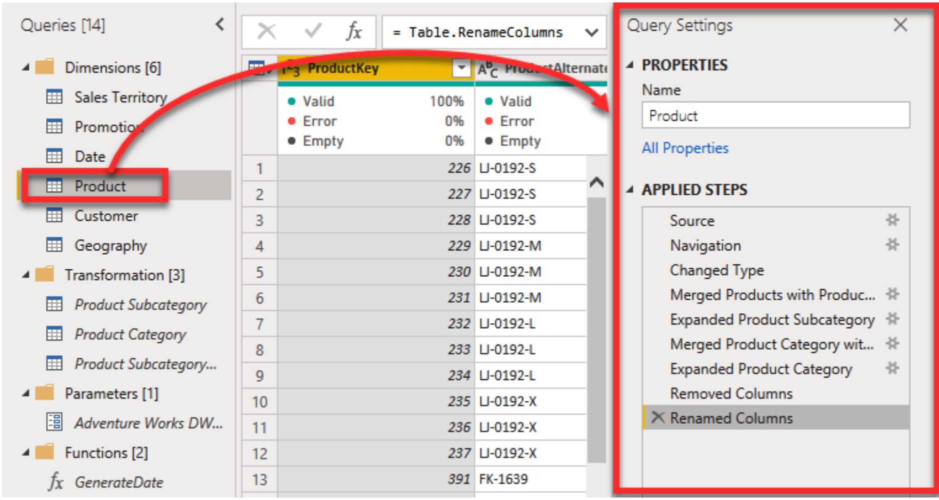


Figure 3.14 – Query Settings pane in Power Query Editor

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:

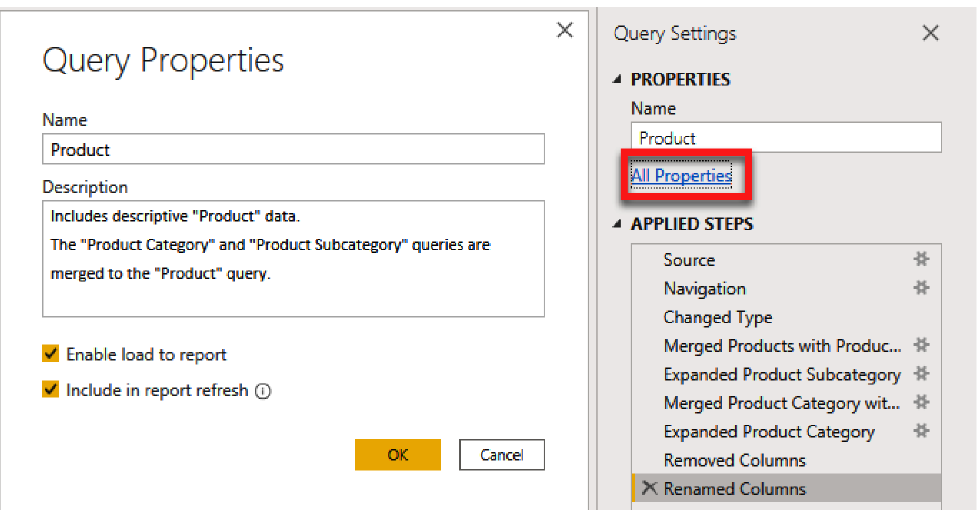


Figure 3.15 – Query Properties from the Query Settings pane

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:

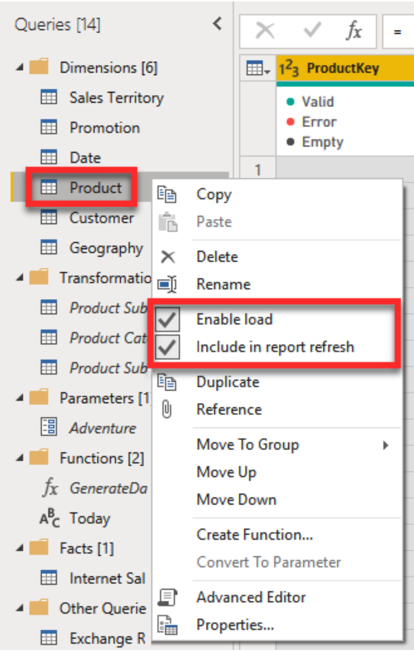


Figure 3.16 – Accessing Enable load and Include in report refresh settings from the Queries pane

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:

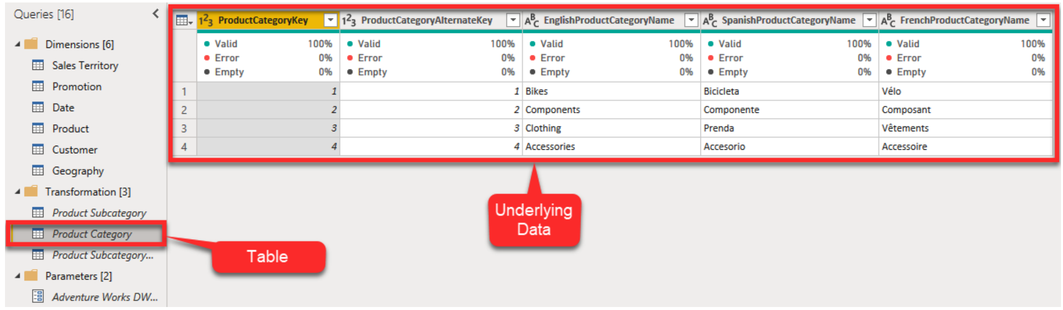


Figure 3.17 – The Data View pane when the selected query from the Queries pane is a table

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

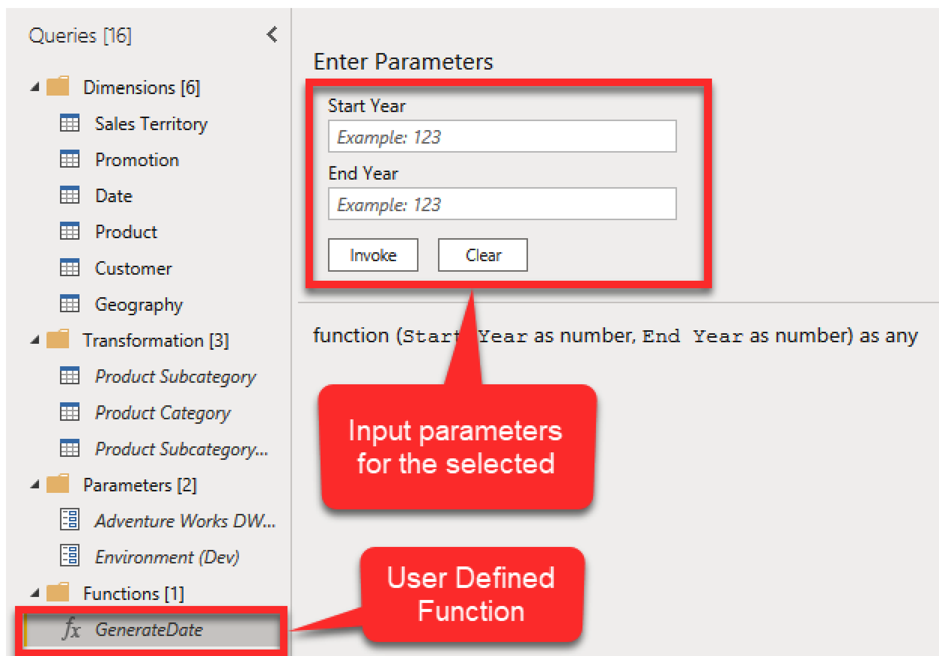


Figure 3.18 – Data View pane when selecting a custom function from the Queries pane

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

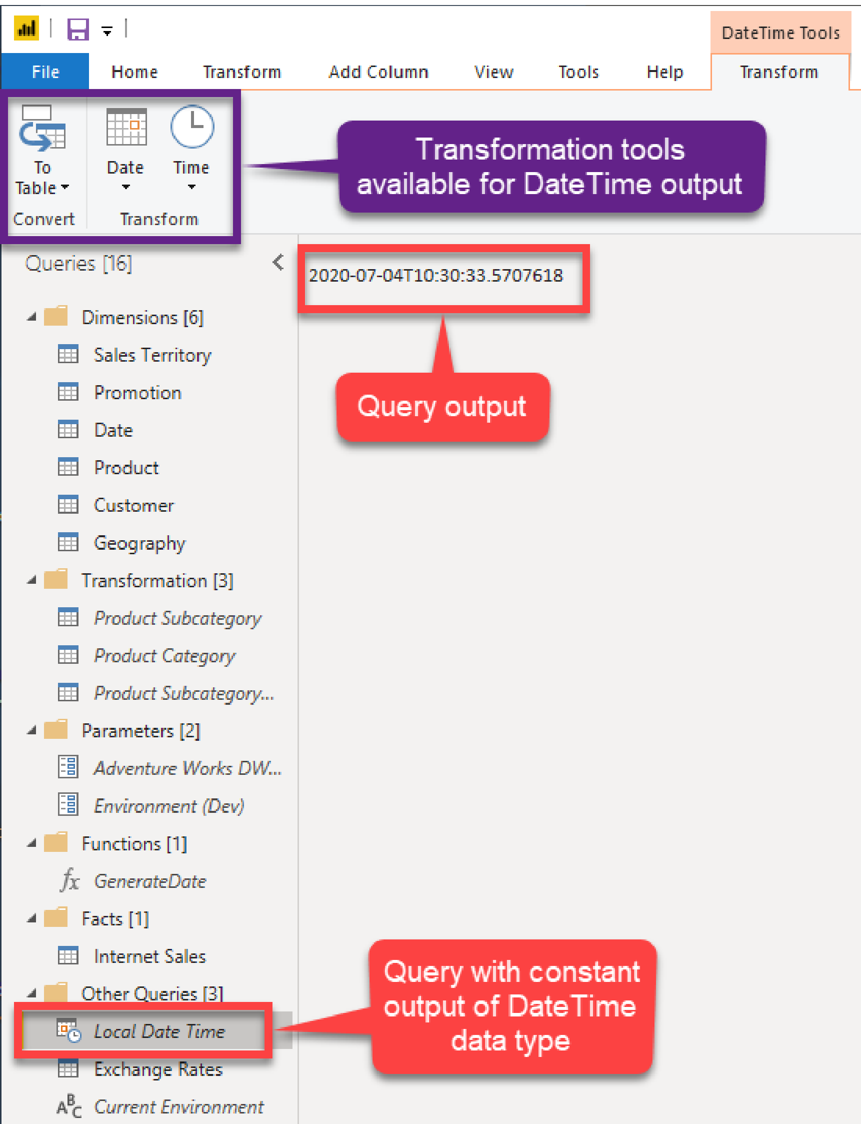


Figure 3.19 – Data View pane when the query output is a constant value

> **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.

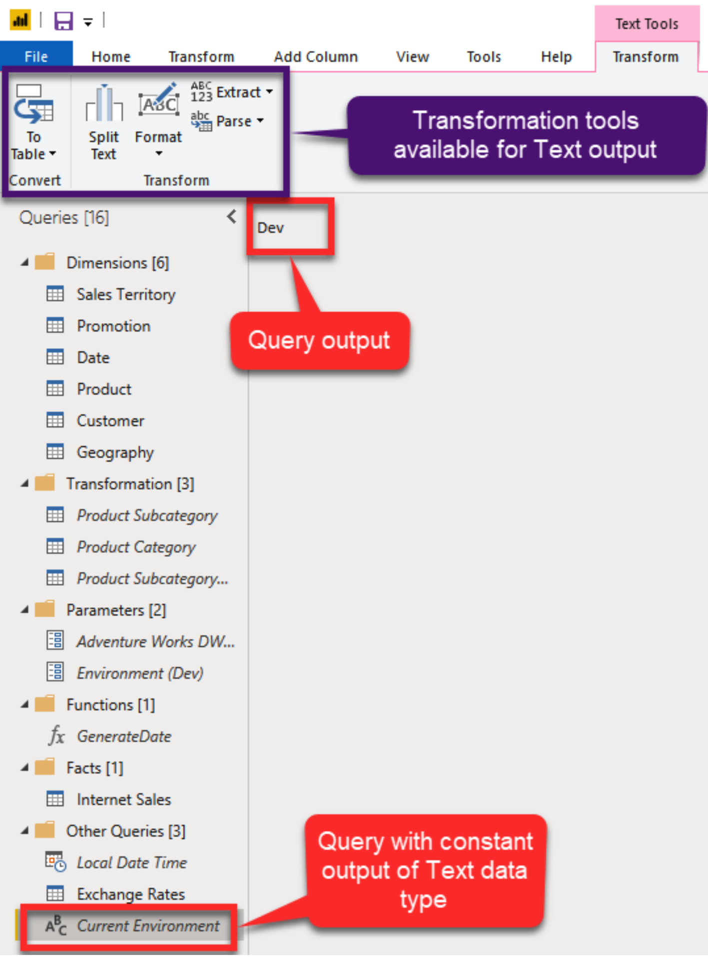


Figure 3.20 – Transformation tools available for a query resulting in a Text value

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:

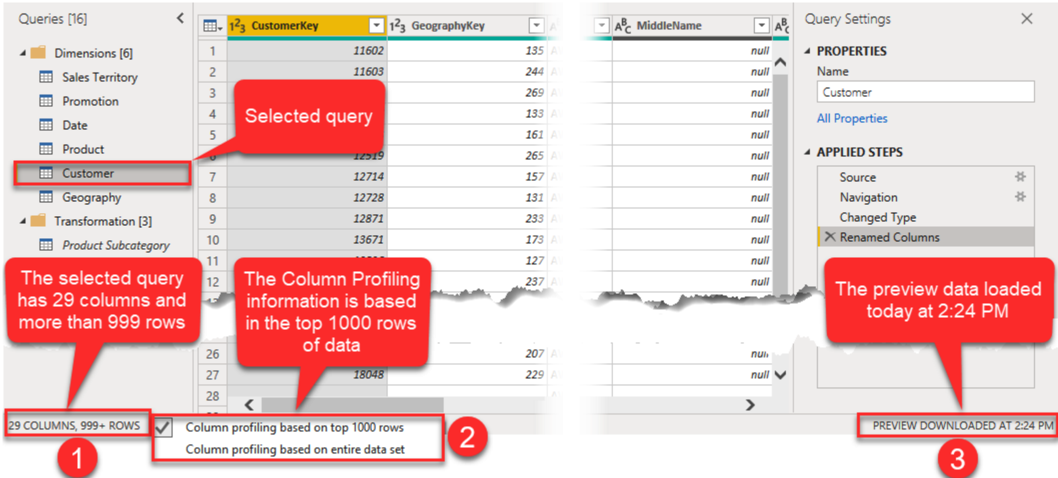


Figure 3.21 – Status bar in Power Query Editor

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:

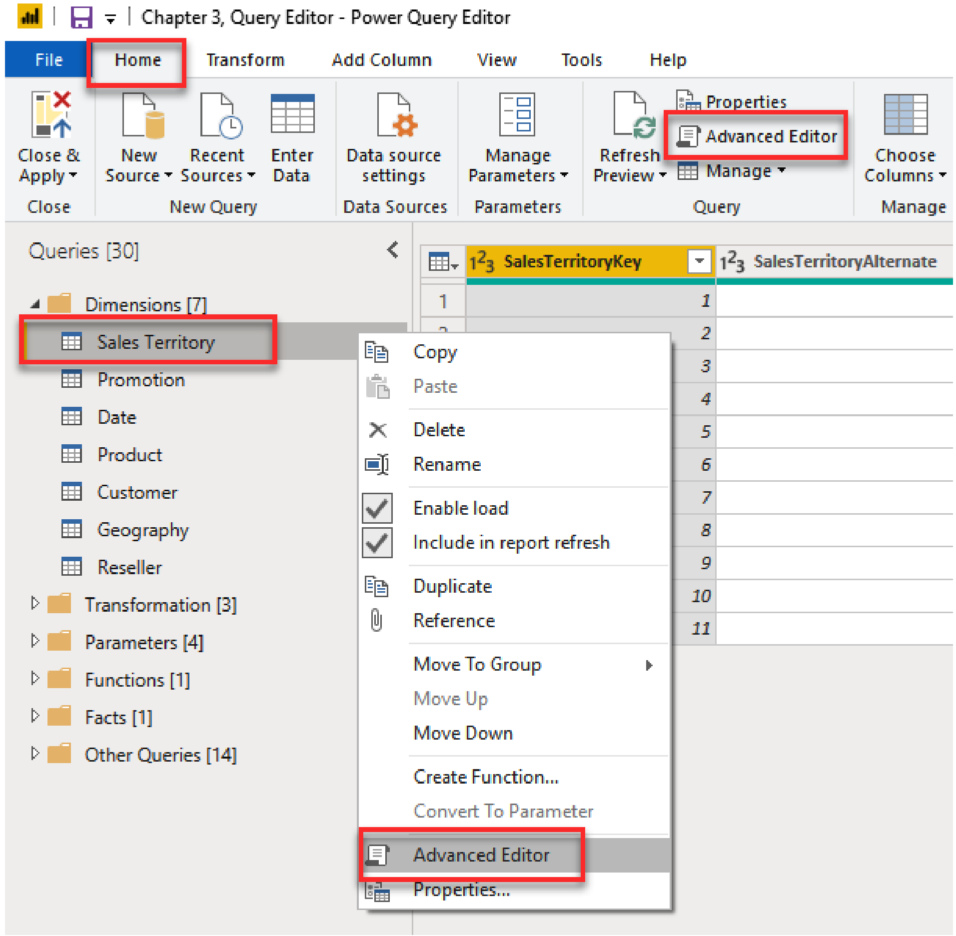


Figure 3.22 – Opening the Advanced Editor

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:

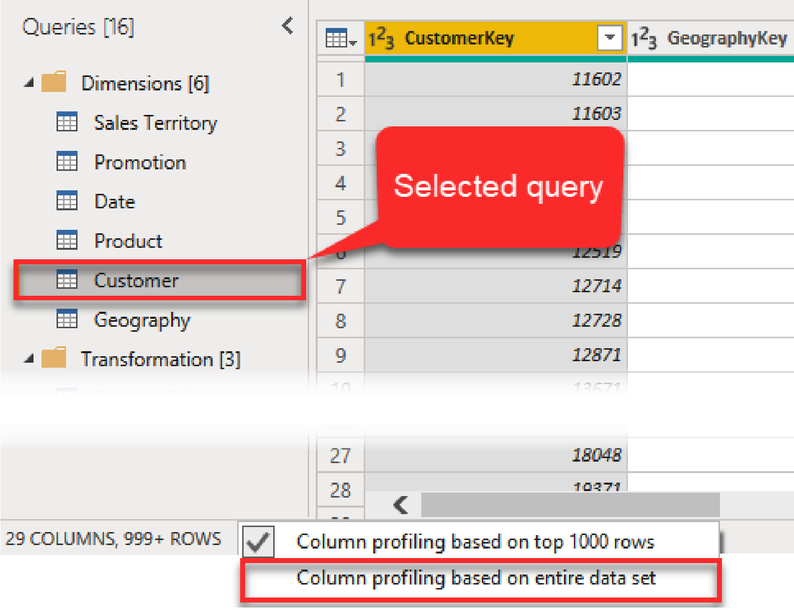


Figure 3.23 – Setting column profiling to be calculated based on the entire dataset

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:

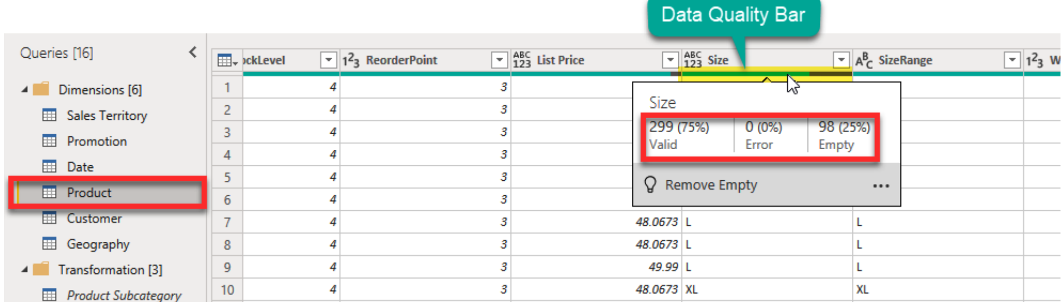


Figure 3.24 – Data Quality Bar in Power Query Editor

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:

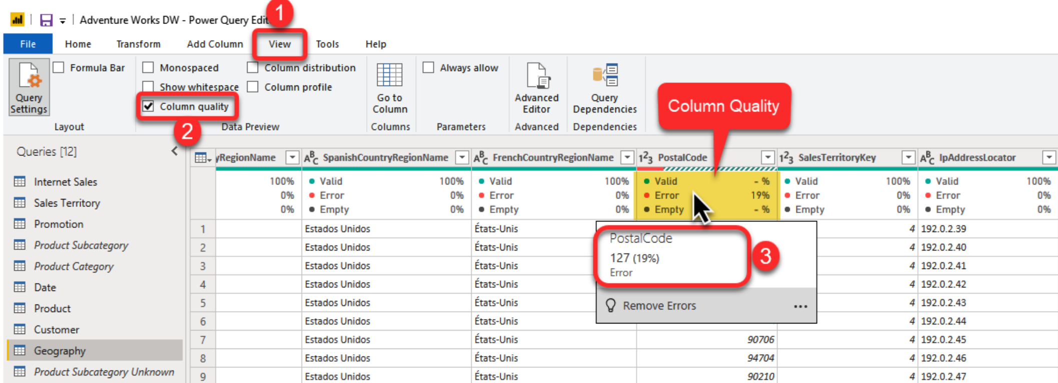


Figure 3.25 – Enabling Column quality in Power Query Editor

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:

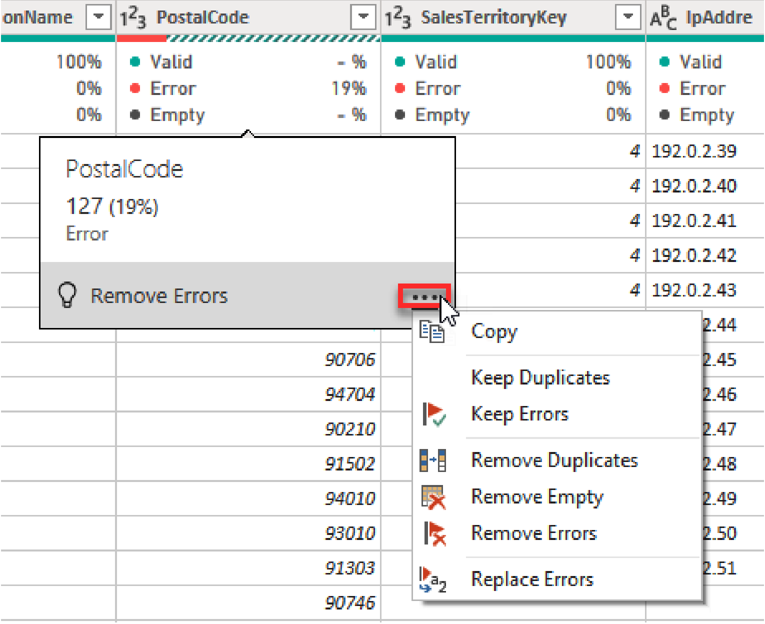


Figure 3.26 – Available options from the Column quality box flyout menu

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:

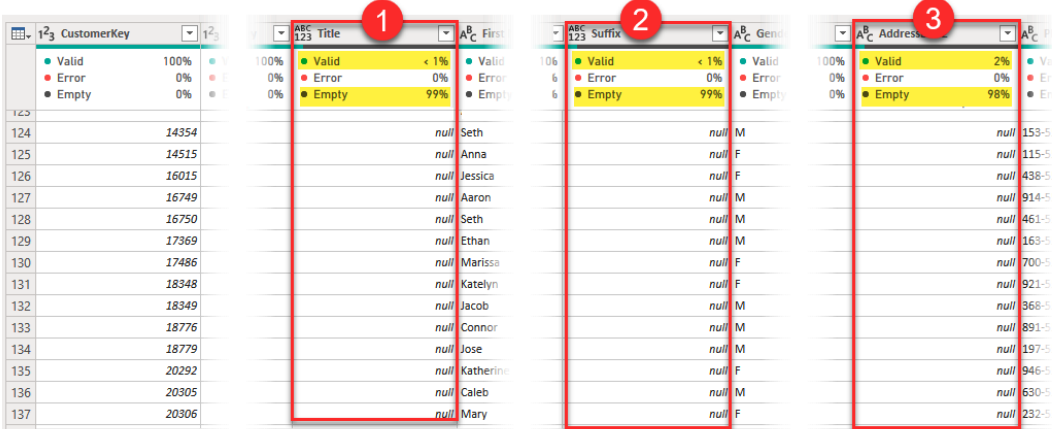


Figure 3.27 – Using Column quality to identify columns with less than 10% valid data

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:

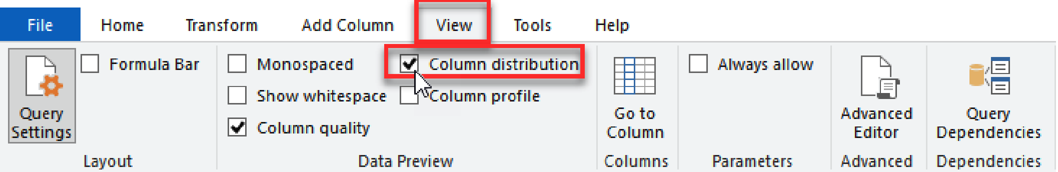


Figure 3.29 – Enabling Column distribution feature from Power Query Editor

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:

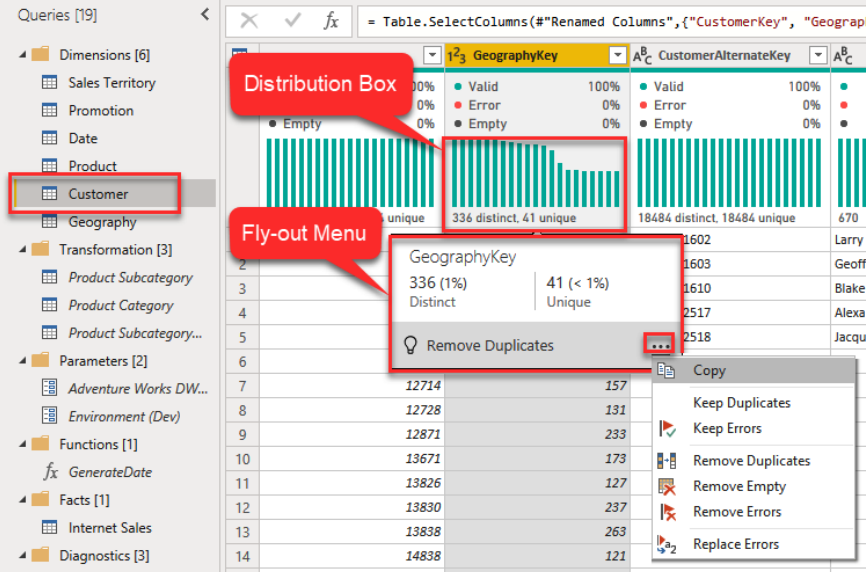


Figure 3.30 – Distribution box flyout menu

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:

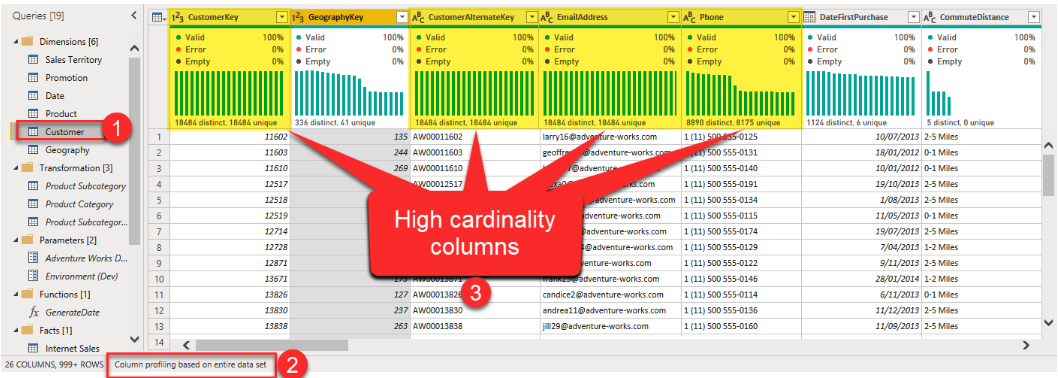


Figure 3.31 – Identifying high-cardinality columns in the Customer table

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:

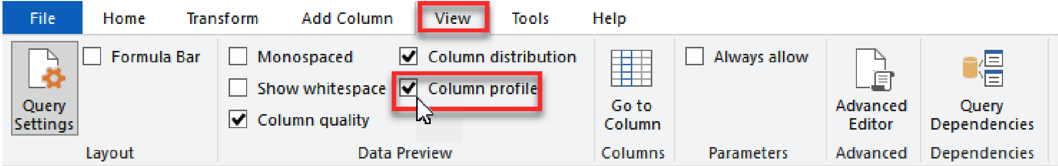


Figure 3.32 – Enabling Column profile from Power Query Editor

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:

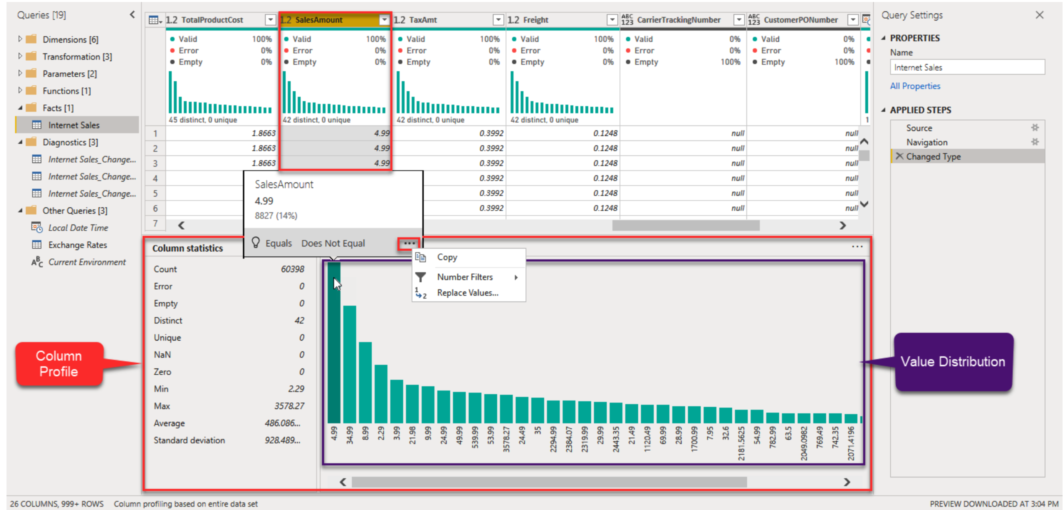


Figure 3.33 – Column profile

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:

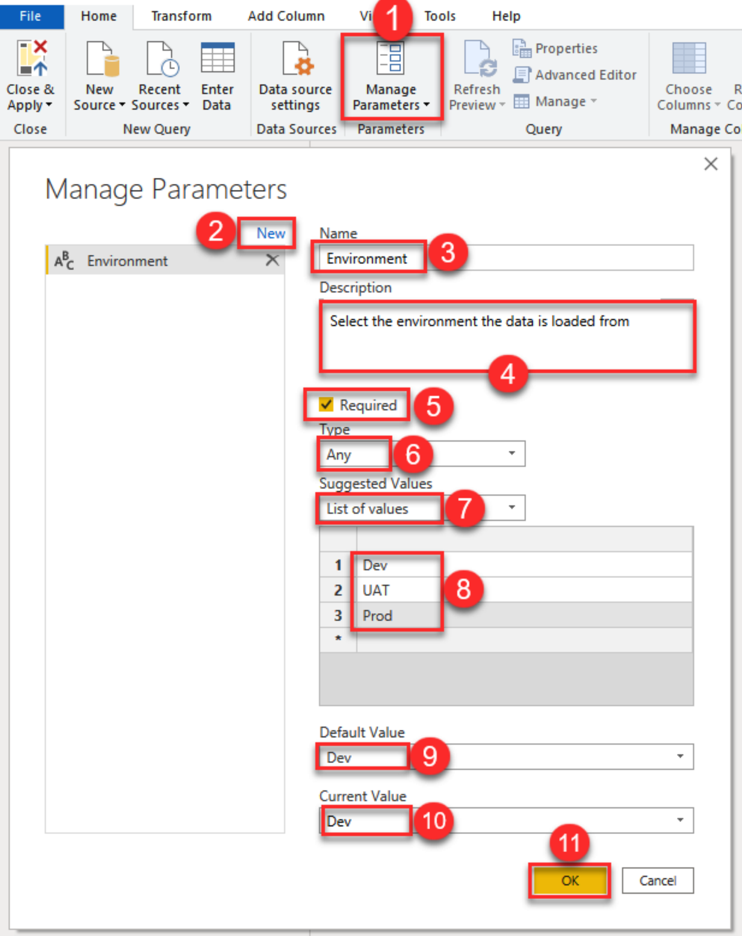


Figure 3.34 – Defining a new query parameter

> **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:

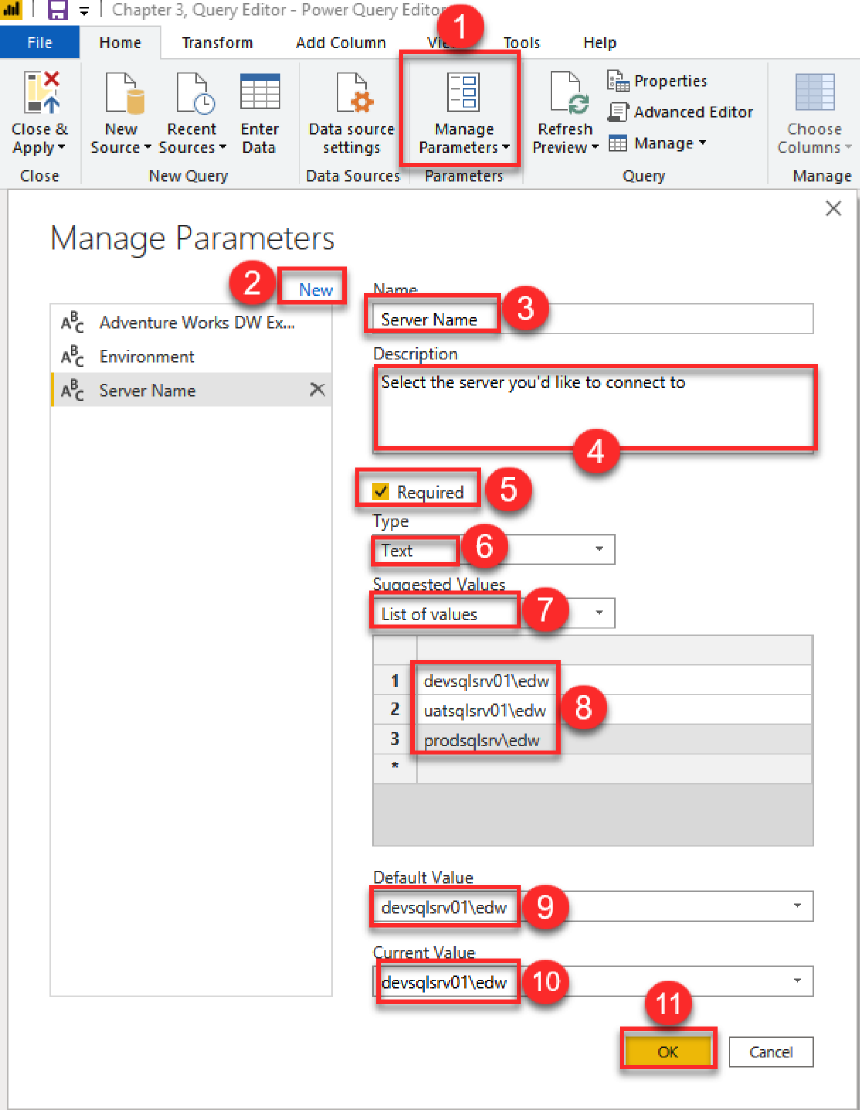


Figure 3.35 – Creating a query parameter holding the server names for different environments

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:

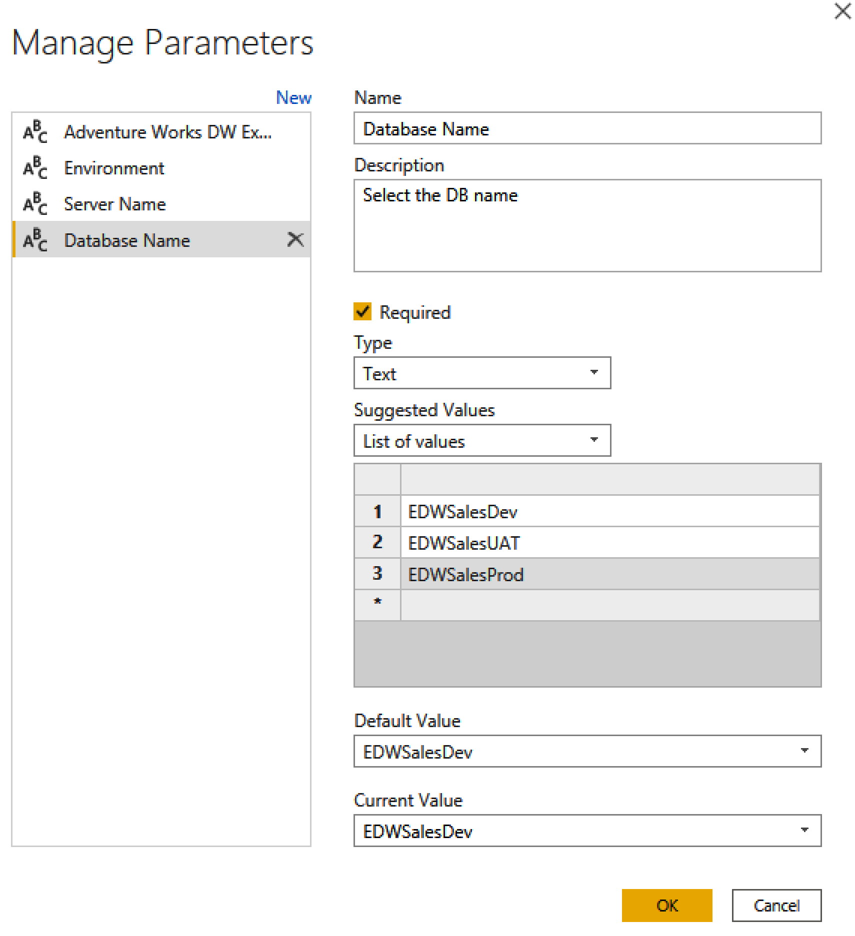


Figure 3.36 – Creating a query parameter holding the database names for different environments

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:

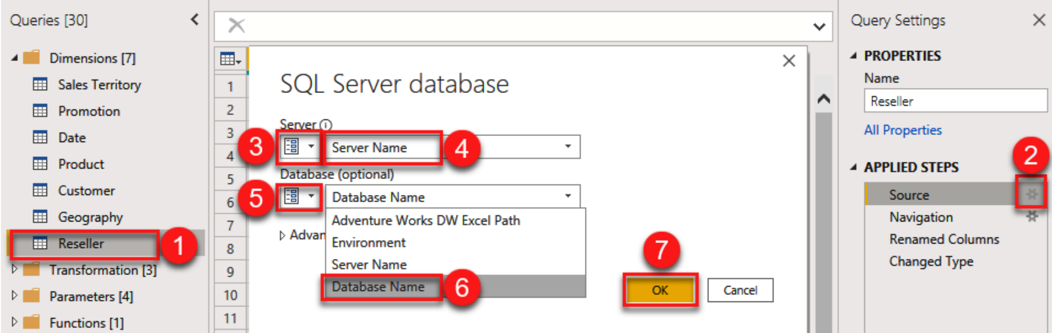


Figure 3.37 – Parameterizing a data source

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:

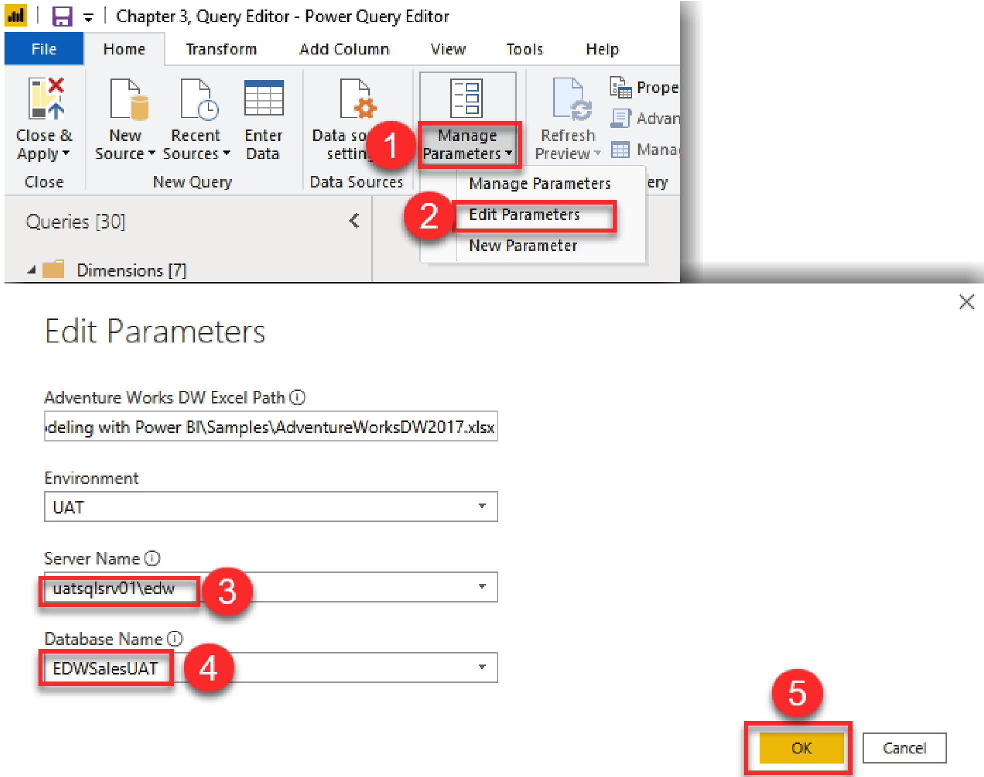


Figure 3.38 – Changing query parameters' values from Power Query Editor

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:

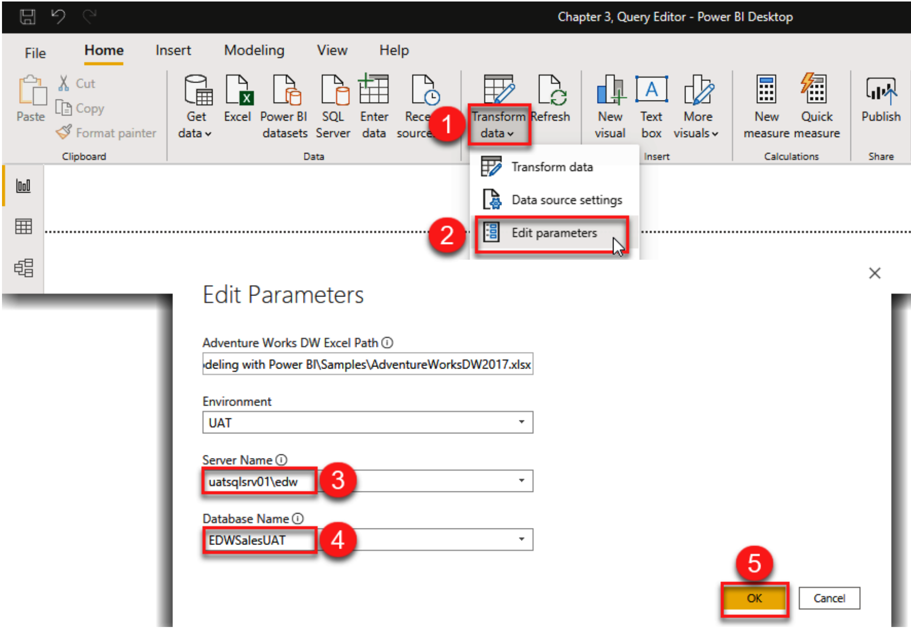


Figure 3.39 – Changing query parameters' values from the main Power BI Desktop window

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:

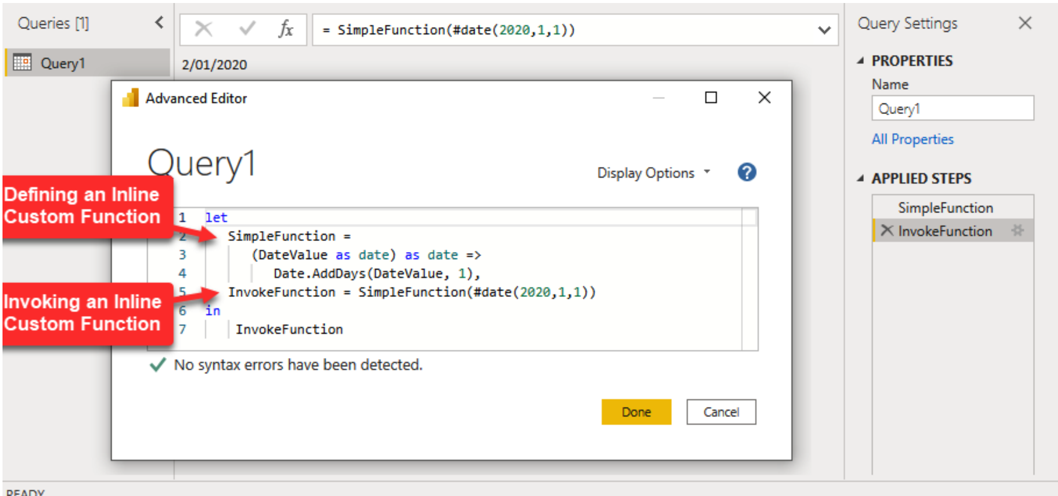


Figure 3.40 – Defining and invoking inline custom functions

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:

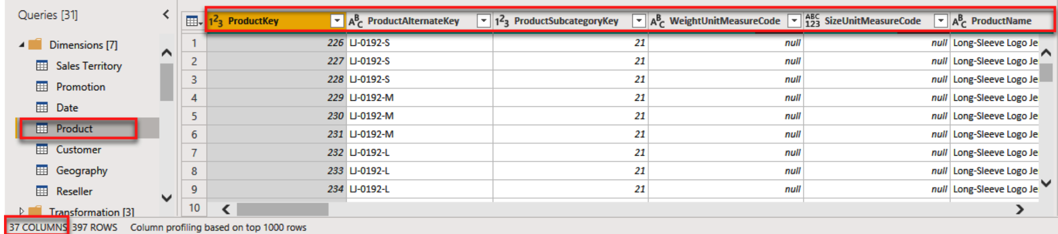


Figure 3.41 – Original column names in the Product table

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:

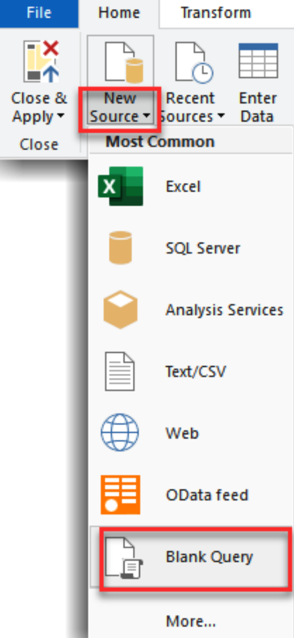


Figure 3.42 – Creating a blank query from Power Query Editor

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**:

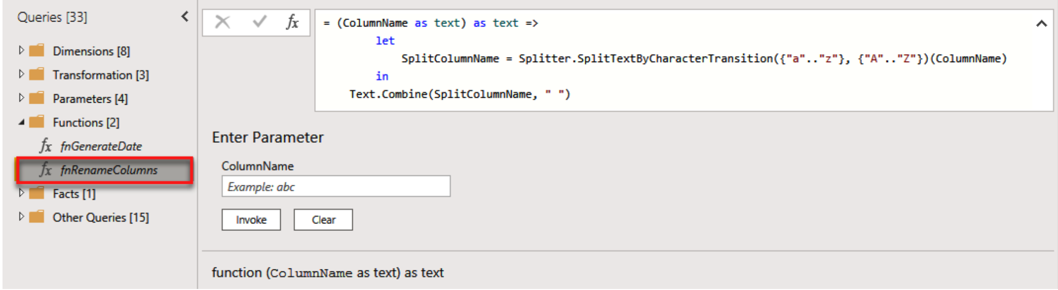


Figure 3.43 – Creating a custom function

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:

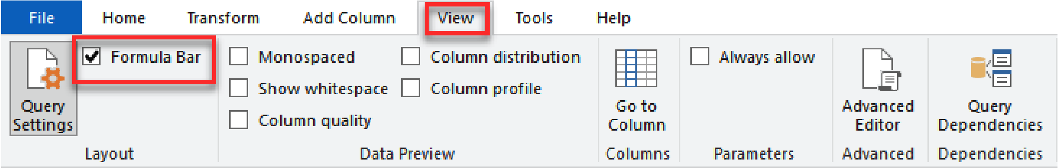


Figure 3.44 – Enabling the Formula Bar option in Power Query Editor

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:

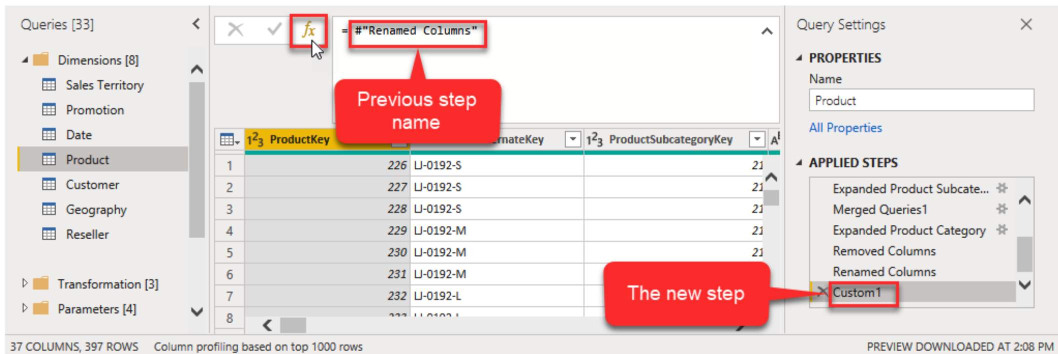


Figure 3.45 – Adding a new step from the Formula Bar

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:

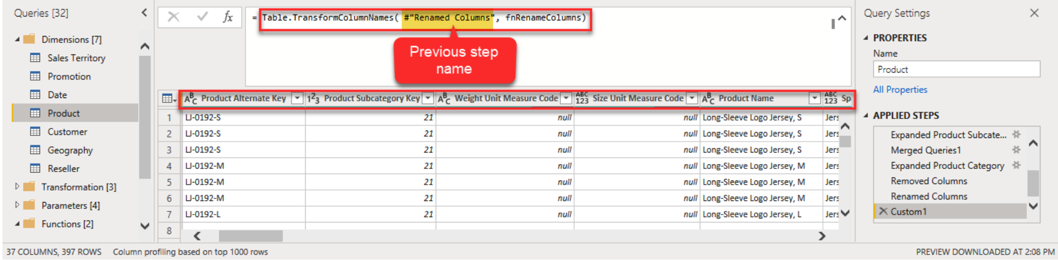


Figure 3.46 – Renaming all columns at once

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:

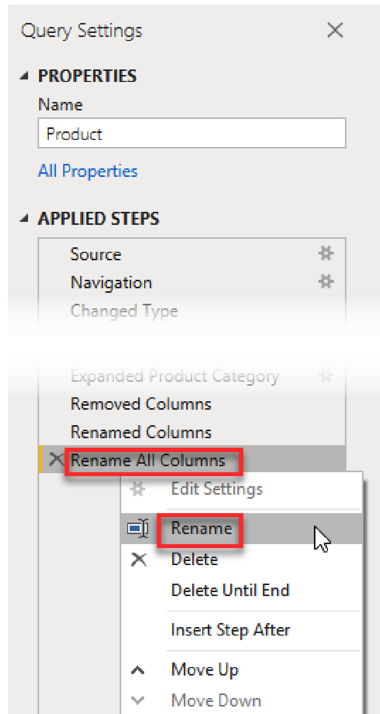


Figure 3.47 – Renaming a query step

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:

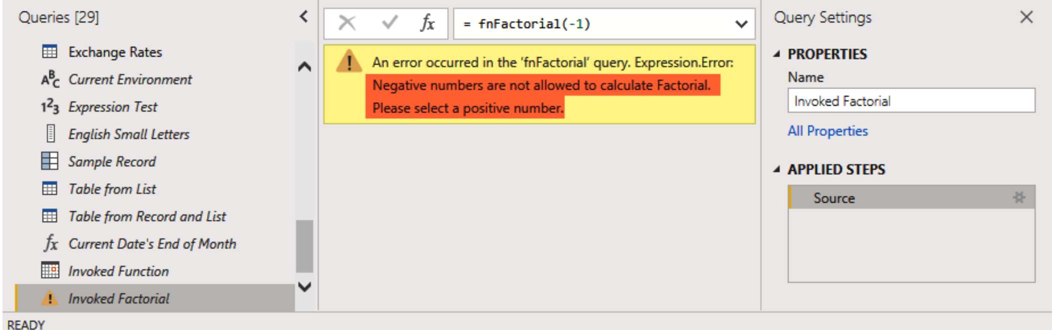


Figure 3.48 – Raising an error when invoking a Factorial function with an invalid value

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

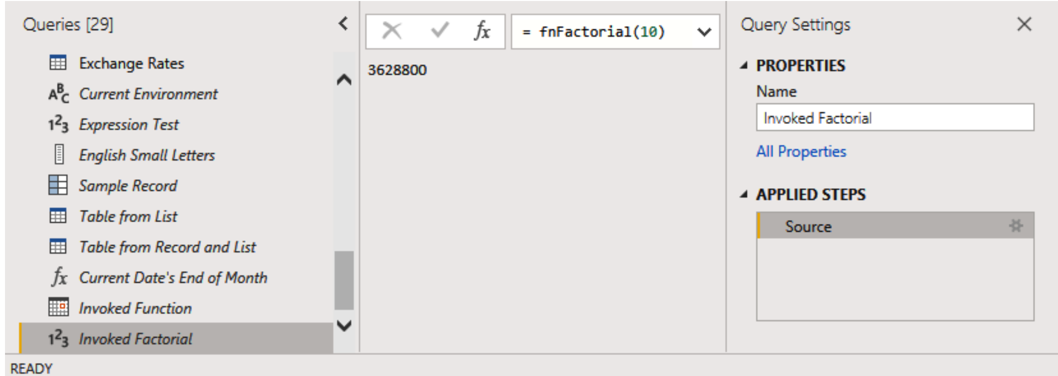


Figure 3.49 – The result of invoking a Factorial function to calculate 10!

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.

4
Getting Data from Various Sources

In the previous chapters, we discussed some aspects of data modeling, including various layers in Power BI, how the data flows between different layers, virtual tables in **Data Analysis Expressions** (**DAX**), and how they relate to data modeling. We also discussed how to leverage the power of query parameters and create custom functions in the Query Editor with the Power Query formula language.

In this chapter, we will learn how to get data from various data sources. We then walk through some known challenges and common pitfalls in getting data from some of those data sources. We will also discuss data source certification and different modes of connection to various data sources.

In this chapter, you will learn about the following topics:

- Getting data from common data sources
- Understanding data source certification
- Working with connection modes
- Working with storage modes
- Understanding dataset storage modes

Getting data from common data sources

With Power BI, we can connect to many different data sources. In this section, we look at some common data sources that we can use in Power BI. We will also look at common pitfalls when connecting to those data sources. But before we start, let's take a moment to discuss a common misunderstanding across many Power BI developers and users on what *get data* means. When we say *get data*, we refer to connecting to a data source from the **Power Query Editor**, regardless of the data source type. Then, we create some transformation steps to prepare the data to be imported into the data model.

While we are working in the **Power Query Editor**, we have not imported any data into the data model unless we click the **Close and Apply** button from the **Home** tab on the ribbon bar (or by clicking the **Apply** drop-down button). Just after we click the **Close and Apply** button, data starts being imported into the data model. At this point, you may ask: *What is the data shown in the* **Data preview** *in the* **Power Query Editor**? The answer is that the data shown in the **Data preview** is only sample data imported from the source system to show how the different transformation steps affect the data. Therefore, we are technically only connected to the data source and we created some transformation steps. Those steps will be applied to the data while importing the data into the data model.

Folder

Getting data from a **folder** is one of the most common scenarios when dealing with file-based data sources. You may have been given a folder containing a mix of altogether different file types, such as Excel, Access, **comma-separated values** (**CSV**), **Portable Document Format** (**PDF**), **JavaScript Object Notation** (**JSON**), a **text** file (**TXT**), and so on. The data structure can also be different, which can make working with the data more complex than it looks. One of the most powerful features of the folder connector in Power Query is that it automatically retrieves all files in the source folder, including those stored in the subfolders. While this is useful in many cases, it can be an issue when we do not want to combine the files stored in subfolders. For those cases, we can filter the results based on the `Folder Path` column to eliminate unwanted data stored in subfolders.

Let's look at the `Folder` data source with a scenario. One of the most common scenarios is when we have Excel files stored in a folder, and the business requires the data held by the Excel files to be analyzed.

Our scenario is that the business needs to analyze the data stored in files exported from an **Enterprise Resource Planning** (**ERP**) system in a folder using an **extract, transform, and load** (**ETL**) tool. The ETL tool generates various file formats as outputs. However, the business only needs to analyze the data from the Excel files. The business archives old data in a folder called `Archive`, which must be excluded from the data model. We are tasked with importing the data from the Excel files stored in the Excel folder, excluding the files stored in the `Archive` folder. The following screenshot shows the folder's structure:

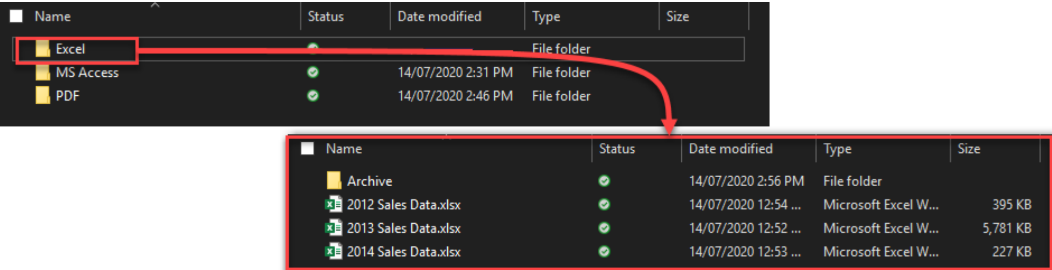


Figure 4.1 – The Excel folder includes an Archive folder that must be excluded from the data model

To achieve our goal, we go through the following steps:

1. In Power BI Desktop, click the **Get data** button.
2. From **All**, click **Folder**.
3. Click **Connect**.
4. Click **Browse…** and navigate to the corresponding folder.
5. Click **OK**.
6. Click **Transform Data**.

The preceding steps are highlighted in the following screenshot:

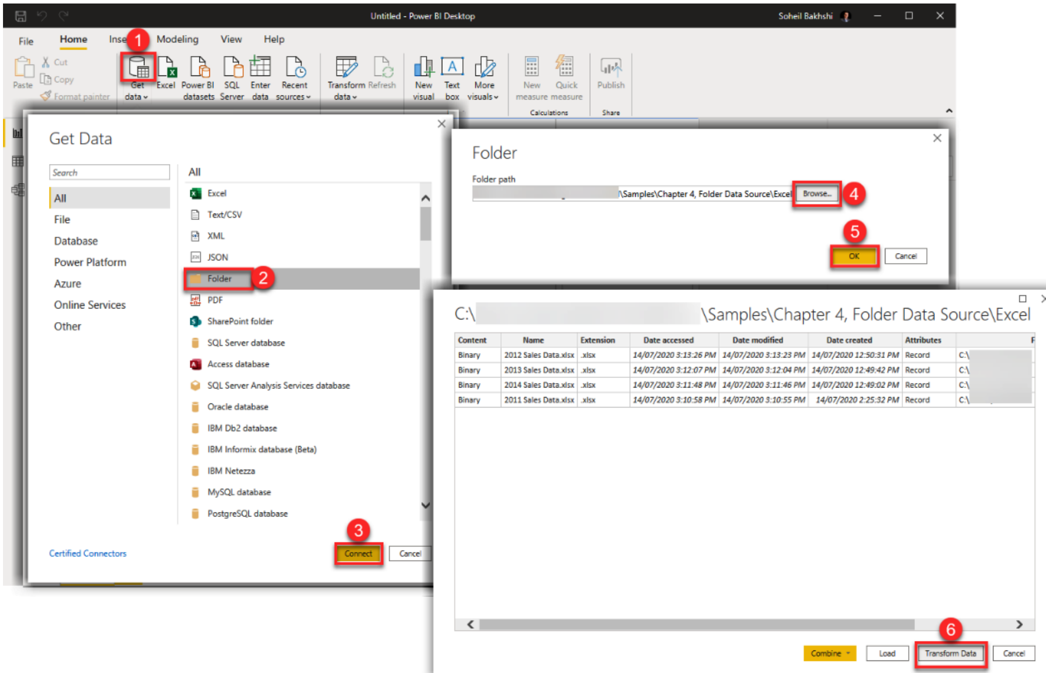


Figure 4.2 – Getting data from the folder

The folder structure illustrated in *Figure 4.1* shows that there are only three Excel files to be analyzed in the source folder. There are four files shown in the result in *Figure 4.2*.

To fix this, we need to filter the results based on the `Folder Path` column, as the following screenshot shows:

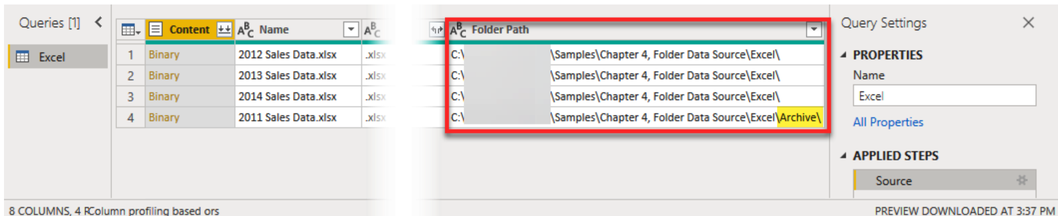


Figure 4.3 – Filtering the results based on the Folder Path column

We should go through the following steps to filter the results:

7. Click on the filter dropdown.
8. Hover over **Text Filters**.

9. Click **Does Not Contain…**.
10. In the **Filter Rows** window, type in `Archive` as that is the subfolder containing data that we want to exclude from the data model.
11. Click the **OK** button.

The preceding steps are highlighted in the following screenshot:

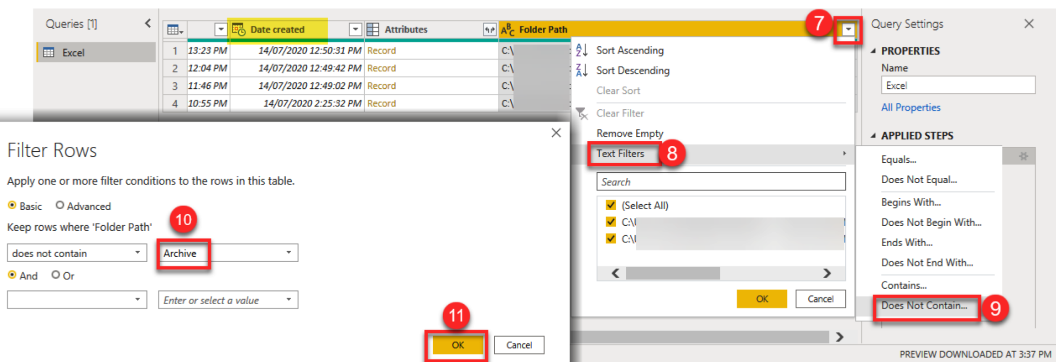


Figure 4.4 – Eliminating the Archive folder

So far, we managed to get the correct files in the query results. The very last step is to combine the contents of the Excel files. To do so, follow these steps:

12. Click the **Combine Files** button on the `Content` column to open the **Combine Files** window, as shown in the following screenshot:

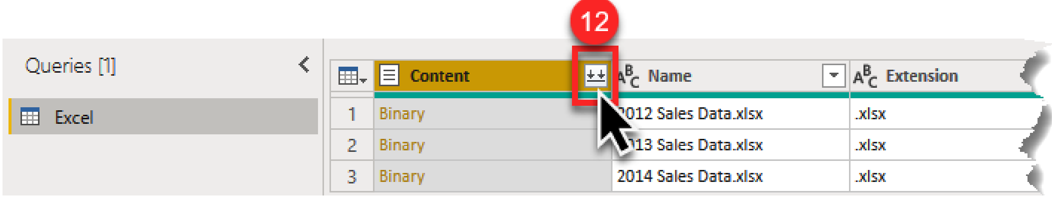


Figure 4.5 – Combining files from column

In the **Combine Files** window, we have the choice to select a sample file. Power BI will use this sample file to create a custom function on the fly to navigate all Excel files. From here, we have the following two options:

- Selecting a table (if any) or a sheet listed under `Parameter1`
- Right-clicking `Parameter1` and then clicking **Transform Data**

Dealing with data stored in tables is generally more straightforward than when it is stored in sheets.

13. Click the `Sales_Data` table.
14. Click **OK**.

The preceding steps are highlighted in the following screenshot:

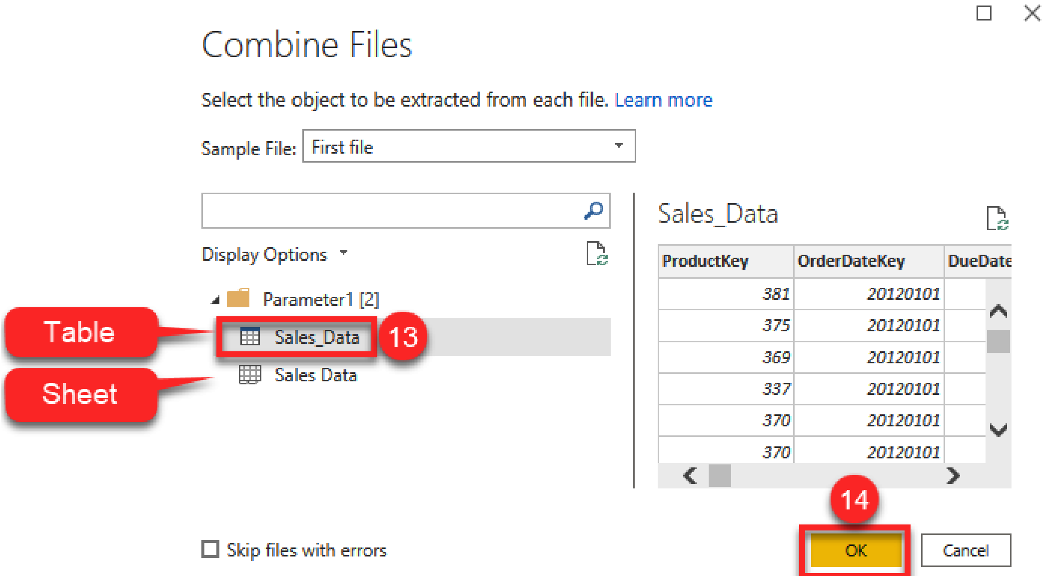


Figure 4.6 – Navigating Excel files to be combined

The preceding steps result in the creation of four new queries in the **Power Query Editor**, as shown in the following screenshot:

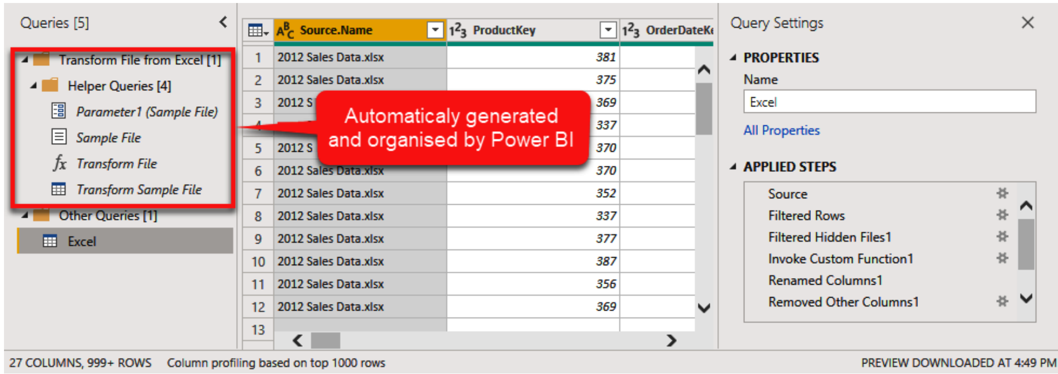


Figure 4.7 – Queries and folders automatically created to combine the files

Let's have a look at the new four queries automatically created by the **Power Query Editor**, as follows:

- `Parameter1`: A query parameter of type `Binary` that is used in the `Transform File` custom function and the `Transform Sample File` query.
- `Sample File`: This query is our original query with one more step navigating to the first binary file, which is the first Excel workbook.
- `Transform File`: A custom function accepting `Parameter1` as an input parameter. Then, based on our choice in the preceding *Step 13*, this navigates through the Excel file, reading the data either from a sheet or from a table (in our case, it is a table).

  `Transform Sample File`: A sample query to open the Excel file using `Parameter1`. The sample query is disabled to load into the data model.

  In some cases, we need to keep some files' metadata, such as `Date Created` (which you can see in *Figure 4.4*). There is some more detailed metadata stored in the `Attribute` column, which is a structured column of records. We leave this to you to investigate more. But for this scenario, we would like to keep the `Date Created` column for our future reference. We now need to modify an automatically created step, as follows:

15. Click the gear icon on the right side of the **Removed Other Columns1** step.
16. Tick the `Date Created` column.
17. Click **OK**.

The preceding steps are highlighted in the following screenshot:

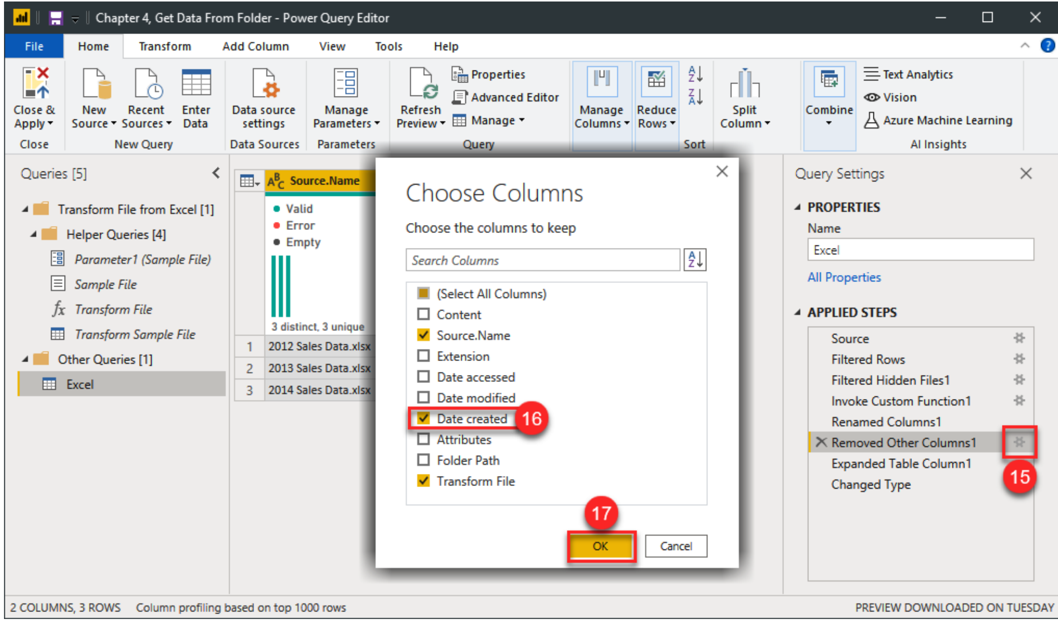


Figure 4.8 – Modifying the automatically created steps to keep some columns

We have successfully combined data stored in different Excel files stored in a folder. As you can see in the following screenshot, we can quickly find out which row is loaded from which file. We can also see when that Excel file was created, which comes in handy for people supporting the report in the future. A good use case is to track back errors and find out which Excel file is troublesome:

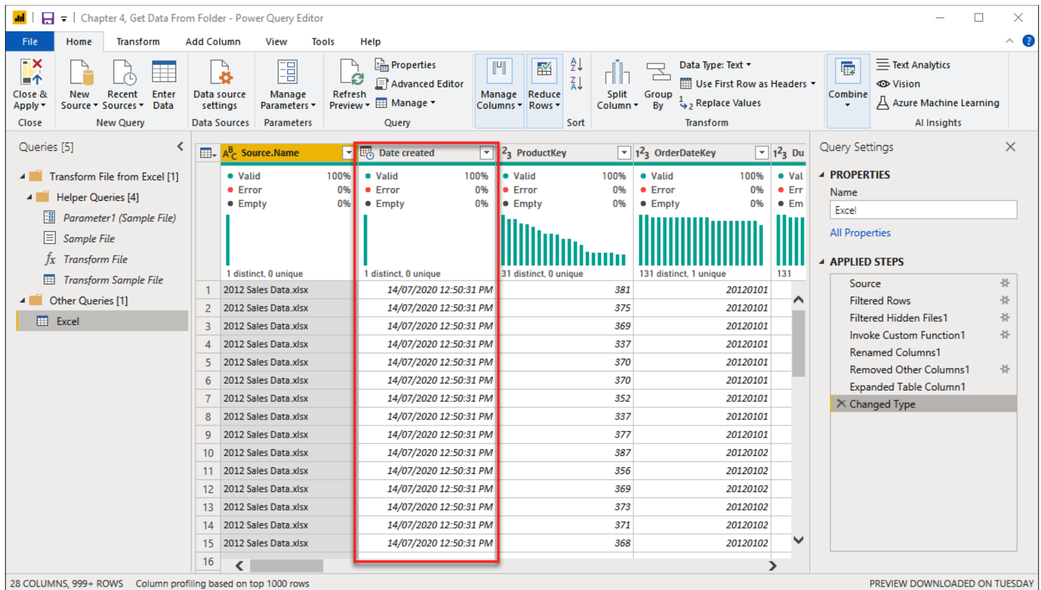


Figure 4.9 – The results of combining Excel files stored in a folder

Finally, we rename the `Excel` table to `Sales`.

CSV/Text/TSV

While there is a dedicated connector for CSV and Txt files in Power BI, no specific connectors support **tab-separated values** (**TSV**) files. In this section, we will quickly look at those three file types. Let's continue with a scenario.

In the previous sections of this chapter, we managed to import sales data stored in a folder. Now, the business received some data dumps stored in various formats. We have `Product` data in CSV format, `Product Subcategory` data in TSV format, and `Product Category` data in Txt format. We need to import the data from the files into Power BI. We use the `Chapter 4, Get Data From Various Sources.pbix` file.

The following screenshot shows the files to import to Power BI:

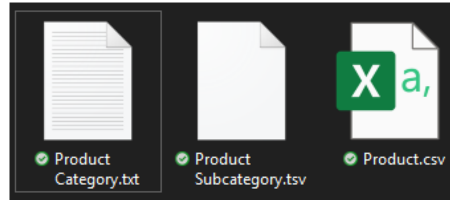


Figure 4.10 – CSV, Txt, and TSV files to be imported to Power BI

We start by getting data from the `Product Category` file, as follows:

1. In **Power Query Editor**, click the **New Source** drop-down button.
2. Click **Text/CSV**.
3. Navigate to the folder holding sample files and select the `Product Category.txt` file.
4. Click **Open**.
5. Power BI correctly detected the **File Origin** and the **Delimiter**, so click **OK**.

The preceding steps are illustrated in the following screenshot:

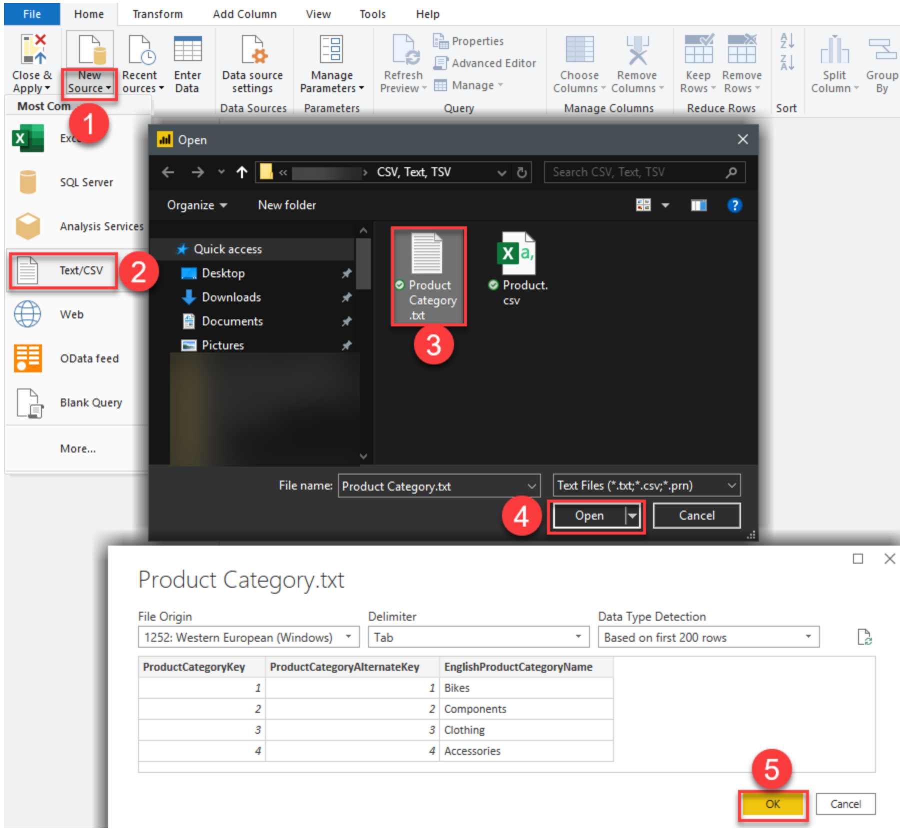


Figure 4.11 – Getting data from a Txt file

We are successfully connected to the `Product Category.txt` file from the **Power Query Editor**.

With that, let's go ahead and get the `Product Subcategory` data, which is stored in a TSV file. As mentioned earlier in this section, there are currently no dedicated connectors for TSV files in Power BI. Nevertheless, that does not mean we cannot get the data from TSV files. As the name suggests, TSV files are indeed for storing data in text format, which is tab-delimited. Therefore, we must get the data from the TSV files using the **Text/CSV** connector. To do this, proceed as follows:

1. Click the **New Source** drop-down button again.
2. Click **Text/CSV**.
3. Navigate to the folder holding sample files, and select `All files (*.*)` from the file-type dropdown.
4. Now, the `Product Subcategory.tsv` file shows up, so select this file.
5. Click **Open**.
6. Power BI correctly detected the **File Origin** and the **Delimiter**, so click **OK**.

The preceding steps are highlighted in the following screenshot:

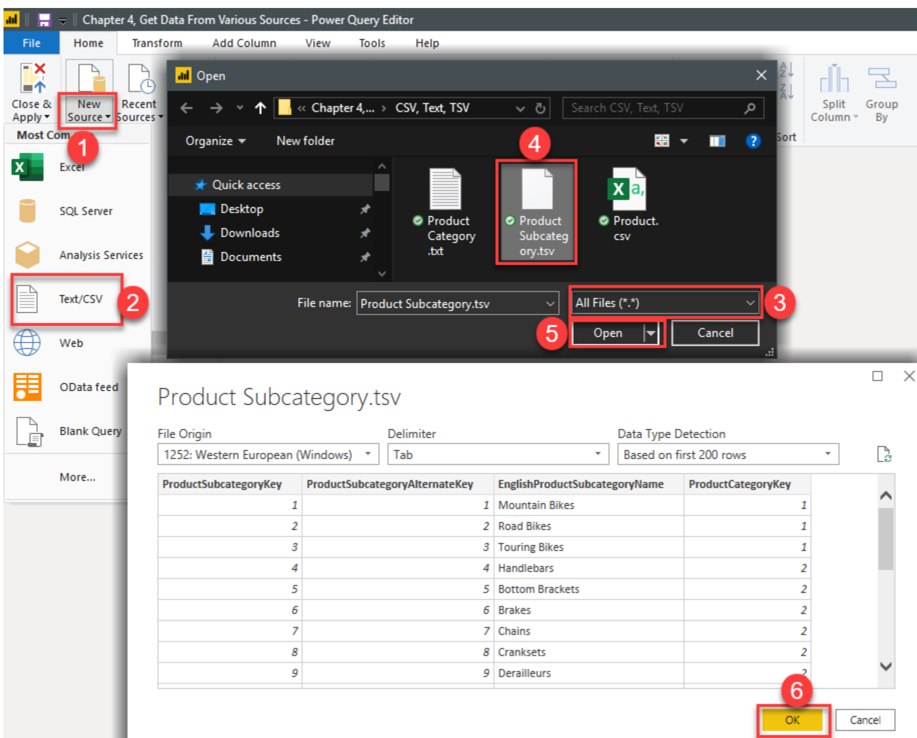


Figure 4.12 – Getting data from a TSV file

So far, we quickly got data from text-based data sources in Txt and TSV file formats, and everything went quite smoothly. But that is not always the case. In reality, we may face many issues when dealing with any file-based data sources that are not curated. Let's look at one of the most common challenges we may face while dealing with text-based data. Now that we have got the `Product Category` and `Product Subcategory` data, it's time to move forward and get the `Product` data stored in `CSV` format. The process is the same as how we got the text data, so we do not repeat those steps.

The quality bar of the `ProductKey` column in *Figure 4.13* reveals some errors in that column. However, the `Product` table has only **607** rows, as shown in the status bar. Note that an error has occurred within the sample data. On other occasions, errors might not be evident until we import the data into the data model. In our case, we can scroll down through the data preview to find a cell that reproduces an error that happens to be right after where the `ProductKey` is `226`. We then click on the erroneous cell to see the error message.

Looking at the data more thoroughly reveals some other issues. There is an incorrect date value in the `ProductAlternateKey` column. The other issue is that the data type of the `ProductSubcategoryKey` column is also incorrect. The reason is trivial; there is also a date value for the `ProductSubcategoryKey` column that is supposed to be of type `number`. While we can remove the errors in some cases, the data looks to be legitimate in our case, but it shows up in an incorrect column. All this is illustrated in the following screenshot:

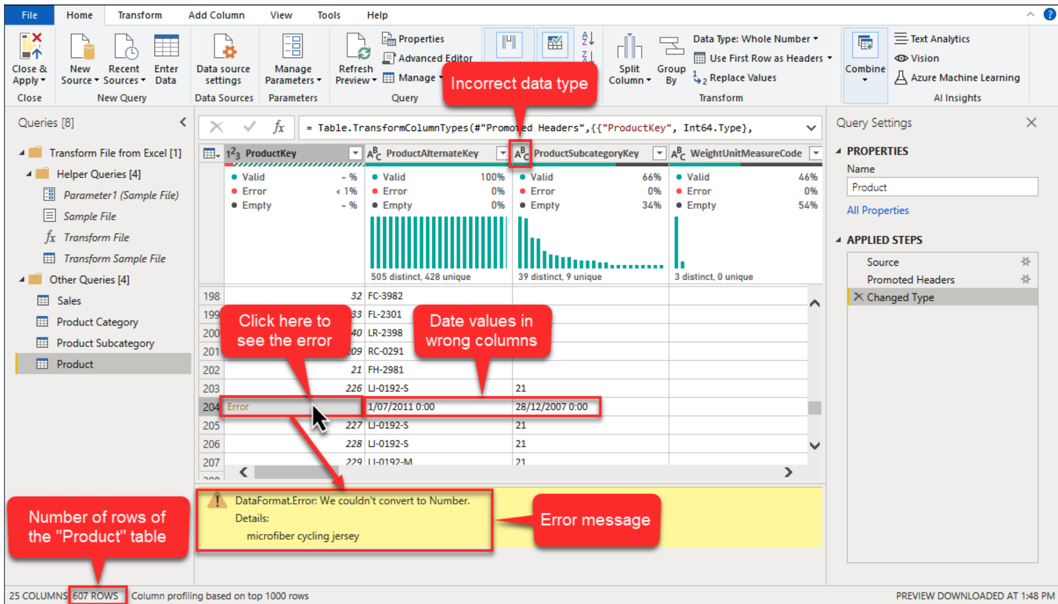


Figure 4.13 – Getting data from CSV-produced errors

Let's have a look at the data in more detail. While we have only **607** rows of data, it makes sense to open the CSV file in a text editor such as Notepad++.

> **Note**
>
> You can download Notepad++ for free from here: `https://notepad-plus-plus.org/downloads/`.

Within Notepad++, we enable **Show All Characters**, and then we search the file where the `ProductKey` value is `226`. So, it is now more obvious what has happened. For some reason, a **carriage return** (**CR**) and a **line feed** (**LF**) have been entered within the text.

When we investigate more, it turns out that the `EnglishProductDescription` column has those characters. This issue can happen when the user presses the *Enter* key from the keyboard while typing the product description into the source system. Then, the exact data is exported from the source system into CSV. This issue may happen in any other column with type text. Some other issues in the data shown in the following screenshot are not trivial when we look at the data. Those issues did not produce any errors in the **Power Query Editor** either, so they can be a bit hard to spot. However, they could potentially cause some issues when we reconcile the data later:

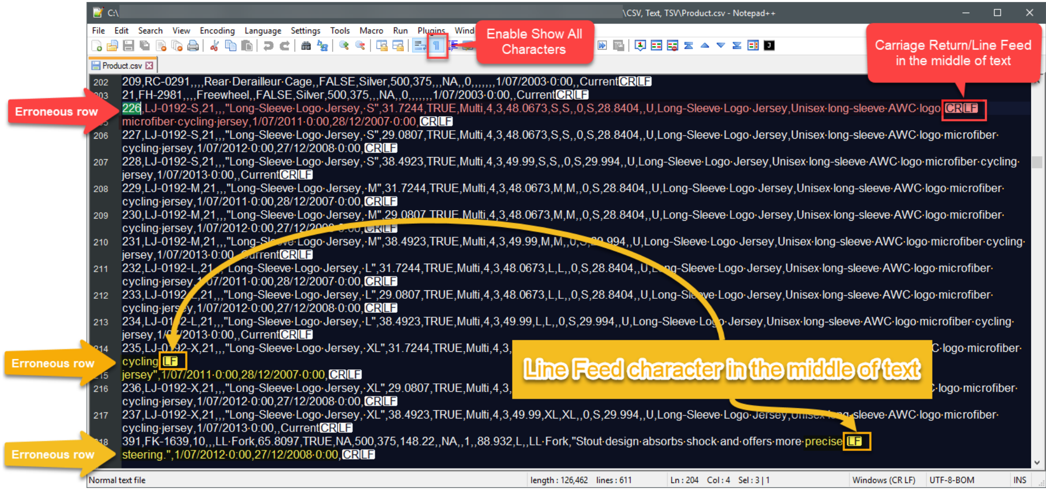


Figure 4.14 – The Product.csv file opened in Notepad++ reveals some issues

While we already found the culprit, the same issue might have happened in other parts of the data where it is not that obvious. The easiest way to deal with the issue is to fix it in the source, but in our case we do not have access to the source anymore. Therefore, we have to fix the issue in the Power Query layer. We may have quickly thought that we could replace the `(CR)(LF)` characters with a blank. But the reality is that the `(CR)(LF)` characters represent a new row of data. Therefore, we cannot simply replace these with a blank and expect to get the issue sorted.

Luckily, in our case, the issue is not that complex to fix. Looking at the erroneous rows, we can see that there was always a space character before the `(CR)(LF)` characters. This is also the case with the `(LF)` character. To fix the issue, we first need to get the text and fix the issues before we transform it into the table. While we can fix the issue that way, what if it happens again in the future in some other `CSV` files? In *Chapter 3*, *Data Preparation in Power Query Editor*, we learned how to create a custom function. To solve the issue we face in our scenario, we need a custom function that does the following:

1. Accepts a file path as a text value.
2. Opens the file as text.
3. Replaces all occurrences of `(CR)(LF)` (read `Space`, `Carriage Return`, `Line Feed`) with an empty string `("")`.
4. On the text results of the previous step, it replaces all occurrences of `(LF)` (read `Space`, `Line Feed`) with an empty string `("")`.
5. Transforms `Text` to `Binary`.
6. Opens the result of the previous step as CSV.
7. Promotes the first row as header.
8. Changes the column types.

With the following expression, we can create a custom function ticking the required boxes:

```
// fnTextCleaner
(FilePath as text) as table =>
let
    Source = File.Contents(FilePath),
    GetText = Text.FromBinary(Source),
    ReplaceCarriageReturn = Text.Replace(GetText, " #(cr,lf)",
""),
    ReplaceLineBreaks = Text.Replace(ReplaceCarriageReturn, "
```

```
#(lf)", ""),
    TextToBinary = Text.ToBinary(ReplaceLineBreaks),
    #"Imported CSV" = Csv.Document(TextToBinary,[Delimiter=",",
Columns=25, Encoding=1252, QuoteStyle=QuoteStyle.None]),
    #"Promoted Headers" = Table.PromoteHeaders(#"Imported CSV",
[PromoteAllScalars=true]),
    #"Changed Type" = Table.TransformColumnTypes(#"Promoted
Headers",{{"ProductKey", Int64.Type}, {"ProductAlternateKey",
type text}, {"ProductSubcategoryKey", type number},
{"WeightUnitMeasureCode", type text}, {"SizeUnitMeasureCode",
type text}, {"EnglishProductName", type text}, {"StandardCost",
type text}, {"FinishedGoodsFlag", type logical}, {"Color",
type text}, {"SafetyStockLevel", Int64.Type}, {"ReorderPoint",
Int64.Type}, {"ListPrice", type text}, {"Size", type
text}, {"SizeRange", type text}, {"Weight", type text},
{"DaysToManufacture", Int64.Type}, {"ProductLine", type text},
{"DealerPrice", type text}, {"Class", type text}, {"Style",
type text}, {"ModelName", type text}, {"EnglishDescription",
type text}, {"StartDate", type datetime}, {"EndDate", type
datetime}, {"Status", type text}})
in
    #"Changed Type"
```

To create a custom function, we need to create a new blank query, open the **Advanced Editor**, then copy and paste the preceding scripts into the **Advanced Query**. We then rename the query to `fnTextCleaner`. We can then simply invoke the function with the `Product.csv` file to fix the issue, as follows:

1. Select the `fnTextCleaner` function from the **Queries** pane.
2. Type in the file path you stored the `Product.csv` file in.
3. Click **Invoke**.
4. This creates a new query named `Invoked Function`.
5. Rename the query as `Product`.

The preceding steps are highlighted in the following screenshot:

Figure 4.15 – Invoking the fnTextCleaner function to fix the text issues in the Product.csv file

As shown in the preceding screenshot, we now get **606** rows, which is correct.

Excel

Excel is one of the most popular data sources used in Power BI. There is a dedicated connector for Excel, so it is straightforward to connect to an Excel data source. However, it is not always that simple to work with Excel data sources. In reality, there are many cases where we get an Excel file full of formulas referencing tables, columns, or cells from other worksheets, or even from other workbooks. In those cases, we may face some issues and get errors generated by the formulas.

There are other scenarios where we get Excel files full of pivot tables having several dimensions. Power BI follows relational data modeling consisting of tables, and tables contain columns, so dealing with multidimensional pivot tables is not always straightforward.

Another common issue when dealing with Excel sources is merged cells leading to missing data in Power BI. You may think of many more different scenarios that make dealing with Excel data quite complex. This section looks at one of the most common scenarios: an Excel file containing a pivot table. The `Yearly Product Category Sales.xlsx` file is a sample Excel file we use in this scenario. The following screenshot shows the contents of this sample file:

		Year					
Category	**Subcategory**	**2010**	**2011**	**2012**	**2013**	**2014**	**Grand Total**
Accessories	Bike Racks				$ 36,960.00	$ 2,400.00	$ 39,360.00
	Bike Stands			$ 159.00	$ 37,683.00	$ 1,749.00	$ 39,591.00
	Bottles and Cages			$ 280.62	$ 55,008.82	$ 1,508.75	$ 56,798.19
	Cleaners				$ 6,908.55	$ 310.05	$ 7,218.60
	Fenders			$ 109.90	$ 44,443.56	$ 2,066.12	$ 46,619.58
	Helmets			$ 909.74	$ 216,028.26	$ 8,397.60	$ 225,335.60
	Hydration Packs			$ 109.98	$ 38,932.92	$ 1,264.77	$ 40,307.67
	Tires and Tubes			$ 577.84	$ 232,276.42	$12,675.06	$ 245,529.32
Bikes	Mountain Bikes	$16,974.95	$1,332,364.80	$2,263,420.53	$ 6,339,999.28		$ 9,952,759.56
	Road Bikes	$26,446.09	$5,743,161.12	$3,554,883.93	$ 5,196,092.90		$14,520,584.04
	Touring Bikes			$ 21,390.87	$ 3,823,410.18		$ 3,844,801.05
Clothing	Caps			$ 71.92	$ 18,870.01	$ 746.17	$ 19,688.10
	Gloves			$ 73.47	$ 33,379.87	$ 1,567.36	$ 35,020.70
	Jerseys			$ 415.92	$ 165,574.11	$ 6,960.65	$ 172,950.68
	Shorts				$ 67,400.37	$ 3,919.44	$ 71,319.81
	Socks			$ 17.98	$ 4,863.59	$ 224.75	$ 5,106.32
	Vests			$ 63.50	$ 33,718.50	$ 1,905.00	$ 35,687.00
Grand Total		**$43,421.04**	**$7,075,525.93**	**$5,842,485.20**	**$16,351,550.34**	**$45,694.72**	**$29,358,677.22**

Yearly Product Category Sales

Figure 4.16 – Yearly Product Category Sales data stored in Excel

The aim is to load the preceding pivot table into Power BI. To do this, proceed as follows:

1. In the **Power Query Editor**, click the **New Source** drop-down button.
2. Click **Excel**.
3. Navigate to the folder containing the Excel file and select the Excel file.
4. Click **Open**.
5. Select the `Yearly Product Category Sales.xls` sheet.
6. Click **OK**.

The preceding steps are highlighted in the following screenshot:

Figure 4.17 – Getting data from Excel

As looking at *Figure 4.17* shows, Power BI already created some steps. Let's look at it more precisely to see how we can turn the pivot table into a regular table, as follows:

- The **Changed Type** step is not necessary, so we can remove it.
- We have to remove the `Grand Total` column appearing in `Column8` and the `Grand Total` row appearing in *row 19*.
- We must fill in the missing data in `Column1`.

- The column headers of the first two columns appear in the data. We need to promote them as column headers.
- The year values must be shown in a column.

The preceding points are explained in the following screenshot:

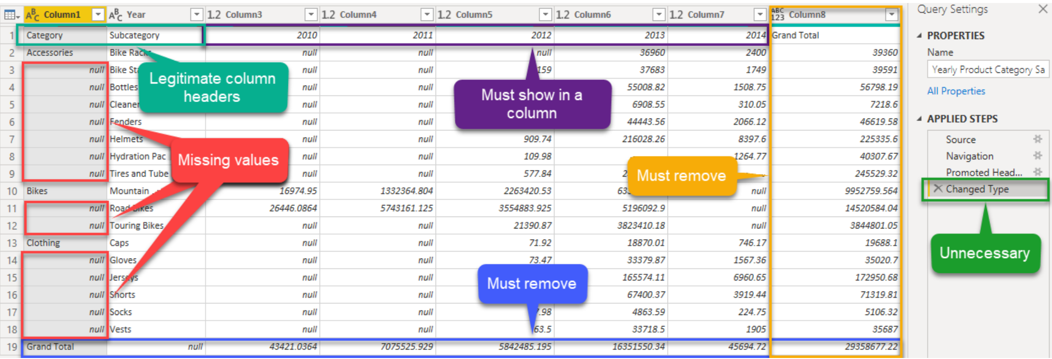


Figure 4.18 – Transforming a pivot table

Follow these next steps to fix the preceding issues one by one:

7. Remove the **Change Type** step.
8. Click the `Column8` column.
9. Click the **Remove Columns** button from the **Home** tab, as shown in the following screenshot:

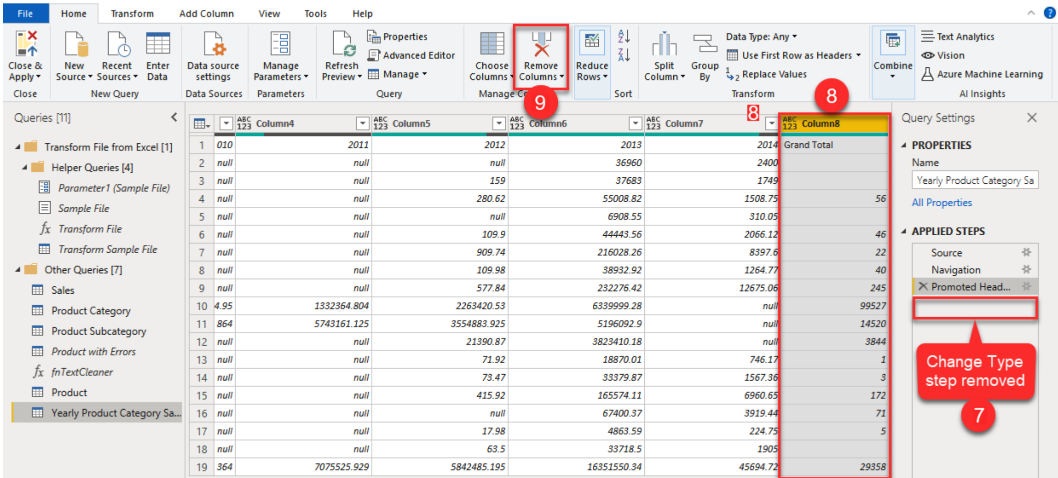


Figure 4.19 – The excessive step and column removed

In the next few steps, we remove the **Grand Total** row, as follows:

10. Click the **Remove Rows** button from the **Home** tab and click **Remove Bottom Rows**.
11. Type `1` in the **Number of rows** textbox.
12. Click **OK**.

The preceding steps are highlighted in the following screenshot:

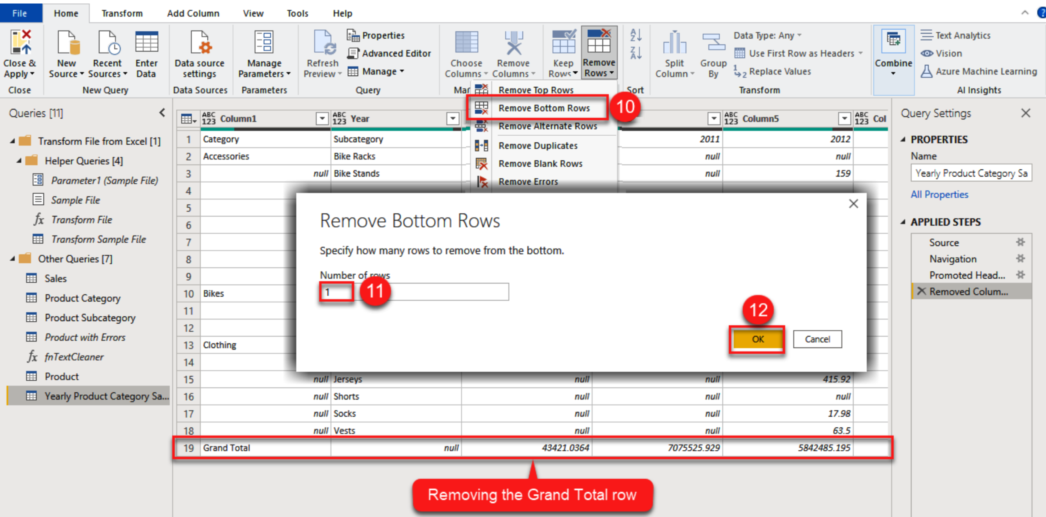


Figure 4.20 – Removing the bottom row

Follow these steps to fill the missing items in `Column1`:

13. Click `Column1`.
14. Click the **Fill** button, then click **Down**.

The preceding steps are highlighted in the following screenshot:

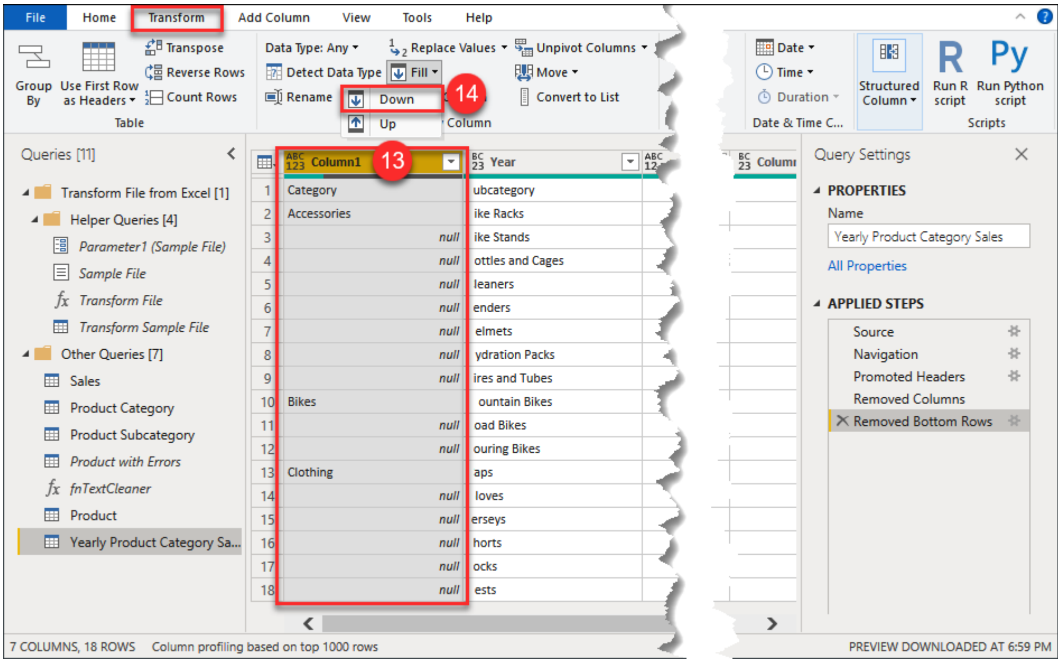


Figure 4.21 – Filling missing values

The next step is to promote column headers. To do this, we use `Category` and `Subcategory` and also year numbers as column headers. Proceed as follows:

15. Click **Use First Row as Headers** from the **Home** tab.

The following screenshot shows the preceding step. Note the results of the previous step to fill the missing values:

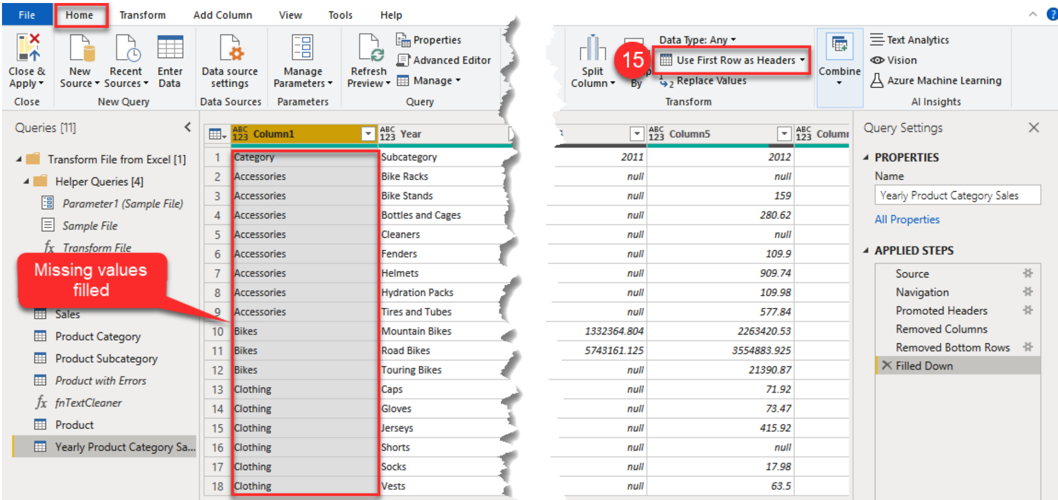


Figure 4.22 – Using the first row as a header

We now need to move the year numbers from column headers to columns. But the earlier step also added a **Changed Type** step that is still not necessary, so we need to remove that as well. To do that, we just need to unpivot the columns with year-number headers. The following steps take care of that:

16. Remove the **Changed Type** step by selecting it and clicking the *X* icon.
17. Select all columns with a year number by pressing down and holding the *Ctrl* key on your keyboard and clicking the columns.

> **Note**
>
> You can select all columns between the first and last click in one go by pressing and holding the *Shift* key on your keyboard, selecting the first column, and then selecting the last column.

18. Right-click on any column header and click **Unpivot Columns** from the context menu.

    The preceding steps are highlighted in the following screenshot:

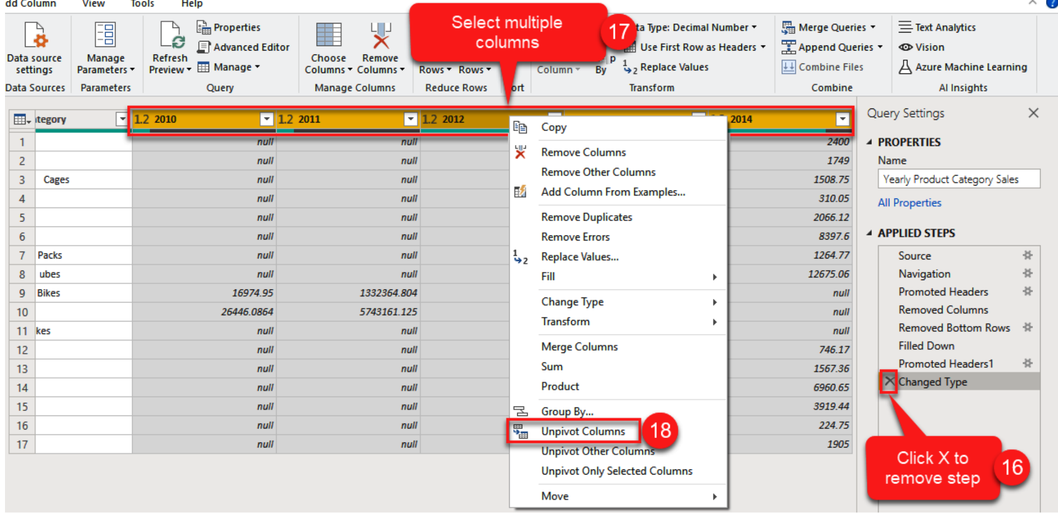


Figure 4.23 – Removing unnecessary step and unpivoting columns

The very last steps are listed here:

19. Rename the `Attribute` column to `Year` and rename `Value` to `Sales`.
20. Change all column types by selecting all columns.

21. Click the **Detect Data Type** button from the **Transform** tab, as shown in the following screenshot:

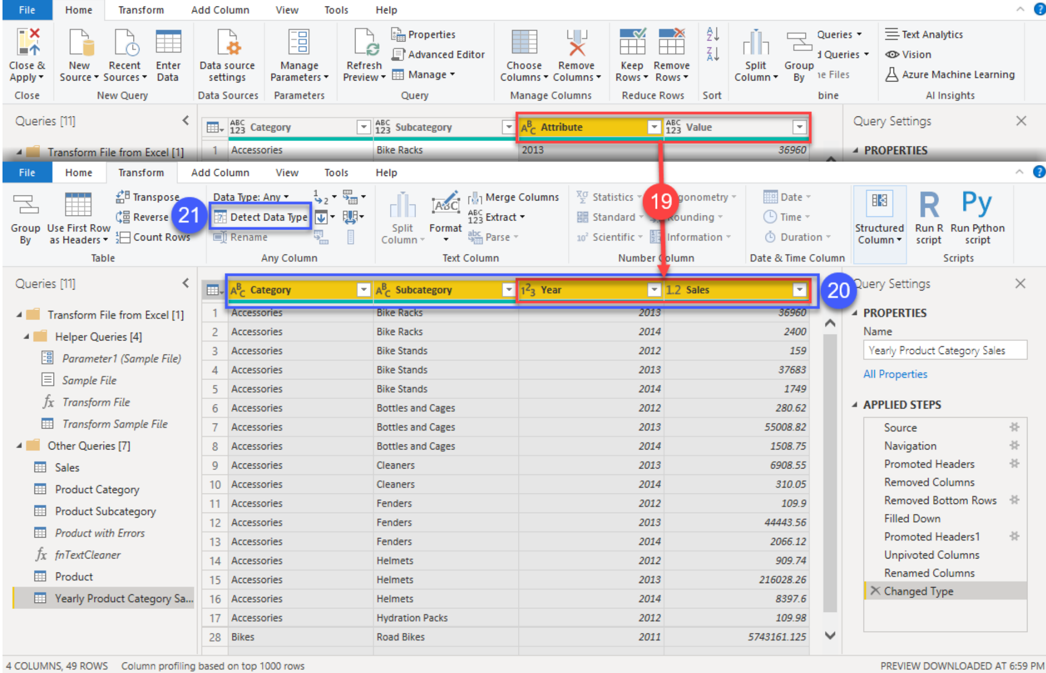


Figure 4.24 – Renaming columns and changing the columns' types

In this section, we dealt with a real-world challenge when working with Excel files. In the next few sections, we look at some other common data sources.

Power BI datasets

Power BI datasets are getting more and more popular when it comes to teamwork and collaboration. When we create a data model in Power BI Desktop and publish it to a Power BI service workspace, the data model turns to a dataset in the Power BI service. We then make the datasets available to others across the organization. This is an efficient method of collaboration in large Power BI projects. The data modelers create the models and publish them to the service, and the report writers make several reports on top of the shared datasets available to them.

> **Note**
>
> The Power BI datasets are not available in the Power BI free licensing plan.

We can **connect live** to a Power BI dataset from Power BI Desktop, so Power BI Desktop turns to a data visualization tool. This means that Power Query and the data model are not accessible anymore. The reason is that the semantic model is now kept in the dataset. Therefore, if there is anything to be changed in the data model, those changes must then be applied to the dataset by the data modelers.

> **Note**
> We will discuss different connection modes in this chapter in the *Working with connection modes* section.

When we *connect live* to a dataset, we can create report level measures within Power BI Desktop. The report level measures do not make any changes in the data model; they are only available within their containing report. Therefore, we cannot access those measures from any other reports of datasets. The following steps show the process to connect to a Power BI dataset:

1. Click the **Get data** button from the **Home** tab.
2. Click **Power Platform**.
3. Click **Power BI datasets**.
4. Click **Connect**.

The preceding steps are highlighted in the following screenshot:

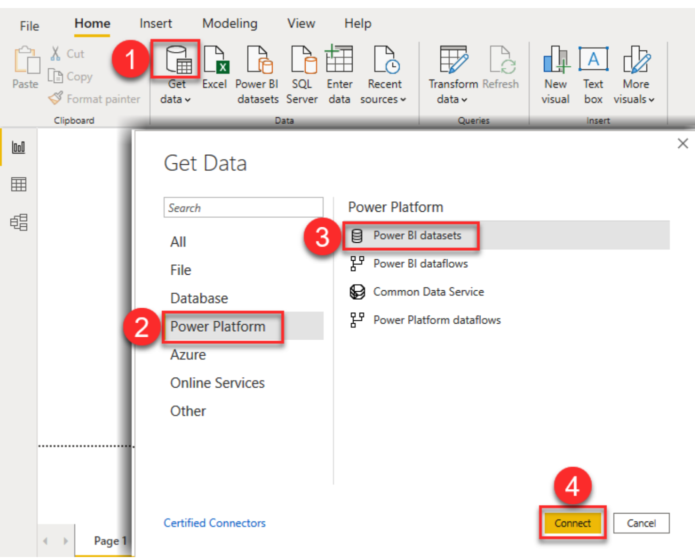


Figure 4.25 – Connecting to a Power BI dataset

Alternatively, we can click the **Power BI datasets** button from the **Data** section from the **Home** tab, as shown in the following screenshot:

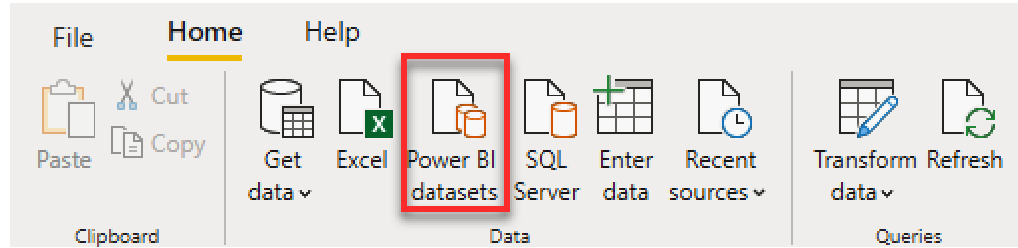


Figure 4.26 – Power BI datasets button from the Home tab

5. Select the desired dataset.
6. Click **Create**.

The preceding steps are highlighted in the following screenshot:

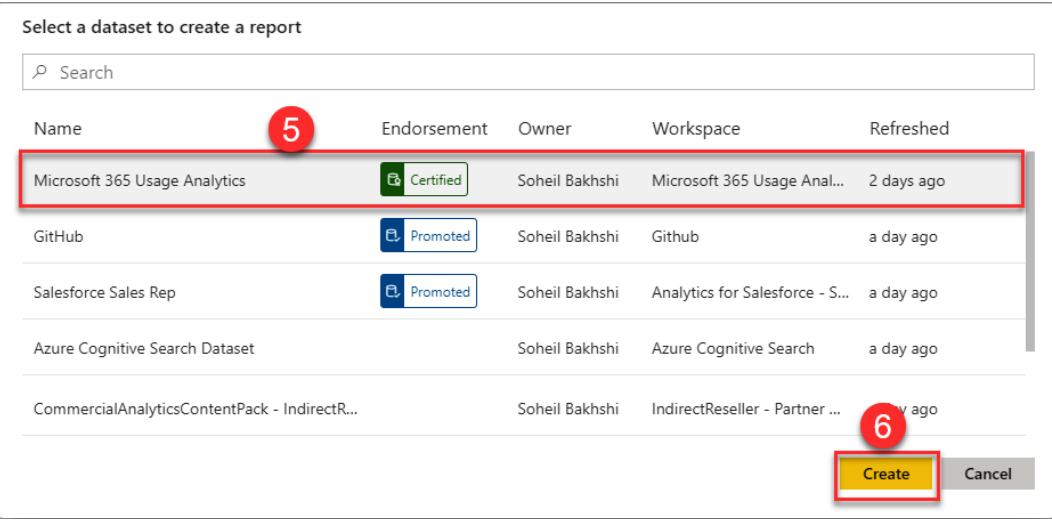


Figure 4.27 – Selecting an available dataset to connect to

After we *connect live* to a dataset, only reporting features are available in Power BI Desktop, as outlined here:

- All tools in the **Data** section from the **Home** tab are disabled.
- The **Transform data** button that opens the **Power Query Editor** is disabled.
- All tools from the **Modeling** tab are disabled except the **New measure** and the **Quick measure** tools, which we can use to create report level measures with the current report.
- The **Data** tab disappears from the left pane, while the **Report** and **Modeling** tabs are still enabled. We will look at the **Modeling** tab later in this section.

The following screenshot shows the tooling changes in Power BI Desktop after connecting to a Power BI dataset:

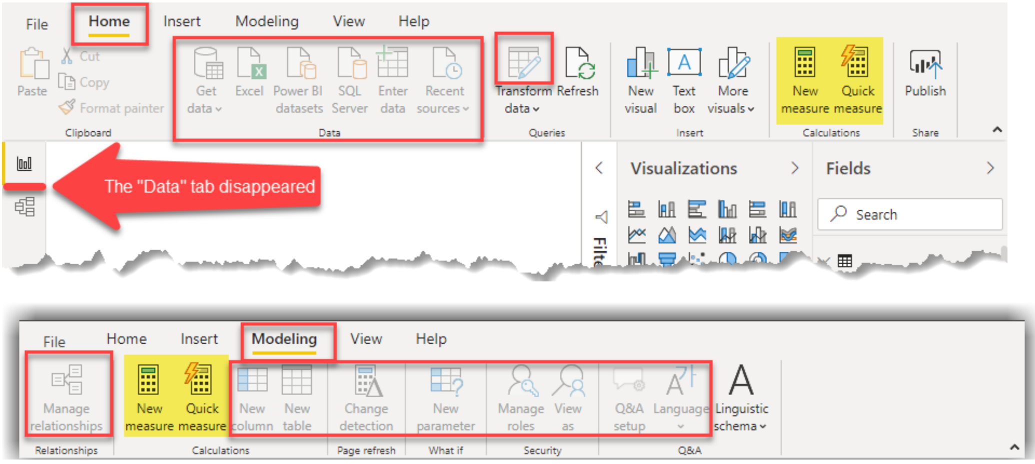


Figure 4.28 – Power BI Desktop tooling changes after connecting to a Power BI dataset

As we see in the preceding screenshot, the **Model** view is still accessible from the left pane. If we click the **Model** view, we can see all tables and their relationships, which is very handy to better understand the underlying data model. The following screenshot shows the **Model** view:

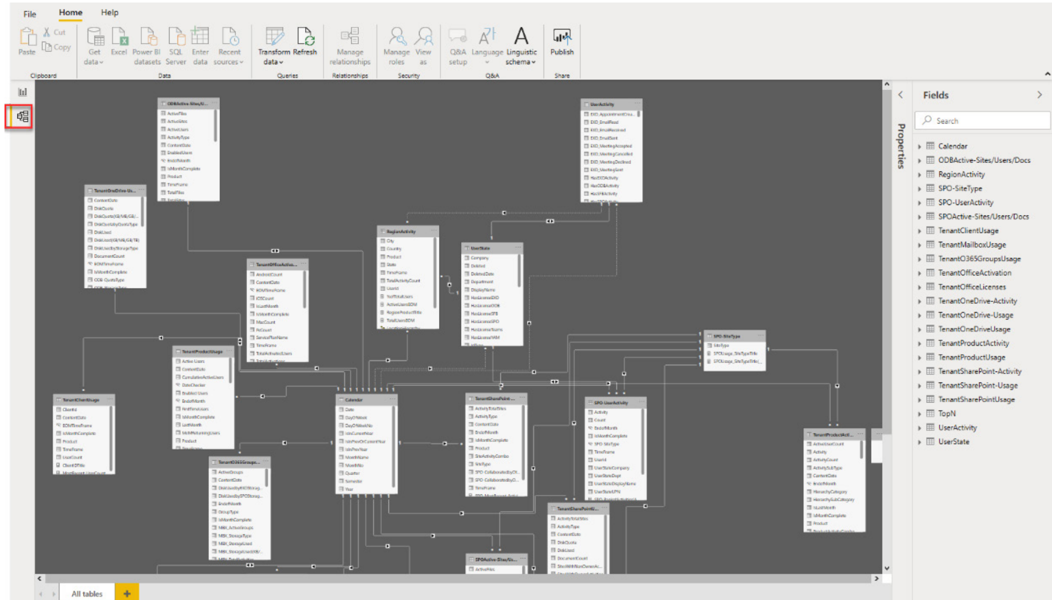


Figure 4.29 – Underlying data model of a connected Power BI dataset

We can also create new layouts on top of the model, as shown in the following screenshot:

Figure 4.30 – Creating new layouts in the data model

There is one more point to note: we can only see and access the datasets from workspaces we are invited to. We also need to have **Admin**, **Member**, or **Contributor** permissions.

Power BI dataflows

A Power BI dataflow is a cloud experience for Power Query in a Power BI service, which opens endless self-service data preparation opportunities to organizations. It is pretty helpful to create a data preparation process in a dataflow that can be made available across an organization, increasing reusability and improving development efficiency. In Power BI, there is a dedicated connection for dataflows.

> **Note**
>
> Currently, **DirectQuery** to dataflows connection is in public preview. It is only available on the dataflows created in a workspace backed by a premium capacity.

If we already have an available dataflow, connecting to it is very easy. Follow these steps to connect to a Power BI dataflow:

1. Click the **Get data** dropdown.
2. Click **Power BI dataflows**.
3. A list of all dataflows available to you shows up in the **Navigation** window. Expand the desired workspace.
4. Expand the dataflow model.
5. Select a table.
6. Click **Load**.

The preceding steps are highlighted in the following screenshot:

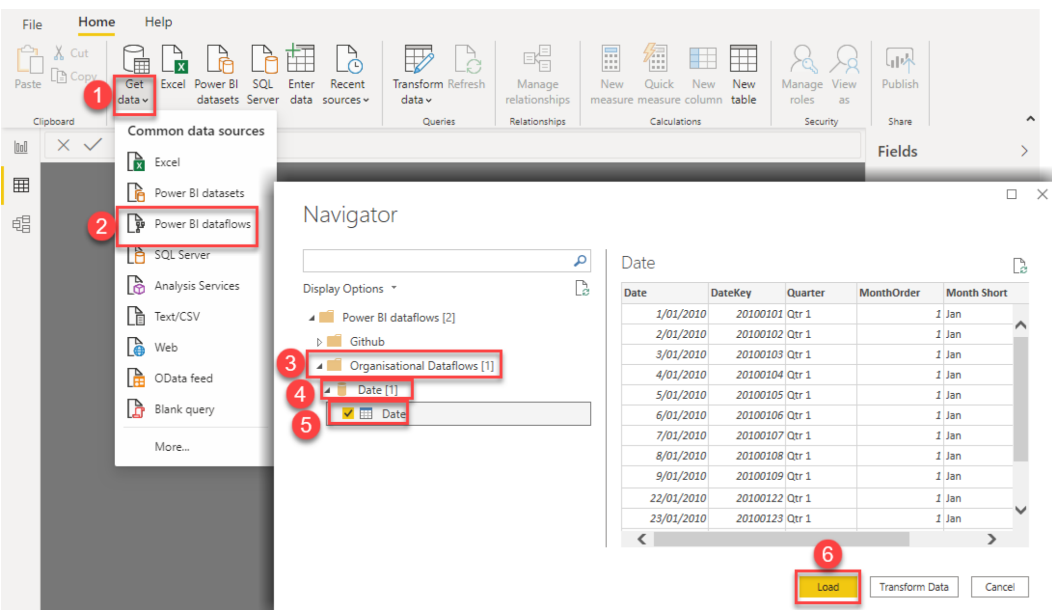


Figure 4.31 – Getting data from Power BI dataflows

By clicking the **Load** button, we import the data from the dataflow into the Power BI data model.

SQL Server

SQL Server is one of the most used data sources for Power BI. When connecting to a SQL Server database, we choose to import the data into the data model or connect to the database in **DirectQuery** mode. We will discuss the connection modes later in this chapter. The following steps will help you to get data from a SQL Server data source, as also shown in *Figure 4.32*:

1. Click the **SQL Server** button from the **Home** tab.
2. Enter the **Server** name.
3. Depending on your case, you may want to enter the **Database** name (optional).
4. You can either select **Import** or **DirectQuery**.
5. Again, depending on your case, you may want to enter some **Transact-SQL** (**T-SQL**) scripts. To do so, click to expand the **Advanced options**.
6. Leave the **Include relationship columns** option ticked (untick this option to stop Power BI from detecting related tables when selecting a table from the **Navigator** page and clicking the **Select Related Tables** option).
7. Leave the **Navigate using full hierarchy** option unticked (tick this option if you would like to see the navigation based on database schemas).
8. If you have **High Availability** (**HA**) settings on the SQL Server instance, then tick the **Enable SQL Server Failover support** item; otherwise, leave it unticked.
9. Click **OK**.

The preceding steps are highlighted in the following screenshot:

Figure 4.32 – Getting data from SQL Server

10. Tick a desired table from the list, as shown in *Figure 4.33*.
11. Click the **Select Related Tables** button.
12. Depending on what you need to do next, you can either click the **Load** button to load the data into the data model or click the **Transform Data** button, which navigates you to the **Power Query Editor**.

The preceding steps are highlighted in the following screenshot:

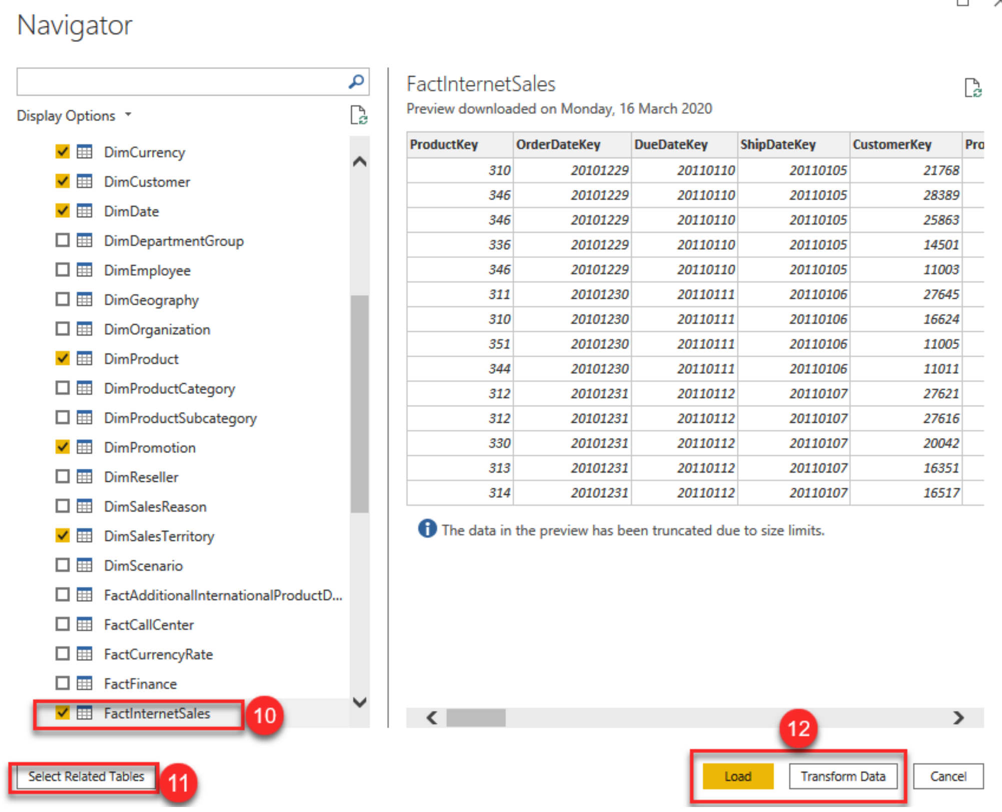


Figure 4.33 – Selecting a table and related tables from the Navigator window

Note

As shown in *Step 5* in *Figure 4.32*, typing T-SQL queries disables query folding in Power Query, which potentially causes some performance degradation during the data refresh. We will discuss this more in *Chapter 7*, *Data Preparation Common Best Practices*, in the *Query folding best practices* section.

SQL Server Analysis Services and Azure Analysis Services

SQL Server Analysis Services (**SSAS**) has two different technologies, multidimensional and tabular models. **Azure Analysis Services** (**AAS**) is a **Platform as a Service** (**PaaS**) version of the tabular model in Azure. There is a dedicated connection for connecting to both types of on-premises versions of SSAS and another dedicated connector supporting AAS. When we connect from Power BI Desktop to SSAS or AAS, there are two connection types available to us: **Connect Live** and **Import**. If we select the **Connect Live** option, Power BI Desktop turns to a reporting tool only. In that case, the semantic model is hosted in either an on-premises instance of SSAS or an AAS instance. When we *connect live* to either SSAS Tabular or AAS, we can create **report level measures**. The report level measures are only available across the report, therefore they are not accessible within the underlying semantic model.

> **Note**
>
> It is advised not to use **Import mode** when connecting to either SSAS or AAS. When we create a model in Power BI, we are indeed creating a semantic model. Importing SSAS or AAS data into a Power BI model means creating a new semantic model on top of an existing semantic model. This is not ideal. Therefore, it is best to avoid **Import mode** unless we have a strong justification not to do so.
>
> In December 2020, Microsoft announced a new version of composite models where we can turn a **Connect Live** connection to an AAS connection or a Power BI dataset to **DirectQuery**. Therefore, we can now connect to multiple AAS or Power BI datasets to build a single source of truth within an enterprise-grade semantic layer in Power BI. We cover composite models in *Chapter 12, Extra Options and Features Available for Data Modeling*.

SSAS Multidimensional/Tabular

To connect to an instance of SSAS, follow these steps:

1. Click the **Get data** dropdown.
2. Click **Analysis Services**.
3. Enter the **Server** name.
4. Enter the **Database** name (optional).
5. Select the connection mode; if you happened to select **Import**, you could write **Multidimensional Expressions** (**MDX**) or DAX expressions to get a result set.

6. Click **OK**.

The preceding steps are highlighted in the following screenshot:

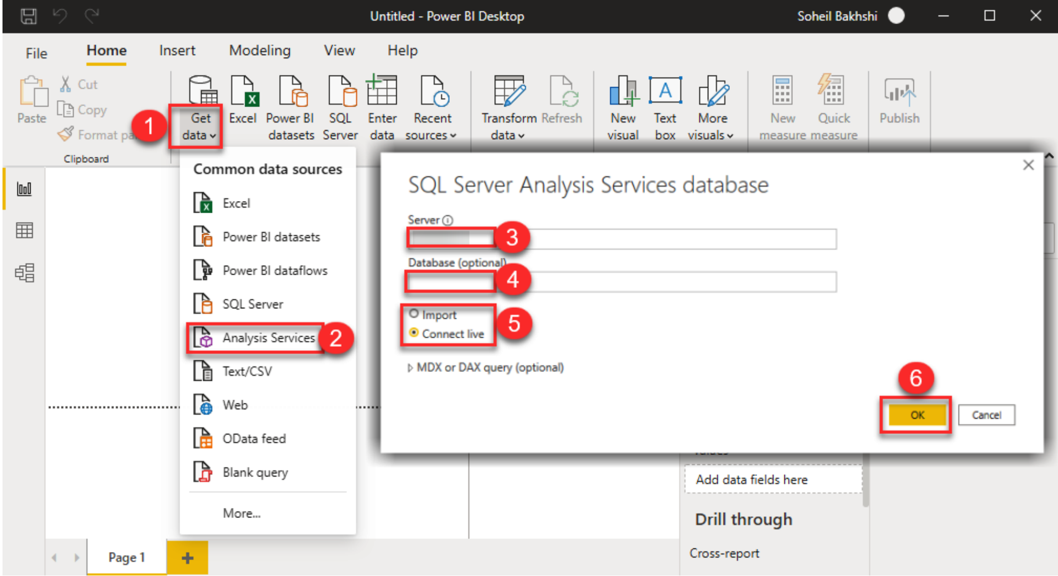


Figure 4.34 – Connecting to SSAS

7. Select the model (you may see a different model name than the one shown in *Figure 4.35*).
8. Click **OK**.

The preceding steps are highlighted in the following screenshot:

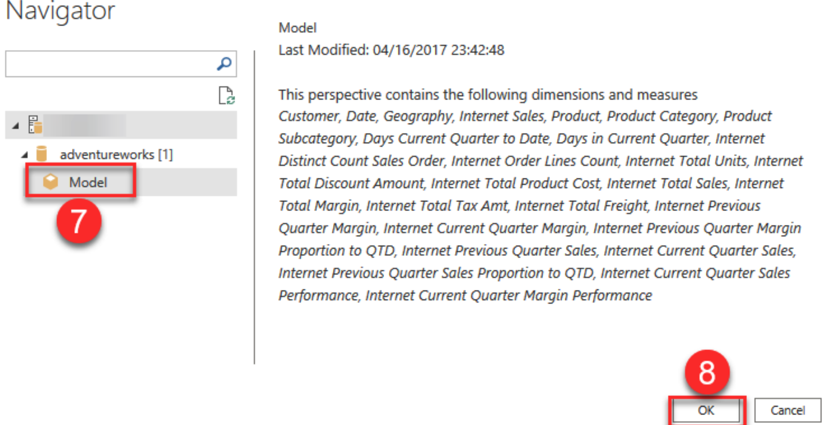


Figure 4.35 – Connecting to the model

AAS

Connecting to an instance of AAS is pretty much the same as connecting to an instance of SSAS. The only difference is that this time, we use its dedicated connector. We can find the **Azure Analysis Services database** connector under the `Azure` folder from the **Get Data** window, as shown in the following screenshot, so we do not go through the steps again:

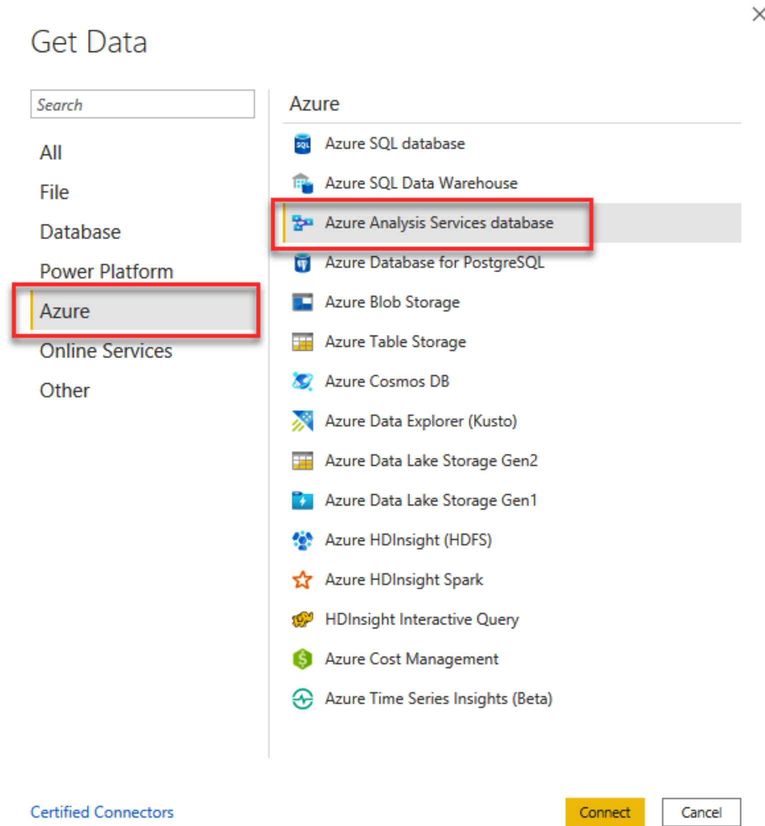


Figure 4.36 – Connecting to an AAS tabular model

We can download a PBIX file directly from the Azure portal containing the connection to the desired model in AAS. The following steps explain how to do this:

1. After logging in to the Azure portal, navigate to your instance of AAS, then click **Overview**.
2. Find the model you would like to connect to and click the ellipsis button on the right.
3. Click **Open in Power BI Desktop**.

The following screenshot shows the preceding steps:

Figure 4.37 – Opening AAS model in Power BI Desktop from the Azure portal

By going through the preceding steps, we download a PBIX file. In Power BI Desktop, we can open the file that is *connected live* to our AAS model.

OData Feed

Another most common data sources for Power BI in organizations is **Open Data Protocol** (**OData**). Many web services support OData, which is one of the reasons this type of data source is quite common. If the source system that is accessible via an OData connection is a **Customer Relationship Management** (**CRM**) or an ERP system, in that case the underlying data model contains many tables, and those tables may have many columns. In some cases, we have had experience of dealing with wide tables with more than 200 columns, therefore having a good understanding of the underlying data model is essential. In this section, we describe how to connect to the underlying data model of Microsoft **Project Web App** (**PWA**), as follows:

1. In Power BI Desktop, click the **Get data** drop-down button.
2. Click **OData feed**.

3. Enter your PWA **Uniform Resource Locator** (**URL**). It must look like this: `https://Your_Tenant.sharepoint.com/sites/pwa/_api/Projectdata`.
4. Click **OK**.
5. Click **Organizational account**.
6. Click the **Sign in** button, then pass your credentials.
7. Click **Connect**.

   The preceding steps are highlighted in the following screenshot:

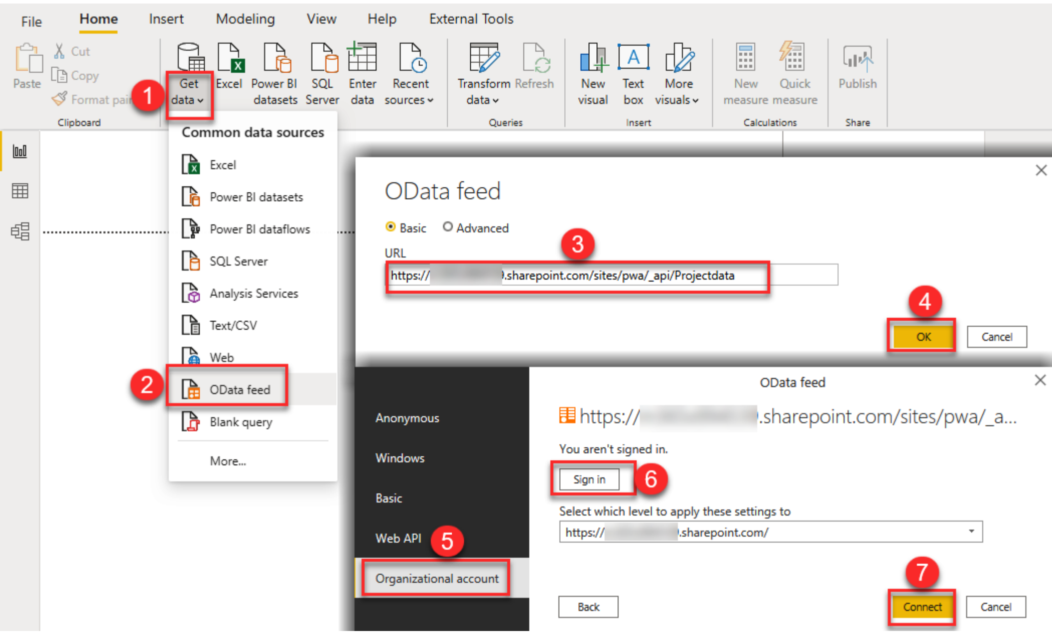


Figure 4.38 – Connecting to Microsoft Project Online (PWA) with OData feed

8. Select the desired tables; in our sample, we selected the **Project** table.
9. Click either **Load** or **Transform Data**; we selected the **Transform Data** option.

The preceding steps are highlighted in the following screenshot:

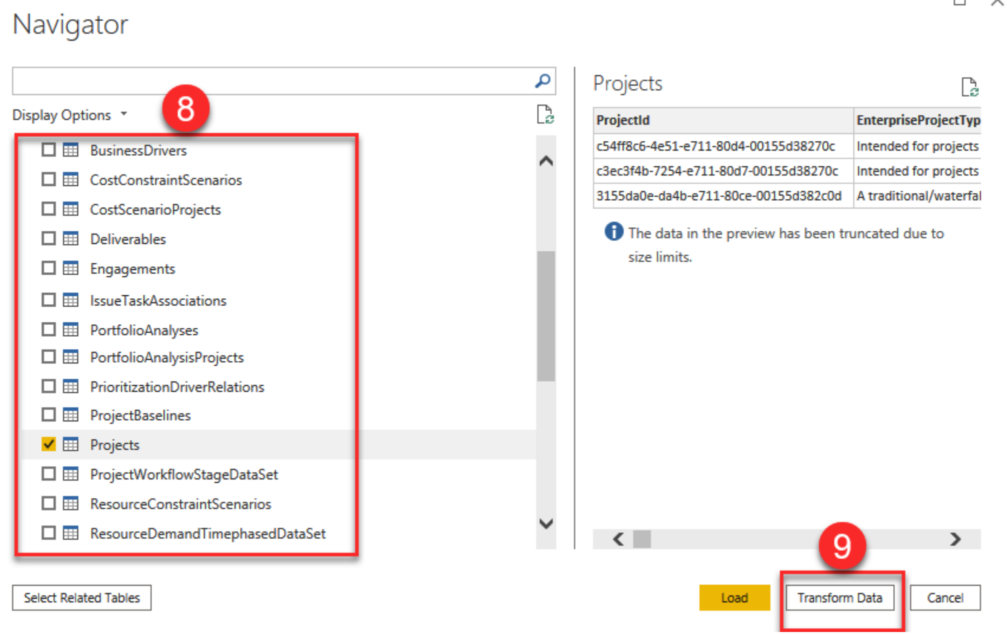


Figure 4.39 – Selecting tables from the Navigator page

As shown in the following screenshot, the **Project** table has 131 columns:

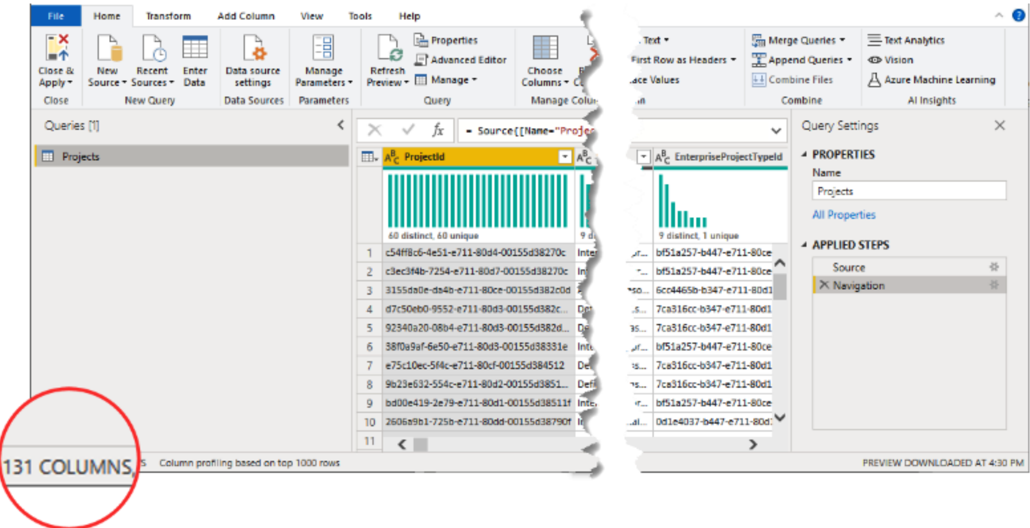


Figure 4.40 – The Project table from the PWA data source in Power Query

We may not need to load all those columns into the data model. Therefore, having a good understanding of the PWA database structure is essential. If we do not know much about the underlying data source, it is wise to involve the **subject-matter experts** (**SMEs**) from the business. They can help to identify the columns that are more relevant to the business.

It is always good to have an idea of the number of tables, their columns, and the number of rows they contain. We provide a `fnODataFeedAnalysis` custom function in *Chapter 7*, *Data Preparation Common Best Practices*, in the *General considerations* section, and this can help to get that kind of information.

Understanding data source certification

While data source certification leans toward Power BI governance, it is crucial to understand what it is and how it affects data modelers. The data source certification is more about the quality of data and assessing the level of trust we can build upon the data available in different data sources. This section will not explain the steps and processes of data source certification as it is out of this book's scope. The results of data source certification group our data sources into three (or more) categories. Different organizations use different terminology to refer to the quality of their data. This section uses **Bronze**, **Silver**, and **Gold** categories, indicating the quality of data contained in data sources. You may use different terms in your data source certification.

Bronze

The Bronze data sources contain uncurated data. The data has never been thoroughly quality controlled. While the data source may have valuable data it may have some duplications, or even incorrect data. The other factor that we may consider is the location in which the data is stored and not the quality of the data itself; for example, when we store Excel files in our personal OneDrive or Google Drive storage. The `Bronze` data sources are typically copied or exported from the web or some other source systems. Perhaps some SMEs have done some analysis on the data. However, the data is not necessarily in good shape to be consumed in analytical tools such as Power BI. The data may be stored in untrusted storage not managed by the organization (such as personal OneDrive storage). The most common data sources that fall into this category are uncurated Excel, CSV, and Txt files. Many organizations strictly ban using `Bronze` data sources as the contained data cannot be trusted and is prone to errors and incorrect figures. The maintenance costs associated with analyzing the `Bronze` data source are often relatively high. If you have to deal with `Bronze` data for any reason, consider more complex data preparation and higher maintenance costs while estimating the development.

Silver

The `Silver` category is data that is semi-curated, and the organization manages the storage. While there might still be some data quality issues here and there, the data can be used in analytical tools to gain insights. Nevertheless, the data preparation costs are reasonable as the data is not in its best shape. The most common data sources categorized as `Silver` are transactional data sources stored by different technologies in different locations. These can typically be transactional databases hosted in SQL Server or Oracle, or Excel files stored in SharePoint or OneDrive for Business. Some organizations even consider some of their data warehouses as being a `Silver` data source, so it depends on how we define the boundaries.

Gold/Platinum

`Gold` or `Platinum` data sources are fully curated from both a data and business perspective. Data quality issues are minimal. The data is in its best possible shape to be analyzed and visualized in analytical and reporting tools such as Power BI. Typical `Gold` data sources can be semantic models hosted in SSAS, either multidimensional or tabular models, **SAP Business Warehouse** (**SAP BW**), data warehouses, and so on. When we deal with a `Gold/Platinum` data source, we are more confident that the data is correct. We expect almost zero data preparation and data modeling efforts from a development perspective.

Working with connection modes

When we get data from a data source, the query connection mode falls into one of the following three different categories:

- **Data Import**
- **DirectQuery**
- **Connect Live**

Every query connecting to one or more data sources in the **Power Query Editor** has any of the preceding connection modes, except Connect Live. When the connection mode is **Connect Live**, a Power BI model cannot currently connect to more than one instance of SSAS, AAS, or a Power BI dataset.

> **Note**
>
> Microsoft released the preview of the new version of the composite models in December 2020. With the new version of composite models, we can **DirectQuery** to more than one instance of AAS or Power BI datasets. The new composite models currently do not support SSAS Tabular. We will discuss composite models in more detail in *Chapter 12*, *Extra Options and Features Available for Data Modeling*.

In this section, we look at the connection modes, their applications, and their limitations.

Data Import

This is the most common connection mode in Power BI Desktop. It is the only option in many connections, such as file-based connections. As its name suggests, the **Data Import** mode imports the data from the source system into the Power BI data model in a different shape. The data is already prepared and transformed in the Power Query layer; then, it is imported into the data model. We can refresh the data manually within Power BI Desktop or automatically after publishing it to the Power BI service.

Applications

The main application for the **Data Import** mode is consolidating data from different sources in a single source of truth. In addition, it gives us data modeling capabilities to create a semantic model in Power BI. We can frequently refresh the data throughout the day. All that goodness comes with some limitations, as outlined next.

Limitations

The Power BI datasets in **Data Import** mode have limited storage size and automatic data refreshes based on the licensing tier.

Storage per dataset is limited to the following:

- 1 **gigabyte** (**GB**) per dataset under the **free** and **Pro** licensing plans
- 100 GB for datasets published to workspaces backed with a **Premium Per User** (**PPU**) capacity
- 400 GB for datasets published to workspaces backed with a **Premium** capacity

Depending on our licensing tier, we can currently schedule the data refreshes from the Power BI service with the following restrictions:

- Up to once a day for the free licensing plan
- Up to 8 times a day for Power BI Pro
- Up to 48 times a day for Power BI Premium and PPU

With the preceding restrictions in mind, if real-time or near-real-time data analytics is required, this mode may not be ideal.

DirectQuery

While the most common connection mode is the **Data Import** mode, for some data sources, an alternative approach is connecting directly to the data source using **DirectQuery**. When a connection is in **DirectQuery** mode, it does not import the data into the model. Instead, it fires multiple concurrent queries back to the data source, which can be a relational database data source, to get the results.

Applications

This connection mode is ideal for supporting real-time data processing scenarios with minimal data latency when the data is stored in a relational database. The other application is when the 1 GB file size limit is not sufficient due to a large amount of data in the data source. While in **DirectQuery** mode we do not import any data into a data model, we still can create a data model with some limitations.

Limitations

Generally speaking, queries in DirectQuery mode are not as performant as similar queries in **Import** mode. DirectQuery fires concurrent queries back to the source system. So, if the source database system is not strong enough to handle many concurrent queries (or it is not well configured), performance issues can lead to timeouts. In that case, the underlying data model and, consequently, the reports will fail.

If the queries in **DirectQuery** mode are complex, then we can expect poor performance. The other drawback is when many concurrent users are running the same report, leading to even more concurrent query executions against the source system.

Moreover, some DAX expressions and Power Query limitations apply to the **DirectQuery**, making this connection mode even more complex to use.

The DirectQuery is also limited to retrieve 1 million rows for the cloud sources and 4 **megabytes** (**MB**) of data per row for an on-premises data source, or a maximum of 16 MB data size for the entire visual under the Power BI Pro license. The maximum row limit for Power BI Premium can be set under the `admin-set` limit.

Therefore, it is essential to undertake thorough investigations before deciding to use **DirectQuery**.

Connect Live

Connect Live mode is used when the report is connected to SSAS, including AAS, either Multidimensional or Tabular. **Connect Live** is indeed the recommended connection mode to be used for reporting in an enterprise **business intelligence** (**BI**) solution. In this connection mode, all business logics are captured in the semantic model and made available for all reporting tools within the SSAS instance we are connecting to. In this mode, the underlying data is kept on the SSAS side, so we do not import any data into Power BI.

Applications

This connection mode is the desired connection mode when the source system is an instance of SSAS Tabular or Multidimensional.

Limitations

When we connect live to an instance of SSAS, Power BI turns to a reporting tool only. Therefore, there are no Power Query or data modeling capabilities available under this connection model. Having said that, we can still see the underlying data model in Power BI Desktop. Moreover, we can create report level measures when connecting live to an SSAS Tabular model (this is not applicable when connecting live to SSAS Multidimensional). While we can see Power BI Desktop's underlying data model, we cannot make any changes to the data model. As all the data processing is done in the SSAS instance, the connecting SSAS instance must be strong enough to respond to the concurrent users.

Working with storage modes

In the previous section, we discussed connection modes for the queries from a Power Query perspective. In this section, we look at different storage modes that apply to tables after the data is loaded into a data model or connected to a data source. Every table in a Power BI data model has a storage mode property that shows if the data is cached or not. There are three types of storage modes, as outlined next:

- **Import**: This means the data is cached into the memory. Therefore, all queries over a table with the storage mode of **Import** get the results from the cashed data.
- **DirectQuery**: The data is not cached; therefore, all queries are fired back to the source system to get the results.
- **Dual**: The tables in this mode can get data either from the cache or directly from the source system. So, depending on the query, data can be retrieved either from the cached data or directly from the data source. For instance, in an aggregation setup, the query results may come from the cached data or directly from the source, depending on the level we drill down. We will discuss aggregations in *Chapter 9, Star Schema and Data Modeling Common Best Practices*.

> **Note**
> All DirectQuery limitations described in the *Working with connection modes* section also apply to the tables with a **Dual** storage model setting.

We can see or change the storage mode property from the **Model** view from the left pane in Power BI Desktop, as follows:

1. Click the **Model** view.
2. Select a table from the **Fields** pane.
3. Expand **Advanced** from the **Properties** pane.
4. Select a **Storage mode** from the drop-down list.

The preceding steps are highlighted in the following screenshot:

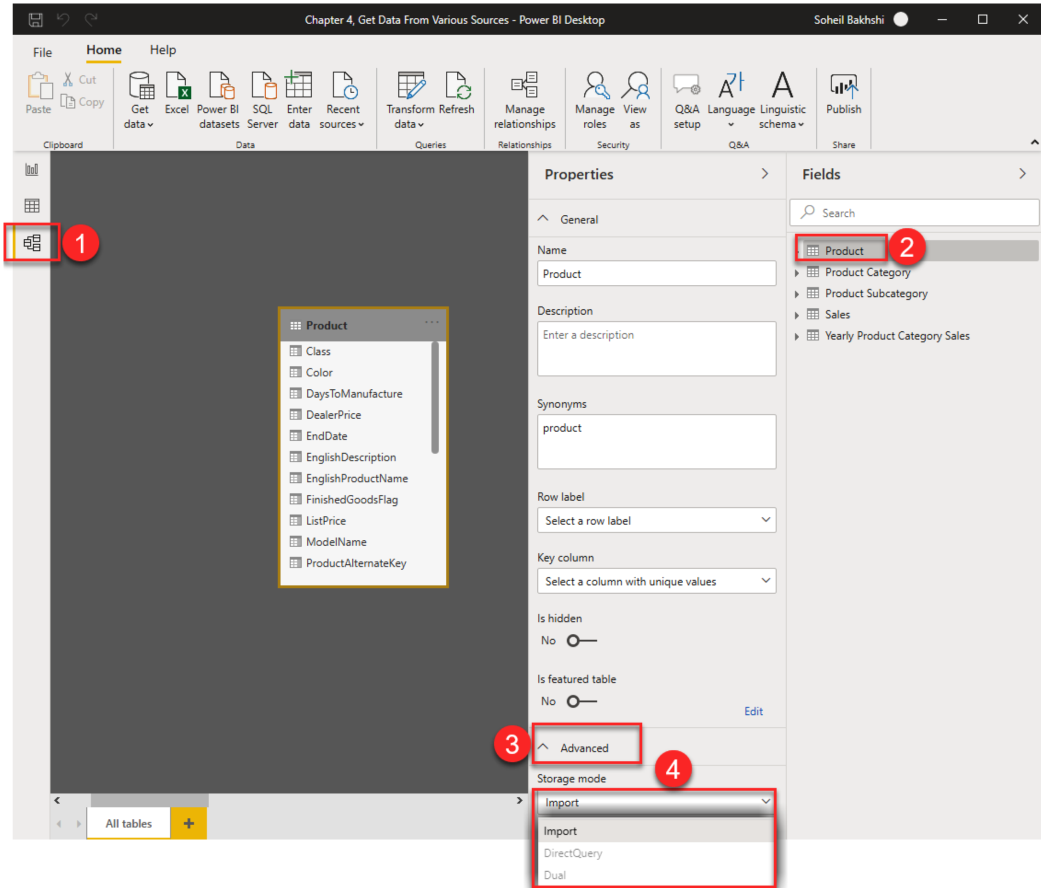


Figure 4.41 – Changing the table's Storage mode property

> **Note**
> We cannot change a table's storage mode from **Import** mode to either **DirectQuery** or **Dual** mode.

In this section, we discussed the storage modes of tables. In the next section, we learn about dataset storage modes.

Understanding dataset storage modes

As you may have already guessed, dataset storage modes refer to whether the data in a dataset is cached in the memory or not. With that in mind, from a dataset perspective, there are three different modes, as outlined next:

- **Import**: When the whole data is cached in the memory. In this mode, all tables are in the **Import** storage mode setting.
- **DirectQuery**: When the data is not cached in the memory. In this mode, all tables are in the **DirectQuery** storage mode setting.
- **Composite** (**Mixed**): When a portion of data is cached in the memory, while the rest is not. In this mode, some tables are in the **Import** storage mode setting; other tables are in **DirectQuery** storage mode or the **Dual** storage mode setting.

To see and edit the dataset storage modes in Power BI Desktop, look at the right side of the status bar, as shown in the following screenshot:

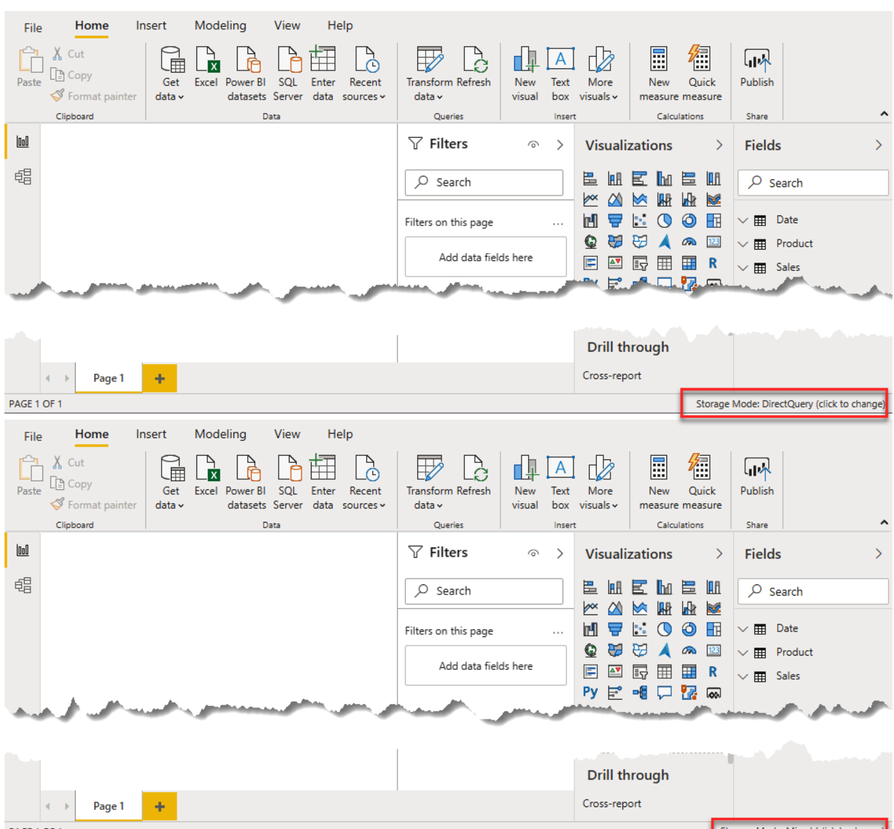


Figure 4.42 – Identifying dataset storage modes in Power BI Desktop

As you see in the preceding screenshot, you can click on the bottom right of the status bar to change the storage mode, as shown in the following screenshot. Note that the storage mode for the **Import** modes cannot be changed:

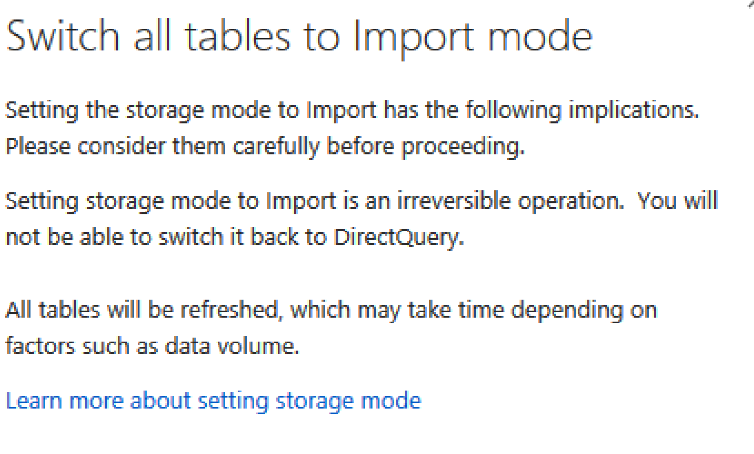


Figure 4.43 – Changing the dataset storage mode

Dataset storage mode is an essential point to think about at the beginning of the project. We need to make sure that the dataset storage mode we use in our Power BI model covers all the business needs. The dataset storage mode has a direct effect on our Power BI architecture. For instance, if a business requires minimal data latency or even real-time data analysis, **DirectQuery** would be a potential choice.

Summary

In this chapter, we learned to work with the most common data sources supported in Power BI, such as folders, CSV, Excel, Power BI datasets, Power BI dataflows, SQL Server SSAS, and OData feed, with some challenging real-world scenarios. We also went through data source certifications and discussed why it is essential to know which data source certification level we are dealing with. We then looked at connection modes, storage modes, and dataset modes, and at how different they are. It is worthwhile emphasizing the importance of understanding different connection modes, storage modes, and dataset modes as they will directly affect our data modeling and overall Power BI architecture.

In the next chapter, we look at common data preparation steps in the **Power Query Editor**, along with real-world scenarios.

5
Common Data Preparation Steps

In the previous chapter, we discussed some data sources that are frequently used in Power BI. We also covered data source certifications and the differences between various connection modes, storage modes, and dataset modes. This chapter will look at common preparation steps such as common table manipulations, common text manipulations, and common Date, DateTime, and DateTimeZone manipulations.

We will look at each of these by providing real-world scenarios that can help you deal with real daily data preparation challenges. Looking at the Power Query Editor, we can see that the data preparation activities are categorized into three separate tabs, as shown in the following screenshot:

1. **Home**: This tab contains more generic actions, such as creating a new query, creating or managing query parameters, and performing some common data preparation steps such as split column, group by, and more.
2. **Transform**: This tab contains more transformation functionalities that can be performed through the UI.
3. **Add Column**: This tab contains data preparation steps related to adding a new column through the UI:

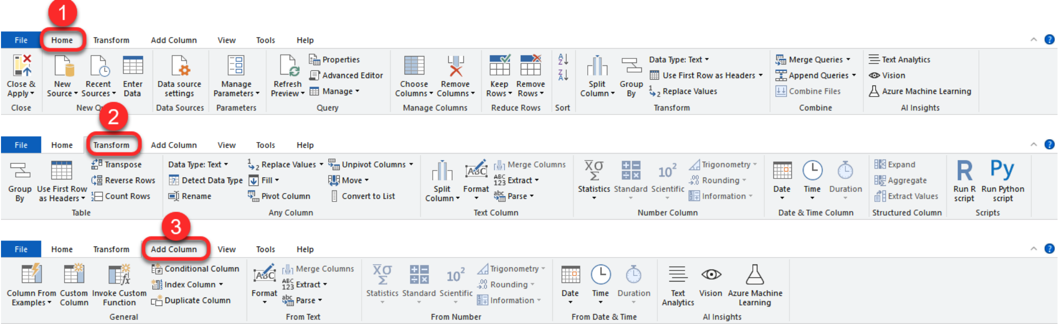


Figure 5.1 – Data preparation functionalities available via the Power Query Editor UI

In the following few sections, we will look at some of the functionalities available under the preceding tabs and some that are not available in any of the preceding tabs. However, they are commonly used during the data preparation phase of data modeling.

This chapter will use the `Chapter 5, Common Data Preparation Steps.pbix` sample file unless stated otherwise. To use the sample file, open it in Power BI Desktop, then change the values of the following query parameters:

- Adventure Works DW Excel path
- Internet Sales in Time Excel path
- Approved Subcategories List path

You can find the source files in GitHub via the following link:

`https://github.com/PacktPublishing/Expert-Data-Modeling-with-Power-BI`.

In Power BI, we can get data from various sources in different formats. Regardless of the format of the data source, the quality of the data is also super important. There are some cases when we must deal with already transformed data, but in reality, there are many more cases where we need to transform the data and prepare it for analysis in the data model. A prevalent example is when the data source is Excel or CSV, and the data is pivoted. Therefore, we need to take that data into the operation room, perform some procedures on them, massage them a bit, and prepare them to get back to life in the analytical world. In this chapter, we'll look closer at the most commonly used data transformations and data manipulation functionalities in Power Query, including the following:

- Data type conversion
- Splitting a column by delimiter
- Merging a column
- Adding a custom column
- Adding a column from examples
- Duplicating a column
- Filtering rows
- Working with Group By
- Appending queries
- Merging queries
- Duplicating and referencing queries
- Replacing values
- Extracting numbers from text
- Dealing with Date, DateTime, and DateTimeZone

Let's dive into these topics and have a look at them in more detail.

Data type conversion

Data type conversion is one of the most common steps we take in Power Query, yet it is one of the most important ones that can become tricky when it's not managed well. One cool feature of Power BI, if enabled, is that it can detect data types automatically. While this is a handy feature in many cases, it can be the root cause of some issues down the road. The critical point to note is how Power BI automatically detects data types. Power BI automatically detects column data types based on the first few hundred rows.

This is when things can go wrong, as the data types are not detected based on the whole dataset. Instead, the data types are detected based on part of it. In most cases, we deal with data type conversion in table values. Either we use the Power Query Editor UI or manually write the expressions; here, we use the following function:

```
Table.TransformColumnTypes(Table as table, TypeTransformations
as list, optional Culture as nullable text)
```

In the `Table.TransformColumnTypes()` function, we have the following:

- `Table` is usually the result of the previous step.
- `TypeTransformations` accepts a list of column names, along with their corresponding data type.

We already discussed the types available in Power Query in *Chapter 3*, *Data Preparation in Power Query Editor*, in the *Introduction to Power Query (M) Formula Language in Power BI* section, in the *Types* subsection. The following table shows the types, along with the syntax we can use to specify the data types:

Type Kind	Type Representation	Specifying Type Syntax 1	Specifying Type Syntax 2
binary	Binary	type binary	Binary.Type
date	Date	type date	Date.Type
datetime	Date/Time	type datetime	DateTime.Type
datetimezone	Date/Time/Zone	type datetimezone	DateTimeZone.Type
duration	Duration	type duration	Duration.Type
list	List	type list	List.Type
logical	Logical	type logical	Logical.Type
null	Null	type null	Null.Type
number	Whole Number	-	Int64.Type
	Decimal Number	type number	Int64.Type
	Fixed Decimal Number	-	Currency.Type
	Percentage	-	Percentage.Type
record	Record	type record	Record.Type
text	Text	type text	Text.Type
time	Time	type time	Time.Type
type	Type	type type	Type.Type
function	Function	type function	Function.Type
table	Table	type table	Table.Type
any	Any	type any	Any.Type
none	None	type none	None.Type

Table 5.1 – Power Query types, their representation, and their syntax

Let's look at this in more detail. We need to open the `Chapter 5, Common Data Preparation Steps.pbix` sample file:

1. Open **Power Query Editor** and select the `Geography` table from the **Queries** pane.
2. When we created the sample file, we did not change the data types. The last step is automatically created, and the data types are automatically detected.
3. Scroll the **Data** view to the right. Here, we can see that the **Column Quality** bar of the `PostCode` column turned red, which means there is an error in the sample data (the top 1,000 rows).
4. Scroll down a bit in the **Data** view to find an erroneous cell. Click a cell that's producing an error to see the error message.

5. As the following screenshot shows, the error was caused by an incorrect data type conversion:

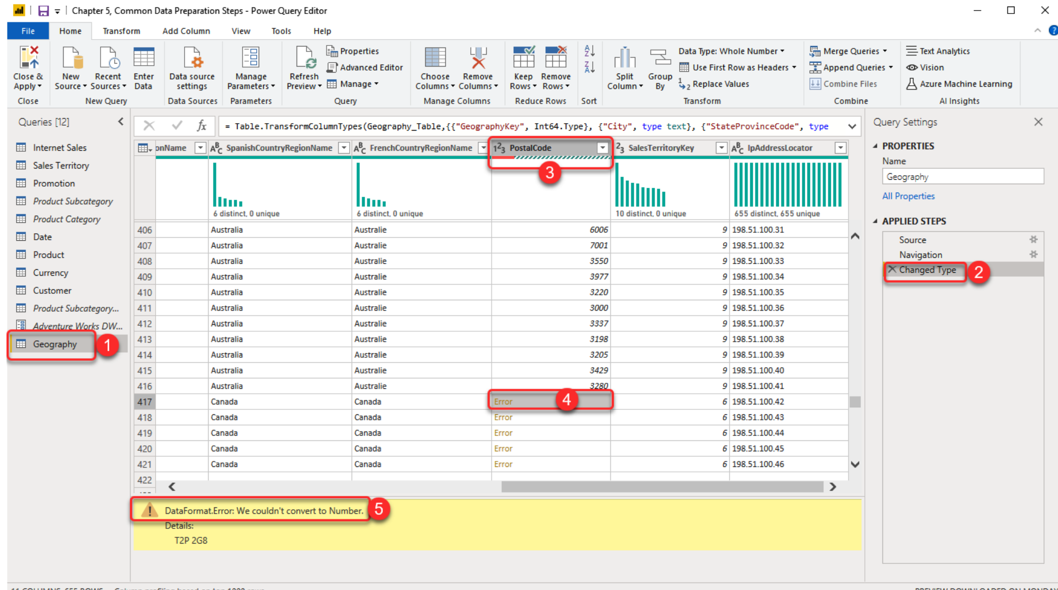


Figure 5.2 – Errors caused by wrong data type detection by Power BI

Fixing the issue is straightforward. Here, we must set the `PostCode` column data type to `text`, which is the correct data type.

6. Click the column type indicator button.
7. Click **Text**.
8. Click the **Replace current** button on the **Change Data Type** message:

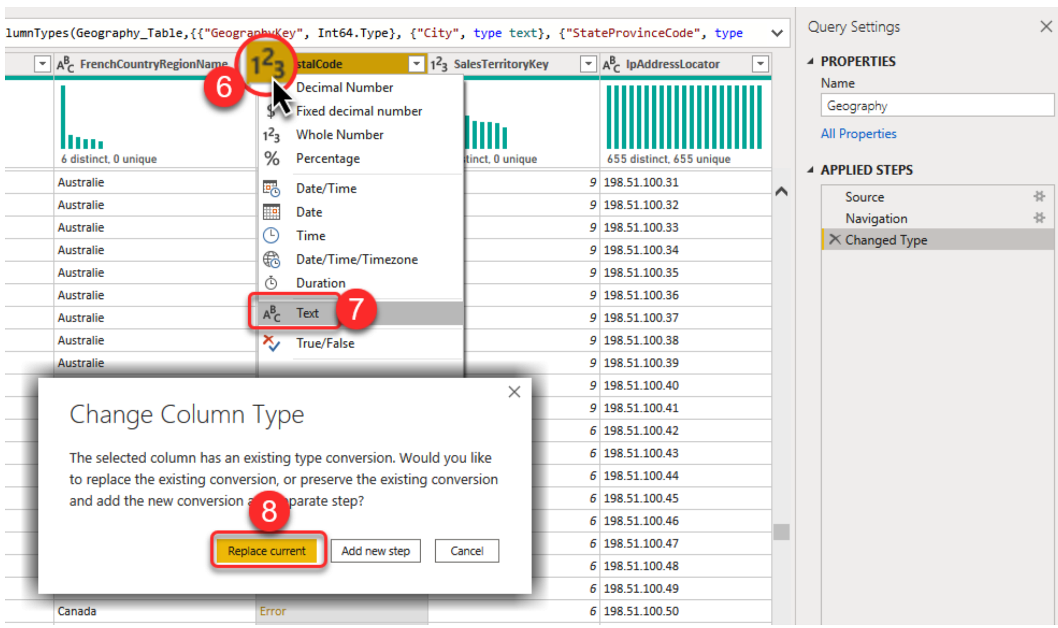


Figure 5.3 – Changing a column data type

As shown in *step 8*, we do not get a new step in **Applied Steps**, but the issue is resolved. In the following screenshot, **Column distribution** shows no indication of any issues anymore:

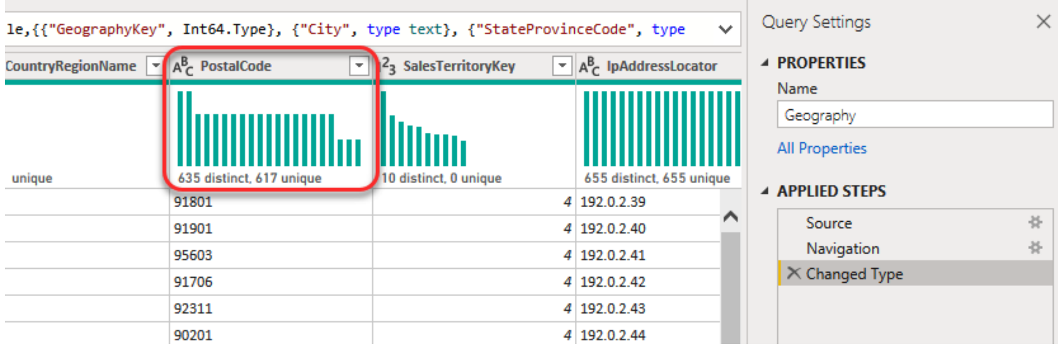


Figure 5.4 – Data type conversion issue resolved

In the preceding example, we quickly found and fixed the issue. But in many cases, data conversion issues and their fixes are not as trivial, even when we do not use the automatic data type detection feature. Let's continue using this sample file and load the data into the data model after fixing the data type issue with the `PostCode` column.

Let's put a table on the report canvas with the `Date` column from the `Date` table. We'll see that the `Internet Sales` measure and another issue quickly appear. As shown in the following screenshot, the `Internet Sales` values haven't been sliced correctly by the **Date** values:

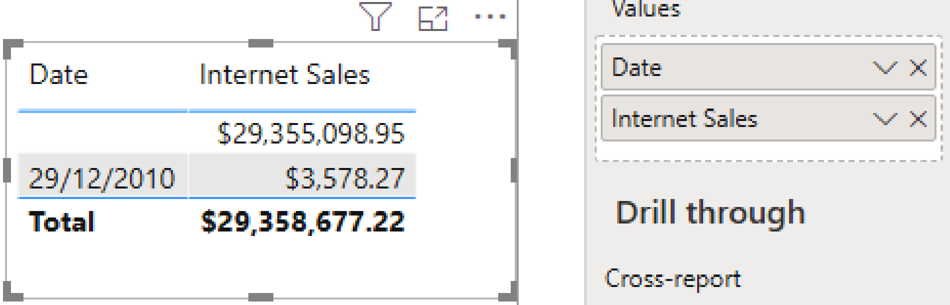


Figure 5.5 – The Internet Sales measure is sliced by one Date only

In cases like this, a few things may go wrong that lead to incorrect values. So, we usually go through some initial checks to narrow down the possibilities and find the root cause(s) of the issue. The following are some of them:

- We look at the **Model** tab from the left pane to review the relationships.
- We look at the related columns to see if there are any missing values.
- We check the related columns to make sure the data types have been selected correctly.

We know that the `Date` column from the `Date` table and the `OrderDateTime` column in the `Internet Sales` table participate in a relationship between the `Date` and `Internet Sales` tables. The following screenshot shows the preceding relationship:

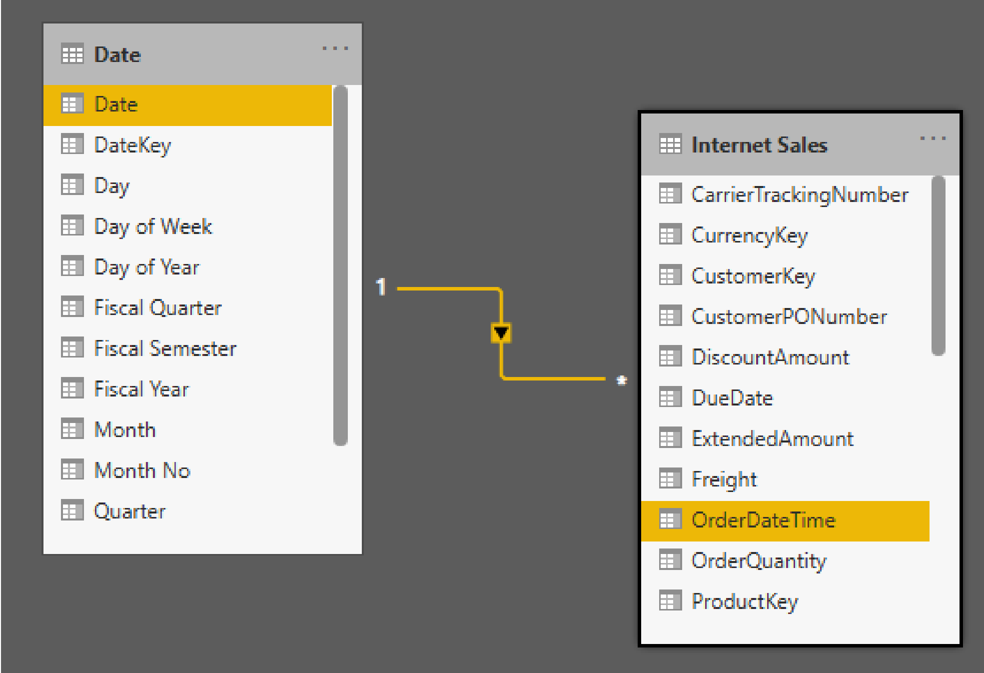


Figure 5.6 – Relationship between the Date and Internet Sales tables

Both columns also contain data, as shown in the following image:

Figure 5.7 – The Date column from the Date table and the OrderDateTime column from the Internet Sales table contributing to the relationship's data

Looking at the data types of both the `OrderDateTime` column in `Internet Sales` and the `Date` column in the `Date` table reveals that the data type of the `Date` column is `Date`, while the data type of the `OrderDateTime` column is `DateTime`. Those data types are compatible from a data modeling perspective. Nevertheless, the data in the `OrderDateTime` column has a time element. Therefore, the only matching values from the `OrderDateTime` column are those with a time element of **12:00:00 AM**. To make sure this is right, we can do a quick test.

We can put three tables on the reporting canvas, as follows:

- *Table1*: Shows the `Date` column from the `Date` table
- *Table2*: Shows the `OrderDateTime` column from the `Internet Sales` table
- *Table3*: Shows the `Date` column from the `Date` table and the `OrderDateTime` column from the `Internet Sales` table side by side

The following screenshot shows the results:

Table1

Date
1/01/2005
2/01/2005
3/01/2005
4/01/2005
5/01/2005
6/01/2005
7/01/2005
8/01/2005

Table2

OrderDateTime
29/12/2010 12:00:00 a.m.
29/12/2010 11:04:40 a.m.
30/12/2010 4:21:59 a.m.
31/12/2010 9:39:18 a.m.
1/01/2011 2:56:37 a.m.
2/01/2011 8:13:56 a.m.
3/01/2011 1:31:15 a.m.
4/01/2011 6:48:34 a.m.

Table3

Date	OrderDateTime
29/12/2010	29/12/2010 12:00:00 a.m.
	29/12/2010 11:04:40 a.m.
	30/12/2010 4:21:59 a.m.
	31/12/2010 9:39:18 a.m.
	1/01/2011 2:56:37 a.m.
	2/01/2011 8:13:56 a.m.
	3/01/2011 1:31:15 a.m.

Figure 5.8 – Date and OrderDateTime do not match

As you can see, there is only one match between the two columns. Now that we've identified the issue, the only step we need to take is to convert the `OrderDateTime` column's data type from `DateTime` into `Date`, as shown in the following image:

1. Open the Power Query Editor and select the `Internet Sales` table from the **Queries** pane.
2. Right-click the `OrderDateTime` column.
3. Click `Date` from the context menu under the **Change Type** submenu.

4. Click the **Replace current** button:

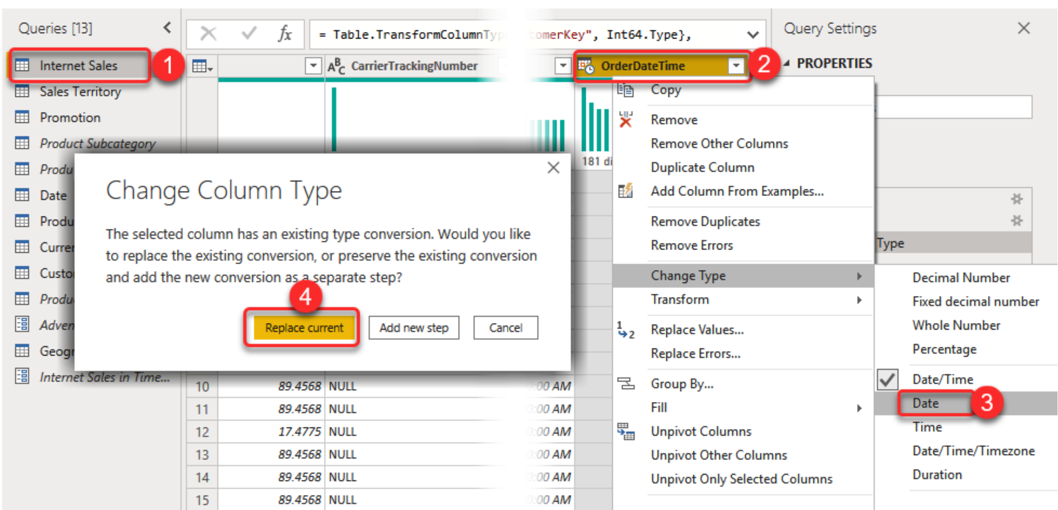


Figure 5.9 – Changing the column's data type from DateTime to Date

Now, let's switch to the **Report** view and see the results. As shown in the following screenshot, the issue has been resolved:

Date	Internet Sales
29/12/2010	$14,477.34
30/12/2010	$13,931.52
31/12/2010	$15,012.18
1/01/2011	$7,156.54
2/01/2011	$15,012.18
3/01/2011	$14,313.08
4/01/2011	$7,855.64
5/01/2011	$7,855.64
6/01/2011	$20,909.78
7/01/2011	$10,556.53
8/01/2011	$14,313.08
Total	**$29,358,677.22**

Figure 5.10 – The correct results after changing the OrderDateTime column's data type from DateTime to Date

As the preceding sample clearly shows, selecting an incorrect data type can significantly affect our data model. So, the critical point is that the key columns in both tables that are used in a relationship must contain the same data type.

Splitting column by delimiter

One of the most common transformation steps is **Split column by delimiter**. In many cases, we may need to split a column by a delimiter, such as when we have people's full names in our data. However, the business needs to have separate `First Name`, `Middle Name`, and `Last Name` columns. Let's look at an example. In the previous section, we converted the `OrderDateTime` column's type into `Date`. But what if the business requires us to analyze the `Internet Sales` data at both the `Time` and `Date` levels? We can do many things to satisfy this new requirement, such as the following:

- Create a new `Time` table, which can be done either using DAX (we discussed this in *Chapter 2*, *Data Analysis eXpressions and Data Modeling*, in the under *Creating a Time dimension with DAX* section) or within the Power Query Editor.
- Split the `OrderDateTime` column into two columns – one `Date` column and one `Time` column.
- Create a relationship between the `Time` and the `Internet Sales` tables.

We'll only look at the second option in this scenario; that is, splitting the `OrderDateTime` column by a delimiter. However, we've already converted the `OrderDateTime` column into `Date`. So, first, we need to convert it back into `DateTime`. We can take the same steps that we took in the previous section and change the data type of the `OrderDateTime` column to `DateTime` from the Power Query Editor. Once we've do that, we can go through the following steps:

1. In **Power Query Editor**, select the `Internet Sales` table from the left pane.
2. Select the `OrderDateTime` column.
3. Click the **Split Column** drop-down button from the **Transform** tab. This will give us seven different splitting options.
4. Click **By Delimiter**.
5. Select **Space** from the **Select or enter delimiter** drop-down list.
6. Tick the **Left-most delimiter** option.

7. Click **OK**:

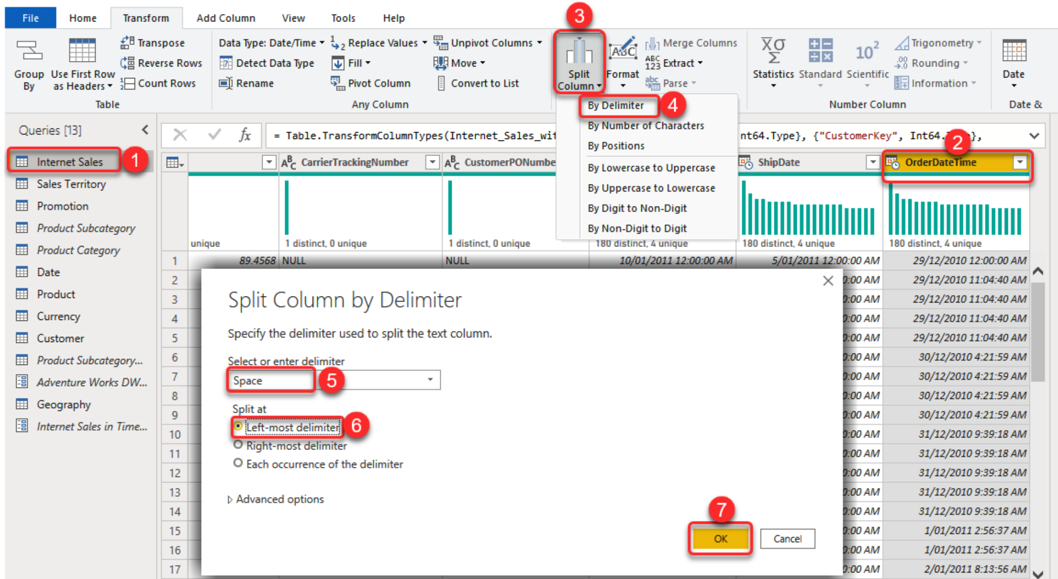


Figure 5.11 – Split column by delimiter

So far, we've added a new transformation step named **Split Column by Delimiter** to split the `OrderDateTime` column into two columns named (by default) `OrderDateTime.1` and `OrderDateTime.2`. The following screenshot illustrates the results:

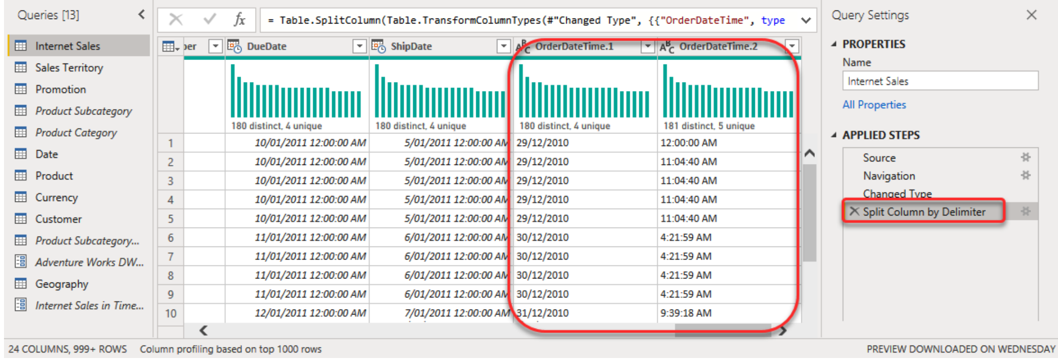


Figure 5.12 – Split column creates two new columns called OrderDateTime.1 and OrderDateTime.2

We will rename the columns in a separate step.

> **Good Practice**
>
> Avoid creating excessive transformation steps when possible. More transformation steps translates into more data processing time and extra load on the Power Query engine.

With the preceding note in mind, we do not rename the two new columns as a new step. Instead, we change the Power Query expression of the **Split Column by Delimiter** step. There are two ways to do so:

A: Change the expressions from **Advanced Editor**, as shown in the following image:

1. Click **Advanced Editor** from the **Home** tab on the ribbon bar.
2. Find the **#"Split Column by Delimiter"** step.
3. Scroll to the right.
4. Change the column names.
5. Click **Done**:

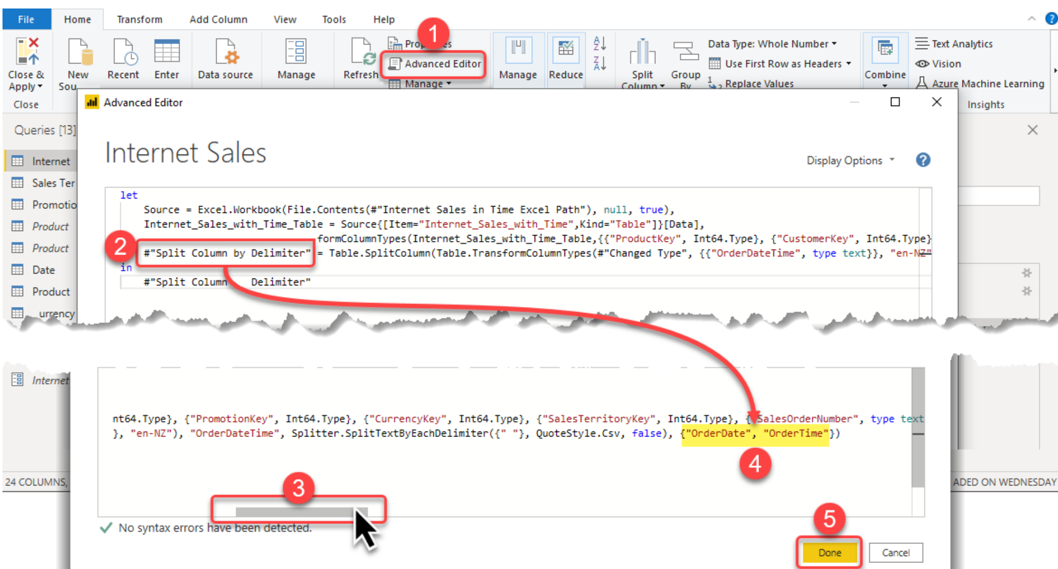


Figure 5.13 – Changing the default column names of the splitter columns from the Advanced Editor

B: Change the expressions from **Formula Bar**, as shown in the following screenshot:

1. Click the **Split Column by Delimiter** step from **Applied Steps**.
2. Click the down arrow on **Formula Bar** to expand it.
3. Change the names of the columns.

4. Click the **Submit** (✓) button:

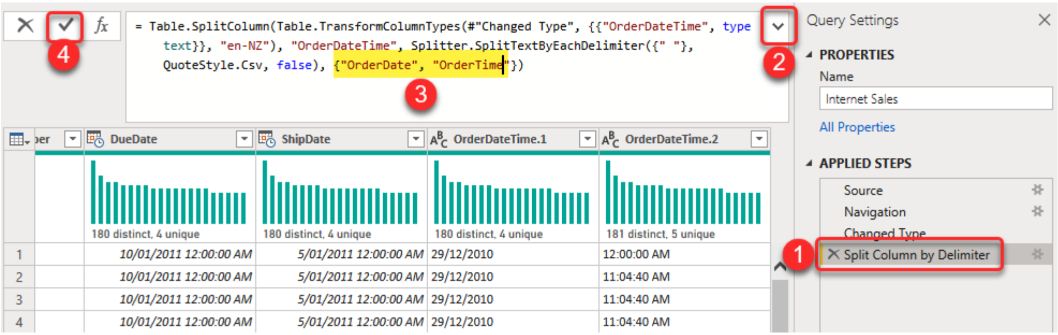


Figure 5.14 – Changing the default column names of the splitter columns from the Formula Bar

The last thing we need to do is change the `OrderDate` column's data types to `Date` and the `OrderTime` column to `Time`.

Merging columns

A typical transformation under the **Add Column** category is **Merge Columns**. There are many use cases where we need to merge the data that's been spread across different columns, such as merging `First Name`, `Middle Name`, and `Last Name` to create a `Full Name` column, or merging a multipart address that's being held in separate columns (`AddressLine1`, `AddressLine2`) to get an `Address` column containing the full address. Another common use case is to merge multiple columns to create a unique ID column. Let's continue with an example from the `Chapter 5, Common Data Preparation Steps.pbix` sample file:

1. Select the `Customer` table from the **Queries** pane of **Power Query Editor**.
2. Select the `First Name`, `Middle Name`, and `Last Name` columns.
3. Right-click one of the selected columns and click **Merge Columns**. Alternatively, we can click the **Merge Column** button from the **Transform** tab (shown in yellow in the following image).
4. Select **Space** from the **Separator** dropdown.
5. Type `Full Name` into the **New column name** text box.

6. Click **OK**:

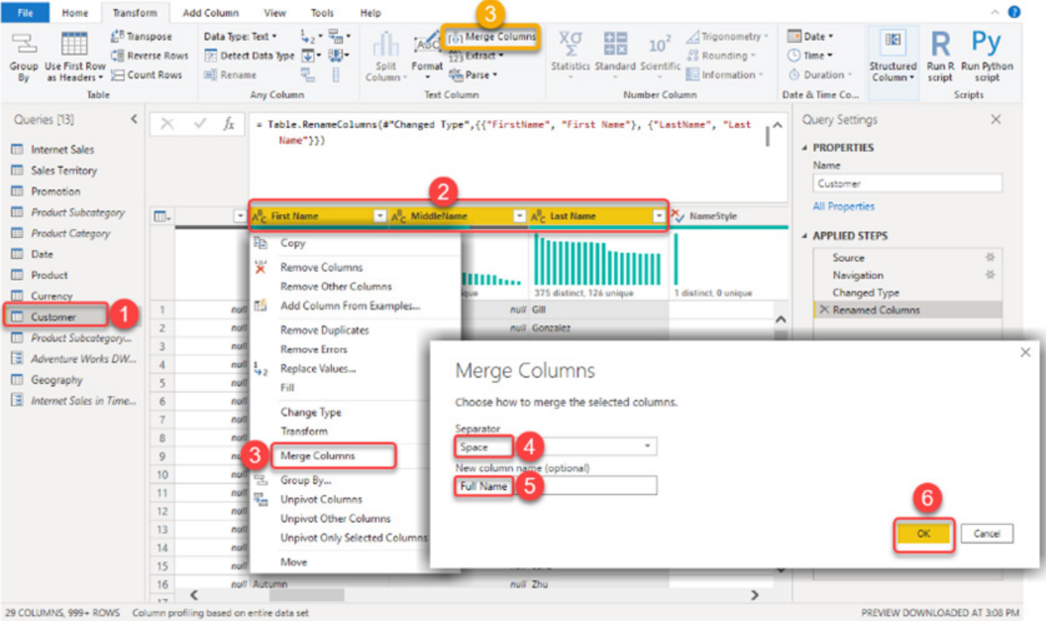


Figure 5.15 – Merge Columns popup

After merging the three columns into one column, the output will look as follows:

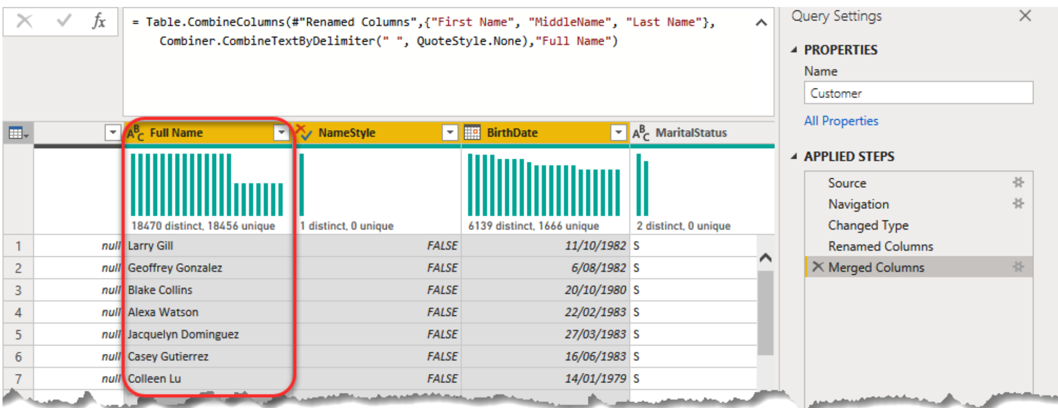


Figure 5.16 – The three selected columns merged into one single column named Full Name

Adding a custom column

In my experience, adding a new column is one of the most common steps that we take during the data preparation phase. There are countless scenarios where we need to add a new column, such as adding some new analytical equations as a new column, creating data clusters in a new column, adding an index column as a new column, or using some **machine learning** (**ML**) and **artificial intelligence** (**AI**) algorithms. You may also have many other scenarios in mind. Whether we use the Power Query Editor UI or manually write the Power Query expressions, we must add a custom column using the following function:

```
Table.AddColumn(Table as table, NewColumnName as text,
ColumnGenerator as function, optional ColumnType as nullable
type)
```

In the `Table.AddColumn()` function, we have the following:

- `Table`: This is the input table value, which can be the result of the previous step or other queries that provide table output.
- `NewColumnName`: This is quite self-explanatory – it's the new column name.
- `ColumnGenerator`: The expressions we use to create a new column.
- `ColumnType`: This is used to specify the data type of the new column. This is an optional operand, but it is a handy one. More on this later in this section.

Let's continue with a scenario. In the `Chapter 5, Common Data Preparation Steps.pbix` sample file, we need to add a column to the `Customer` table to show if the customer's annual income is below or above the overall average income. To do so, we need to calculate the average annual income first. We have the annual income of all customers captured in the `YearlyIncome` column. To calculate the average income, we must reference the `YearlyIncome` column and calculate the average. We can reference a column within the current table by referencing the step's name, along with the column's name. So, in our case, because we want to get the average of `YearlyIncome`, the Power Query expression will look like this:

```
List.Average(#"Merged Columns"[YearlyIncome])
```

In the preceding expression, `#"Merged Columns"` is the name of the previous step. The result of referencing a column supplies a list of values, so by using the `List.Average(as list, optional precision as nullable number)` function, we can get the average of the values of a list. In our example, this is the `YearlyIncome` column.

Let's add a new custom column by following these steps:

1. Select the `Customer` table from the **Queries** pane.
2. Click the **Custom Column** button from the **Add Column** tab of the ribbon.
3. Type in a name for the new column.
4. Type in the expression shown in the following lines of code:

```
if [YearlyIncome] <= List.Average(#"Merged
Columns"[YearlyIncome]) then true else false
```

5. Click **OK**:

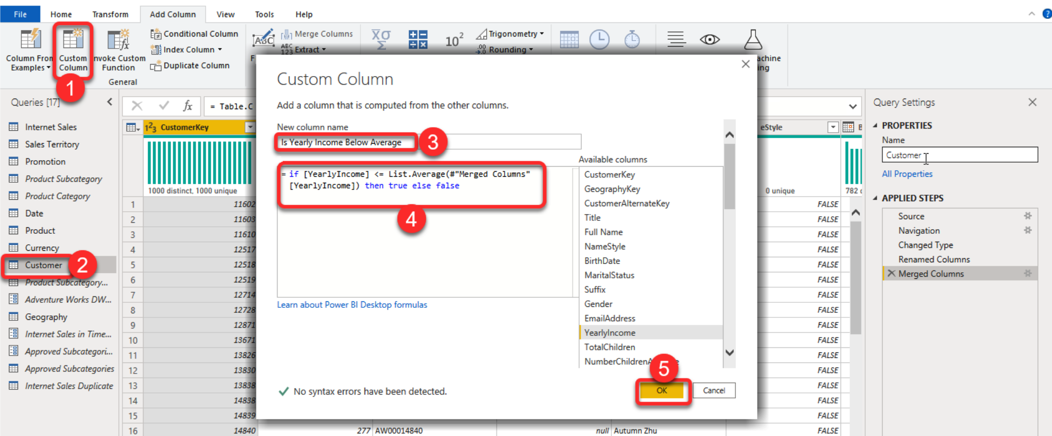


Figure 5.17 – Adding a new custom column

In the preceding screenshot, we are specifying the operands of the `Table.AddColumn()` function within the Power Query Editor UI, like so:

- Number **3** is the `NewColumnName` operand
- Number **4** is the `ColumnGenerator` operand

The preceding steps result in a new custom column with `TRUE` or `FALSE` values indicating whether the customer's yearly income is below the average of all customers' yearly income. The following screenshot shows the results of the preceding steps:

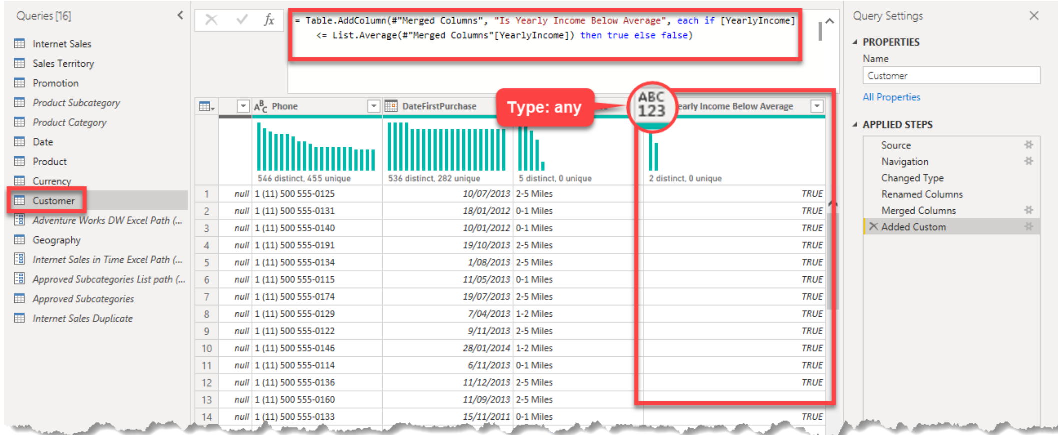


Figure 5.18 – Result of adding a new custom column

As you can see, the new column's data type is `any`, while we expect the output to be `logical`. So, here, we have two options. The most trivial one that most developers do is add another **Changed Type** step to convert the new column into `logical`. This is not a good practice. What if we need to add some more custom columns? Are we going to add a **Changed Type** step after every new custom column? No, we do not need to add any extra steps after adding a new custom column. The second option, which is indeed a best practice to go for, is to use the `ColumnType` optional operand of the `Table.AddColumn()` function. The following expression shows the use of the `ColumnType` optional operand within the `Table.AddColumn()` function:

```
Table.AddColumn(#"Merged Columns", "Is Yearly Income Below
Average", each if [YearlyIncome] <= List.Average(#"Merged
Columns"[YearlyIncome]) then true else false, type logical)
```

The output now looks as follows, without us adding any new steps:

Figure 5.19 – Specifying the data type of the new custom column

We can use this column in our data model. We can then use this column in our data visualizations to analyze the data that's relevant to the customers' yearly income.

Adding column from examples

Adding column from examples is a brilliant feature of Power Query. It not only helps speed up the development process but also helps developers learn Power Query. The idea is that we can create a new column from sample data by entering the expected values in a sample column. Power Query then guesses what sort of transformation we are after and generates the expressions needed to achieve the results we entered manually. We can create new columns from selected columns or by all columns. Let's have a quick look at this feature by example.

We want to extract the usernames of the customers from their `EmailAddress` column, while the email structure is `UserName@adventure-works.com`, from the `Customer` table. The following steps show how we can achieve this by adding a column from examples:

1. Select the `Customer` table from the **Queries** pane.
2. Select the `EmailAddress` column.
3. Click the **Column From Examples** drop-down button from the **Add Columns** tab of the ribbon.

4. Click the **From Selection** option:

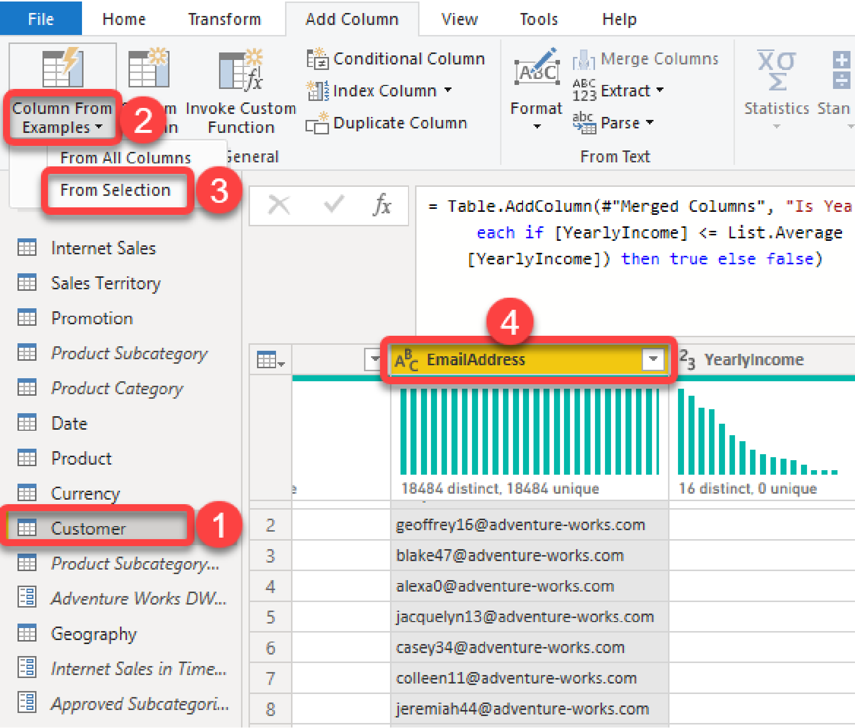


Figure 5.20 – Adding a column from examples

5. Type in some expected results by double-clicking a cell and typing in a value, as shown here:

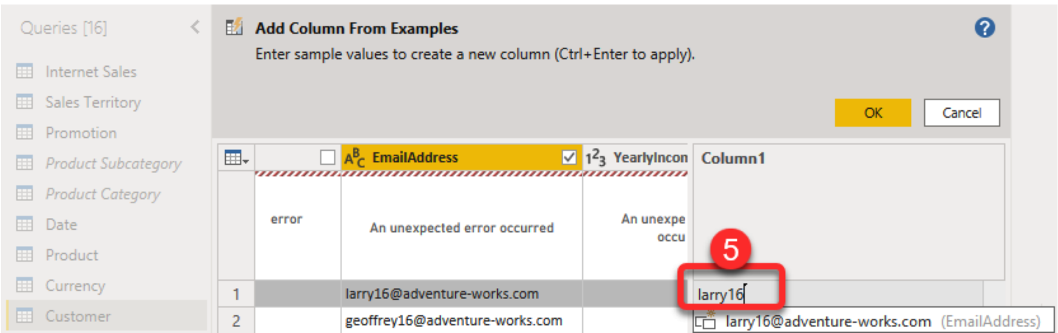


Figure 5.21 – Entering example values

6. Press *Enter* on your keyboard. At this point, if Power Query correctly guessed what we are after, we can enter a name for the new column. Otherwise, we must continue entering more examples to help Power Query making a better guess.
7. Click **OK**:

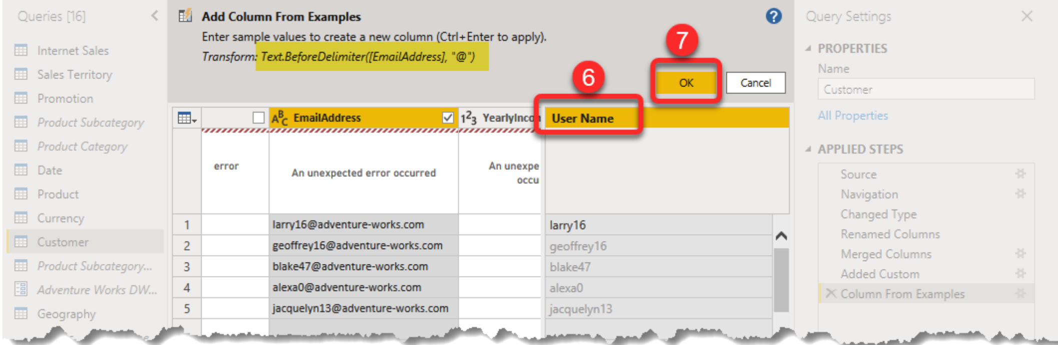


Figure 5.22 – Naming the new column and confirming the new column from example

Note the section highlighted in the preceding image. We can learn how to write Power Query expressions by looking at the Power Query expressions appearing in the highlighted section. One of the drawbacks of this method is that it only allows us to enter the example for a couple of rows, which does not make the best pattern for Power Query to guess what we are after. The other downside is that Power Query cannot guess the logic of the entered examples in complex cases. Therefore, it does not work properly.

Duplicating a column

Another common transformation step under the **Add Column** tab is duplicating a column. In many scenarios, we may wish to duplicate a column, such as when we want to keep the original column available in our model while we need to transform it into a new column. Let's revisit the scenario that we looked at earlier in this chapter in the *Split column by delimiter* section. In that scenario, we split the `OrderDateTime` column from the `Internet Sales` table into two columns, `Order Date` and `Order Time`. In this section, we will do this another way. We will work on a duplicate table we created from `Internet Sales` before splitting the `OrderDateTime` column for this scenario. We will name the new table `Internet Sales Duplicate`.

> **Note**
>
> Duplicating a query will be explained in more detail in the *Duplicating and referencing queries* section of this chapter.

The following steps show how to achieve this:

1. Select the `Internet Sales Duplicate` table from the **Queries** pane.
2. Select the `OrderDateTime` column.
3. Click the **Duplicate Column** button from the **Add Column** tab of the ribbon (the duplicate column can also be accessed by right-clicking the column). This creates a new column called `OrderDateTime - Copy`:

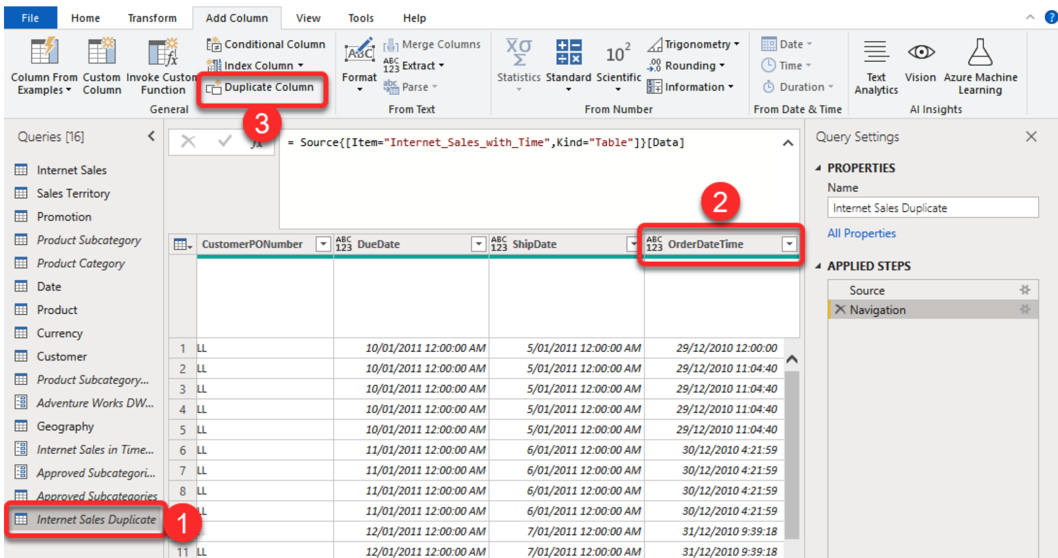


Figure 5.23 – Selecting and duplicating a column

4. Change the type of the `OrderDateTime` column to `Date`.
5. Change the type of the `OrderDateTime - Copy` column to `Time`:

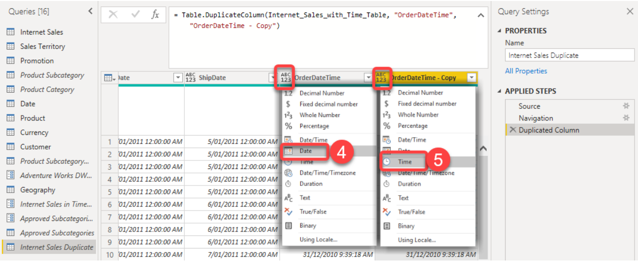


Figure 5.24 – Changing the columns' types

6. Rename the `OrderDateTime` column to `Order Date` and rename the `OrderDateTime - Copy` column to `Order Time`:

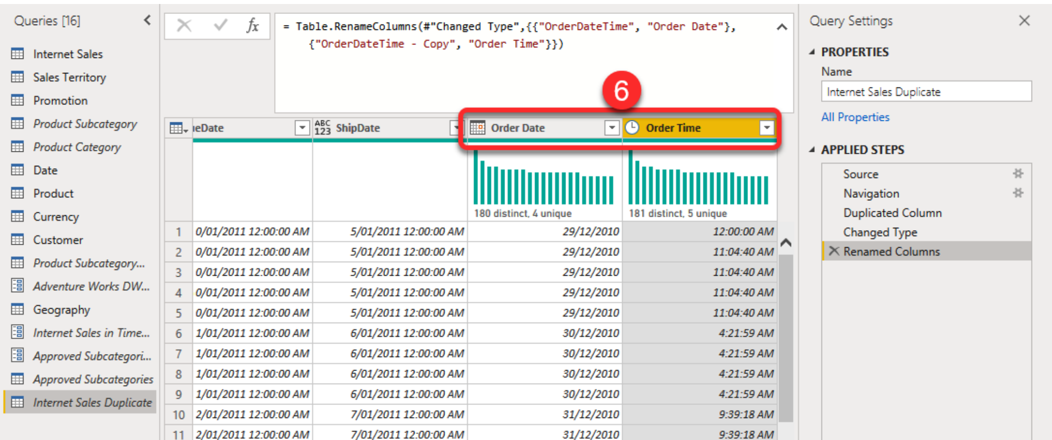


Figure 5.25 – Renaming the columns

We could rename the `OrderDateTime - Copy` column in **Duplicated Column** itself by changing the expression from `Table.DuplicateColumn(Internet_Sales_with_Time_Table, "OrderDateTime", "OrderDateTime - Copy")` to `Table.DuplicateColumn(Internet_Sales_with_Time_Table, "OrderDateTime", "Order Time")` to reduce the transformation steps. However, since we also need to rename the `OrderDateTime` column, it makes sense to rename both columns in a single step.

> **Note**
>
> This approach has no advantages over the previous approach, where we *Split column by delimiter*. So, it is down to the developer's preference to take on any of the explained approaches.

Filtering rows

The other common transformation is *Filtering rows*. There are many use cases where we may want to restrict the results by specific values. For instance, we may want to filter the `Product` table to show the products with a `Status` of `Current`. Filtering the rows based on columns' values is very simple. We have to select the desired column, then click the arrow down button (⏷) from the column's caption, and select the values we want to use for filtering the rows. This is shown in the following screenshot:

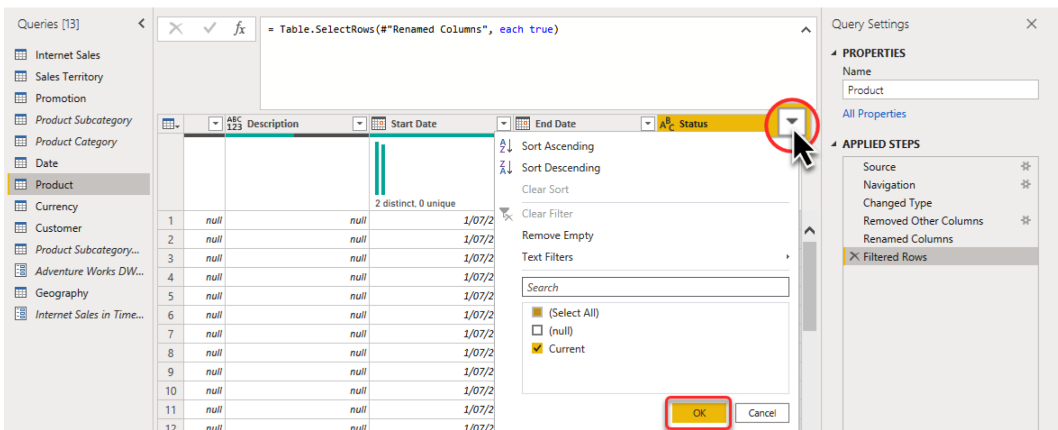


Figure 5.26 – Filtering rows

While this is a straightforward step to take, not all filtering use cases are simple, such as when we do not have specific values to filter the rows upon. However, we have a list that the business provided, specifying the values to use in the filters. Let's look at this with a scenario.

The business provides a list of **Approved Product Subcategories** every season in Excel format. We need to filter the `Product` table on the `Product Subcategory` column by the `Approved Product Subcategories` column from the `Approved Subcategories` table. We've already connected the `Chapter 5, Common Data Preparation Steps.pbix` sample file to the `Approved Subcategories List.xlsx` file, as shown in the following screenshot:

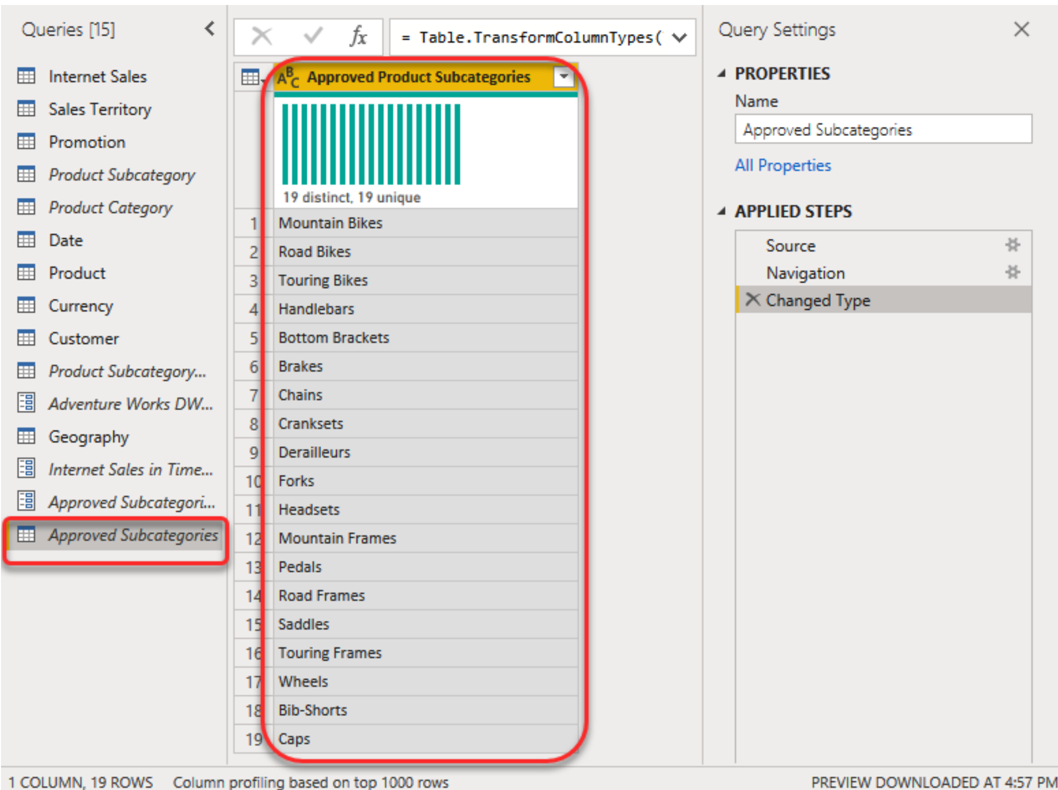


Figure 5.27 – Approved Subcategories

To filter a column in a table by the values of a column from another table, we need to know how to reference a column from another table. We can reference a column from another table like so:

```
#"Query_Name"[Column_Name]
```

The result of the preceding structure is a `list` value. In our scenario, the `Approved Product Subcategories` column from the `Approved Subcategories` table filters the `Product Subcategory` column from the `Product` table. So, we can use the `List.Contains(list, values)` function to get the matching values with the following structure:

```
List.Contains(#"Referenced_Table"[Referenced_Column], [Column_
to_be_Filtered]
```

So, the `List.Contains()` function in our scenario looks like this:

```
List.Contains(#"Approved Subcategories"[Approved Product
Subcategories], [Product Subcategory])
```

We can use the Power Query Editor UI to filter the `Product Subcategory` column from the `Product` table with a dummy value. Then, we can replace that value in the code with the `List.Contains()` function. Follow these steps to get this done:

1. Select the `Product` table from the **Queries** pane.
2. Filter the `Product Subcategory` column by any value; we used **Bike Racks** in the filter.
3. Click **OK**:

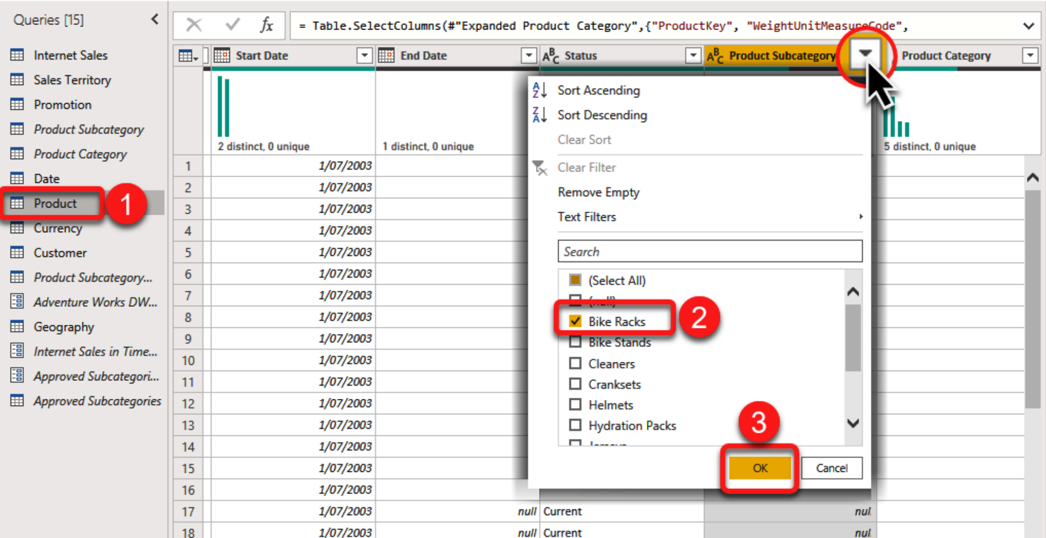


Figure 5.28 – Filtering the rows of the Product table by a value of the Product Subcategory column

The following is the Power Query expression that the UI generates:

```
= Table.SelectRows(#"Removed Other Columns", each
([Product Subcategory] = "Bike Racks"))
```

As shown in the following screenshot, the generated expression looks like this:

Figure 5.29 – Power Query expression generated by the Power Query Editor UI

The `Table.SelectRows(table, condition)` function accepts a table, which is the previous step, named `#"Removed Other Columns"`, and a condition, called `each ([Product Subcategory] = "Bike Racks")`. So, here is how we read the preceding code in natural English:

Select rows from the `#"Removed Other Columns"` step where each value in the `Product Subcategory` column is `"Bike Racks"`.

We want to change the preceding code so that we can select rows from the `#"Removed Other Columns"` step where `each` value in the `Product Subcategory` column is contained in the `Approved Product Subcategories` column from the `Approved Subcategories` table.

Now, let's continue.

4. We need to change the `condition` parameter of the `Table.SelectRows(table, condition)` function from `([Product Subcategory] = "Bike Racks")` to `List.Contains(#"Approved Subcategories"[Approved Product Subcategories], [Product Subcategory])`, as shown here:

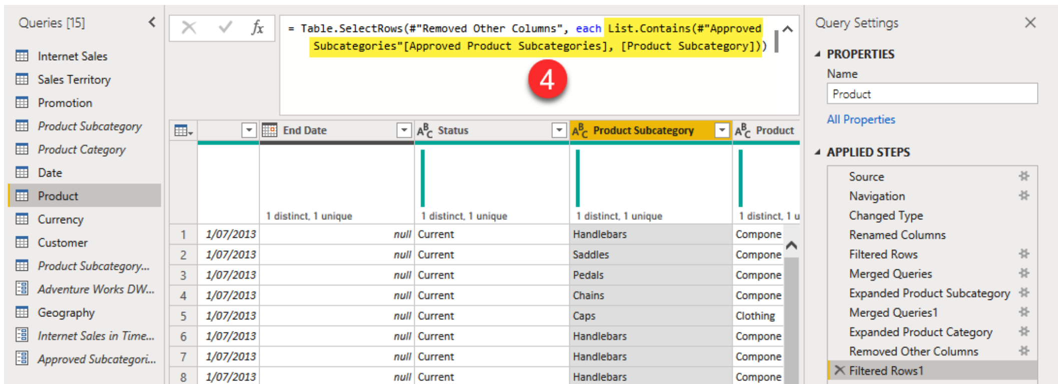


Figure 5.30 – Changing the condition of the Table.SelectRows function

The `Product` table is now being filtered by the values of the `Approved Product Subcategories` column from the `Approved Subcategories` table.

Working with Group By

One of the most valuable and advanced techniques in data modeling is creating summary tables. In many scenarios, using this method is very beneficial. We can use this method to manage our Power BI file's size; it also improves performance and memory consumption. Summarization is a known technique in data warehousing where we want to change the granularity of a fact table. But in Power Query, there are other cases where we can use the `Group By` functionality to cleanse data. Nevertheless, from a data modeling point of view, we summarize a table by grouping by some descriptive columns and aggregating the numeric values.

Let's go through a scenario and see how the `Group By` functionality works.

> **Note**
> The `Group By` operation changes the table's structure. To keep the `Internet Sales` table unchanged for the future scenarios in this chapter, we will reference the `Internet Sales` table and name it `Internet Sales Summary`. We'll use that table in the following steps.

We want to summarize the `Internet Sales Summary` table by the following columns:

- `ProductKey`
- `CustomerKey`
- `SalesTerritoryKey`
- `Order Date`

Then, we would like to aggregate the following columns by the `Sum` operation:

- `SalesAmount`
- `OrderQuantity`
- `TaxAmt`
- `Freight`

To achieve the goal of the preceding scenario, follow these steps:

1. Select the `Internet Sales Summary` table from the **Queries** pane.
2. Select the columns mentioned previously that are participating in the `group by` action.
3. Click the **Group By** button from the **Transform** tab of the ribbon.
4. Type in `Sales Amount` for **New column name**.
5. Select the `Sum` operation.
6. Select the `SalesAmount` column.
7. Click the **Add aggregation** button.
8. Repeat *steps 4* to *8* to add the other aggregations.

9. Click **OK**:

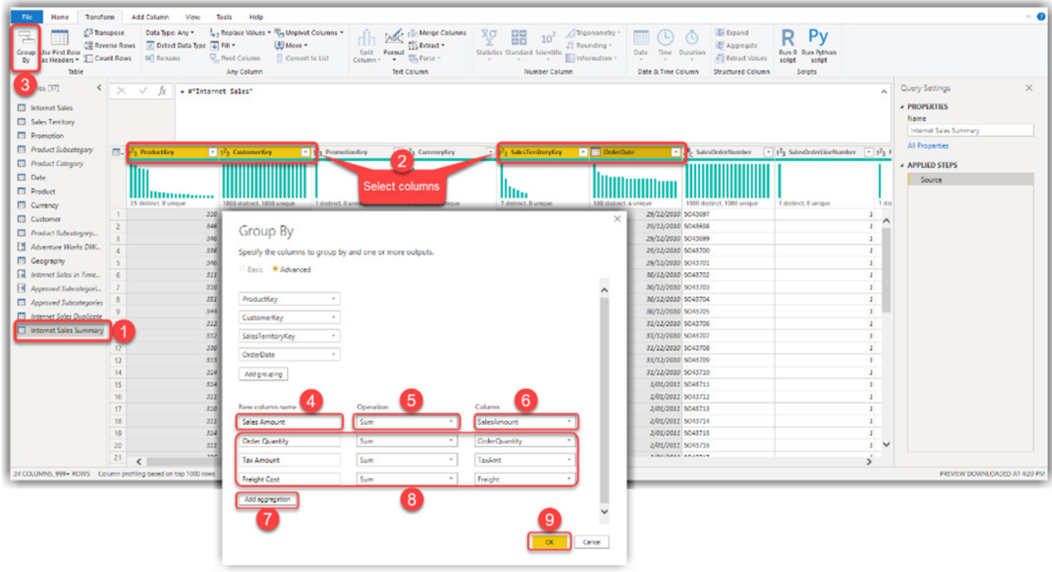


Figure 5.31 – Group By columns in Power Query

The result of the preceding operation will look as follows:

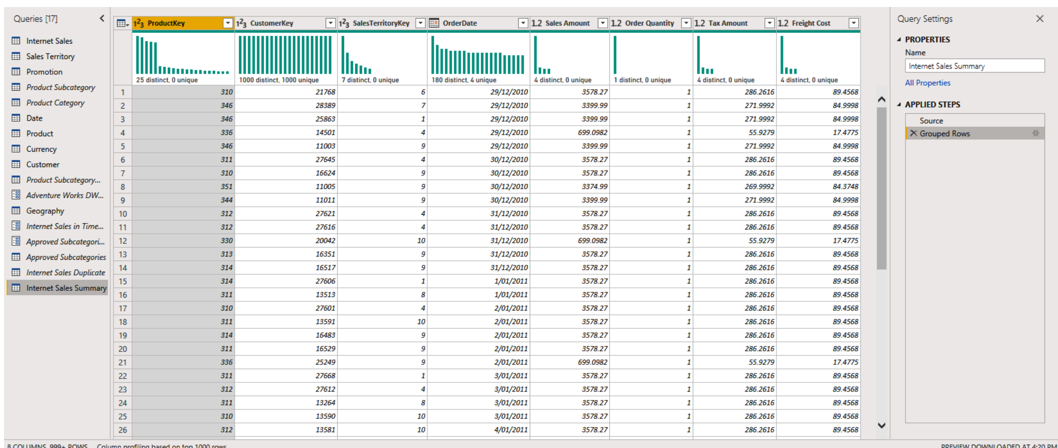


Figure 5.32 – Result of the Group By operation

We can then load the summary table into the model and create a relationship. Usually, the summary table has much fewer rows than the original table. In some cases, due to business requirements, we must unload the original table from the data model and only use the summary table.

Appending queries

There are some scenarios where we get data with the same structure from different sources, and we want to consolidate that data into a single table. In those cases, we need to append the queries. We have two options to append the queries:

- Append the queries to the first query
- Append the queries as a new query

The latter is prevalent when we follow ETL best practices. We unload all the queries, append the queries as a new query, and load them into the data model. Therefore, all the unloaded queries work as ETL pipelines. This does not mean that the first option is not applicable.

Suppose we have a simple business requirement that can be achieved by appending two or more queries to the first query. In that case, we may wish to use the first option instead. The critical point to note when we're appending queries is that the `Table.Combine(tables as list, optional columns as any)` function accepts a list of tables. When the column names in the tables are the same, it appends the data under the same column name, regardless of the columns' data types. Therefore, if we have two tables and both tables have a `Column1` column, then the data of those two columns, regardless of their data types, will be appended with the same column name.

If the data types do not match, then the data type will be `any`. Remember, Power Query is case sensitive. Therefore, if we have the `column1` column in `Table1` and the `Column1` column in `Table2`, those columns will be treated as different columns when we append the two tables. So, we end up having a table with both `column1` and `Column1` columns. Let's look at an example. For this example, we will use `Chapter 5, Append Queries.pbix` sample file, which contains sales data for different years in different file formats. As shown in the following screenshot, we have three separate queries for each year's sales data:

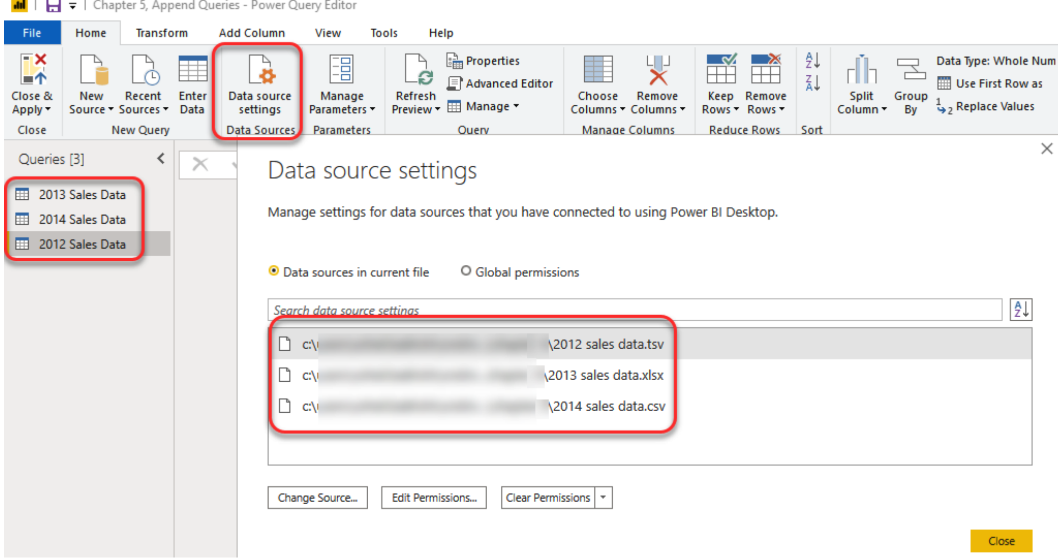


Figure 5.33 – Sales data spread across three separate queries coming from different data sources

We need to consolidate these queries into a single `Sales` query. Therefore, let's append the three queries into a new query named `Sales`:

1. Select the `2012 Sales Data` query from the **Queries** pane.
2. Click the **Append Queries** drop-down button from the **Home** tab of the ribbon.
3. Click the **Append Queries as New**.
4. Click **Three or more tables**.
5. Select `2013 Sales Data` and `2014 Sales Data` from the **Available tables** list.
6. Click the **Add>>** button.
7. Click **OK**:

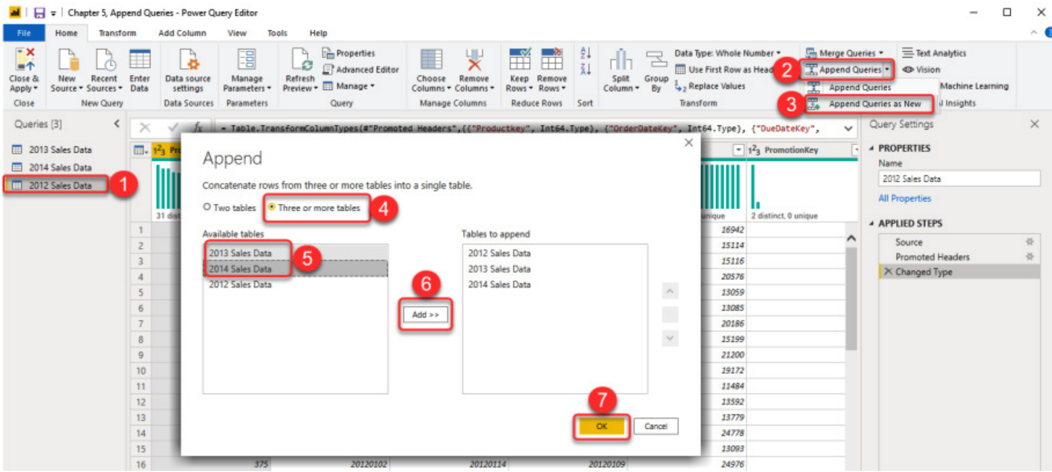


Figure 5.34 – Appending queries as a new query

The preceding steps create a new query named `Append1`.

1. Rename the query to `Sales`.
2. Unload all the queries that we appended by right-clicking on `2012 Sales D`.
3. Untick the **Enable load** option as shown in the following image:

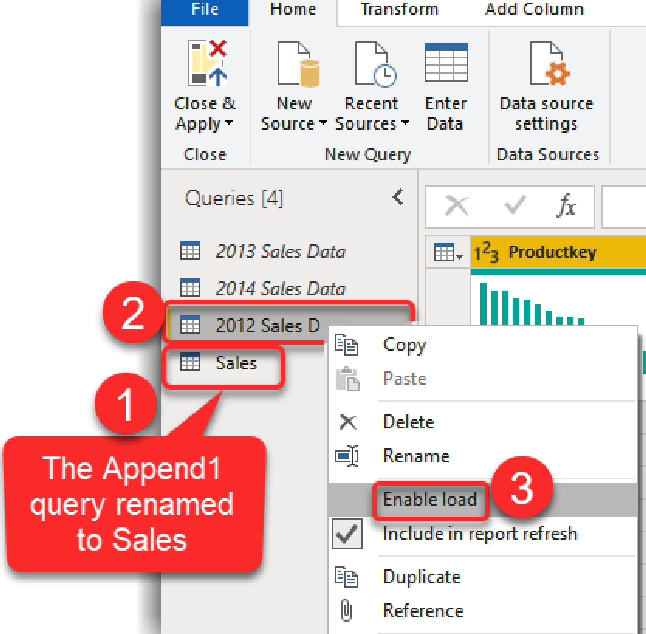


Figure 5.35 – Renaming the Append1 query and unloading the original queries

Looking at the results of the preceding steps raises an issue with the data. The following screenshot highlights this issue:

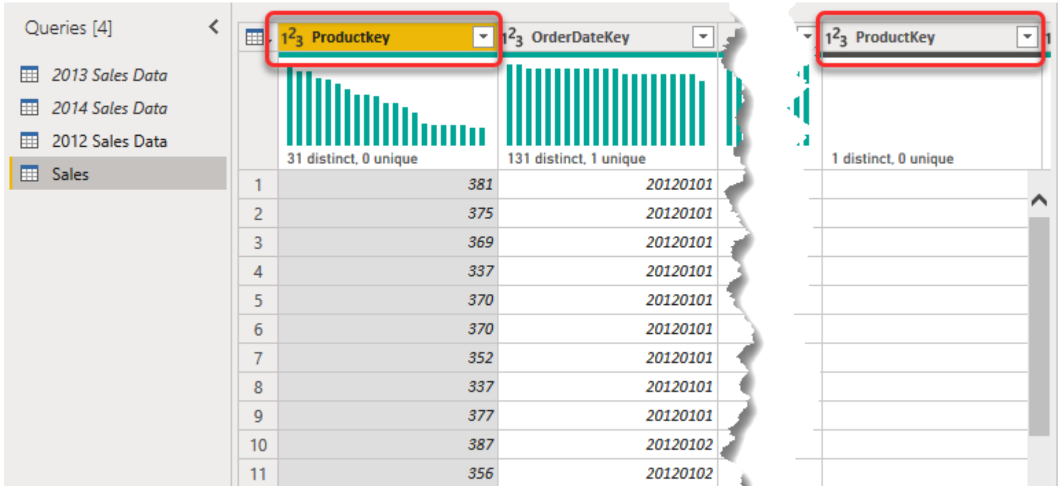


Figure 5.36 – There are two ProducyKey columns as a result of the case sensitivity of Power Query

As shown in the preceding screenshot, there are two `Product Key` columns – one called `ProductKey` and another called `Productkey`. This is a result of Power Query's case sensitivity. So, we need to rename one of those columns from the sourcing query. Looking at these queries reveals that the `Productkey` column comes from the `2012 Sales Data` query, while the other two queries contain `ProductKey`. We'll leave this issue for you to fix.

Merging queries

The Merge Queries functionality is one of the other common transformation operations we may use in Power Query. The merge queries functionality is useful when you want to denormalize snowflakes and absorb the data that's stored in different tables into one table. Power Query uses one of the following functions behind the scenes when we use Merge Queries from the Power Query Editor UI, depending on the matching type we select via the UI. This can be seen in the following screenshot:

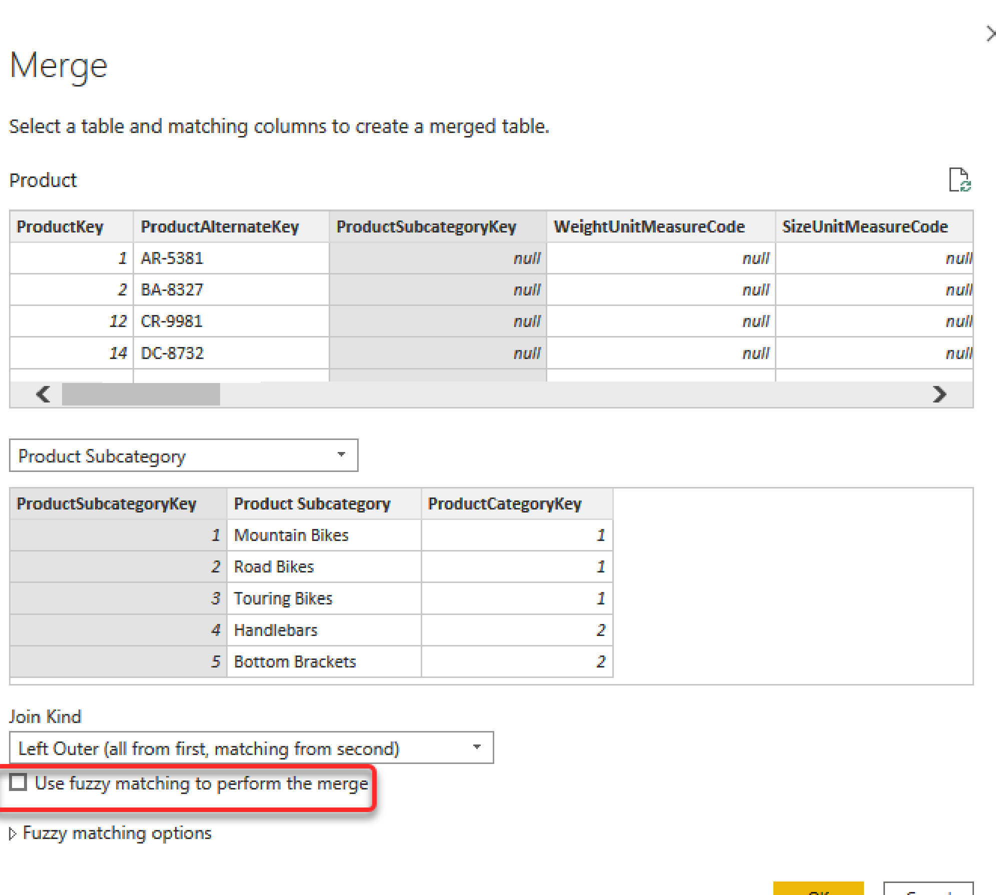


Figure 5.37 – Merging queries via the UI uses different Power Query functions, depending on the matching type

If we do not tick the **Use fuzzy matching to perform the merge** box, then the following function will be generated by the Power Query Editor:

```
Table.NestedJoin(FirstTable as table, KeyColumnofFirstTable
as any, SecondTable as any, KeyColumnofSecondTable as any,
NewColumnName as text, optional JoinKind as nullable JoinKind.
Type)
```

Otherwise, the Power Query Editor will generate this one:

```
Table.FuzzyNestedJoin(FirstTable as table,
KeyColumnofFirstTable as any, SecondTable as table,
KeyColumnofSecondTable as any, NewColumnName as text, optional
JoinKind as nullable JoinKind.Type, optional JoinOptions as
nullable record)
```

In both preceding functions, the join kind is optional, so if it's not specified, **Left Outer** is used. We can choose to either use numeric enumerations to specify the join kind or explicitly mention the join kind. The following table shows the join kinds and their respective enumerations:

Join Kind	Enumeration
JoinKind.Inner	0
JoinKind.LeftOuter	1
JoinKind.RightOuter	2
JoinKind.FullOuter	3
JoinKind.LeftAnti	4
JoinKind.RightAnti	5

Table 5.2 – Power Query Join Kinds in Merge Queries

A difference between the `Table.NestedJoin()` function and the `Table.FuzzyNestedJoin()` function is that the `Table.NestedJoin()` function uses the equality of the key columns' values, while the `Table.FuzzyNestedJoin()` function uses text similarities on the key columns.

Important Notes for Merging Two Queries

Merge Queries allows you to have composite key columns in the merging tables; therefore, we can select multiple columns while selecting the key columns of the first table and the second table.

There are six different join types. Therefore, we need to understand how the join types are different and which join types suite our purpose.

In this section, we will focus on the join kinds and how they are different:

- `Inner`: Joins two queries based on the matching values of the key columns from both tables participating in the join operation.
- `LeftOuter`: Joins two queries based on all values of the key columns from the first table and matches the values of the key columns from the second table.
- `RightOuter`: Joins two queries based on all the values of the key columns from the second table and matches the values of the key columns from the first table.
- `FullOuter`: Joins two queries based on all the values of the key columns from both tables participating in the join operation.
- `LeftAnti`: Joins two queries based on all the values of the key columns from the first table that do not have any matching values in the key columns from the second table.
- `RightAnti`: Joins two queries based on all the values of the key columns from the second table that do not have any matching values in the key columns from the first table.

Let's use a graphical representation of different joins to understand this:

Figure 5.38 – Different join kinds

We already went through a scenario in *Chapter 1*, *Introduction to Data Modeling in Power BI*, in the *Star Schema (dimensional modeling), snowflaking* section where we denormalized the `Product Subcategory` and `Product Category` tables into the `Product` table. Therefore, we will not explain this again. For your reference, we used the `Adventure Works, Internet Sales.pbix` sample file.

Duplicating and referencing queries

Duplicating and referencing a query are somehow similar. We duplicate a query when we need to have all the transformation steps we already took on the original query. At the same time, we want to change those steps or add some more transformation steps. In that case, we must change the original query's nature, translating it so that it has a different meaning from a business point of view. But when we reference a query, we are referencing the final results of the query. Therefore, we do not get the transformation steps in the new query (the referencing query). Referencing a query is a common way to break down the transformation activities in a more organized way. This is the preferred way of doing data preparation for most **Extract, Transformation, and Load** (**ETL**) experts and data warehousing professionals. In that sense, we can do the following:

- We can have base queries that are connected to the source system that resemble the **Extract** part of the ETL process.
- We can reference the base queries and go through the **Transformation** steps of the ETL process.
- Finally, we can reference the transformation queries to prepare our Star Schema. This is the **Load** part of the ETL process.

In the preceding approach, we unload all the queries for the first two points as they are our transformation steps, so we only enable data load for the queries of the Load part (the last point). Duplicating and referencing a query is simple; right-click a desired query from the **Queries** pane and click either **Duplicate** or **Reference**, as shown in the following screenshot:

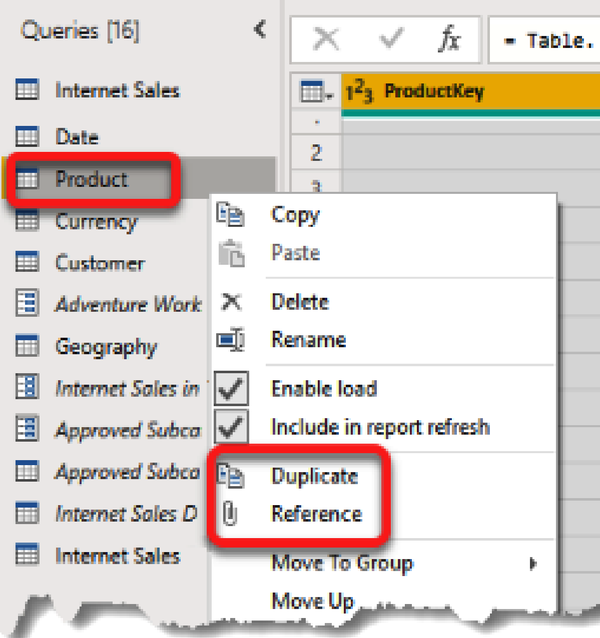


Figure 5.39 – Referencing or duplicating a query

Notes

When we reference a query, we have to be vigilant about making changes to the base query. Any changes to the base query may break the referencing query or queries.

When the base query is loaded into the data model, if we make changes to the base query and load the data into the data model, the base query and all the queries referencing it will be refreshed. However, this is not the case when we duplicate queries. The duplicated query is independent of its base query.

So far, we've looked at the most common table manipulations in Power Query. In the next section, we'll look at common text manipulations.

Replacing values

A critical part of data preparation is data cleansing. When it comes to data cleansing, replacing values is one of the most common transformation activities we perform. A simple example is where we have a description column in which the users of the source system enter free text data, and we would like to replace some part of the description with something else.

When we use the Power Query Editor UI to replace a value in a table, it uses the `Table.ReplaceValue(table as table, OldValue as any, NewValue as any, Replacer as function, columnsToSearch as list)` function behind the scenes. If we want to replace the value of a `List`, it uses the `List.ReplaceValue(list as list, OldValue as any, NewValue as any, Replacer as function)` function. Depending on the value's data type that we want to replace, the `Replacer` function can be either `Replacer.ReplaceText` or `Replacer.ReplaceValue`. The difference between the two is that we can use `Replacer.ReplaceText` to replace text values, while we can use the `Replacer.ReplaceValue` to replace any values. Replacing a value from the UI is relatively simple, as follows:

1. Click a column that we want its values to be replaced.
2. Click **Replace Values** from the context menu.
3. If the selected column's type is text, then more options are available to us under **Advanced options**, such as **Replace using special characters**. In our example, we want to replace semicolons with colons.
4. Type `;` into the **Value To Find** text box.
5. Type in `,` into the **Replace With** text box.
6. Click **OK**:

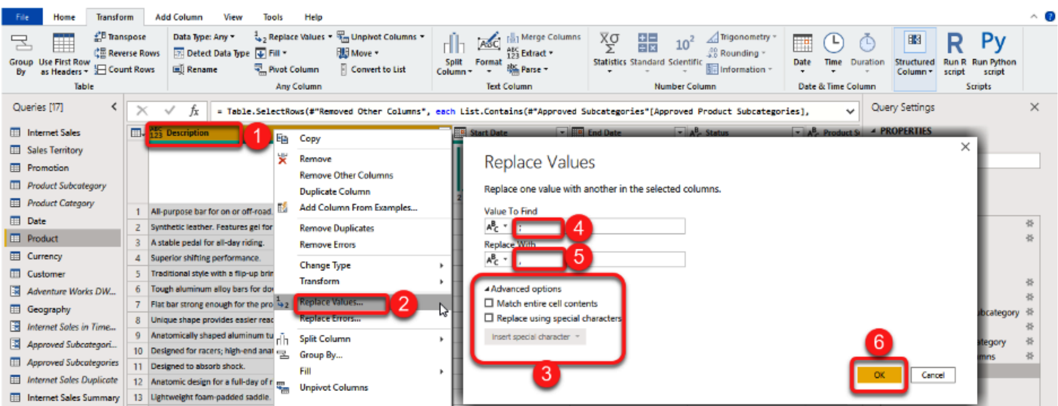


Figure 5.40 – Replace Values popup

However, in most real-world cases, we face more challenging scenarios. Let's look at some.

The business requires us to show product descriptions in a table visualization. There are some long descriptions. The business wants to cut off excessive text in the `Description` column from the `Product` table when the description's length is greater than 30, and then show "…" at the end of the description. These three dots indicate that the text values in the `Description` columns have been cut. The following steps show how to use the Power Query Editor UI to generate the preliminary expression and how we change it to achieve our goal:

1. Select the `Description` column.
2. Click the **Replace Values** button from the **Transform** tab of the ribbon.
3. Type in dummy values in both the **Value To Find** and **Replace With** text boxes.
4. Click **OK**:

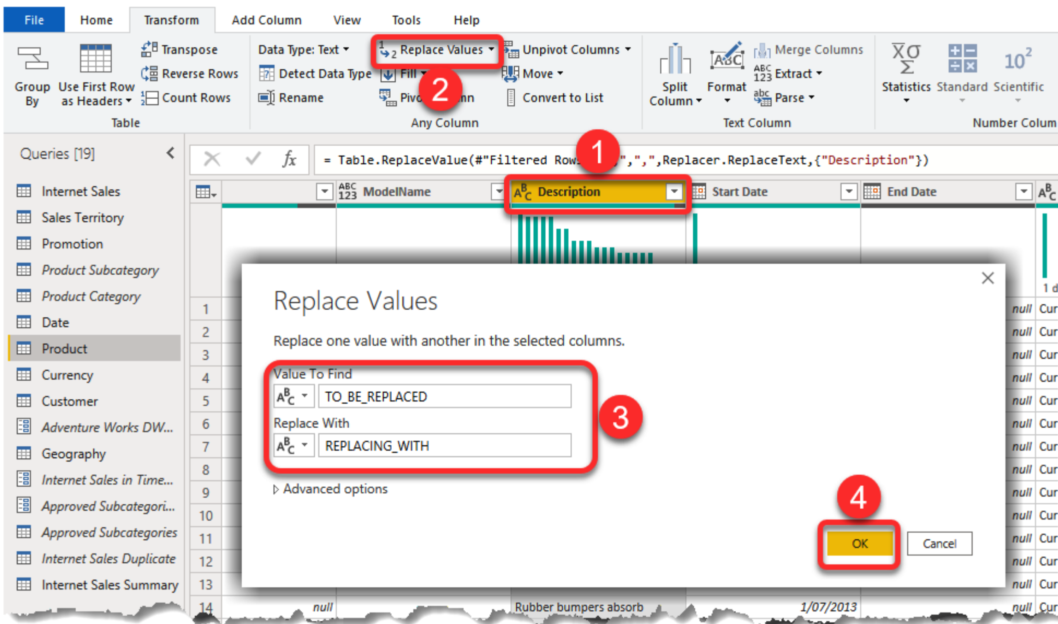


Figure 5.41 – Replacing a dummy value

5. In the generated expression, replace the `TO_BE_REPLACED` string with the following expression:

```
each if Text.Length([Description]) > 30 then [Description] else
""
```

6. Replace `REPLACING_WITH` with the following expression:

```
each Text.Start([Description], 30) & "..."
```

7. Click the **Submit** (✓) button.

The overall expression must look like this:

```
= Table.ReplaceValue(#"Replaced Value", each if Text.
Length([Description]) > 30 then [Description] else "",
each Text.Start([Description], 30) & "...", Replacer.
ReplaceText,{"Description"})
```

The preceding code block will return the following output:

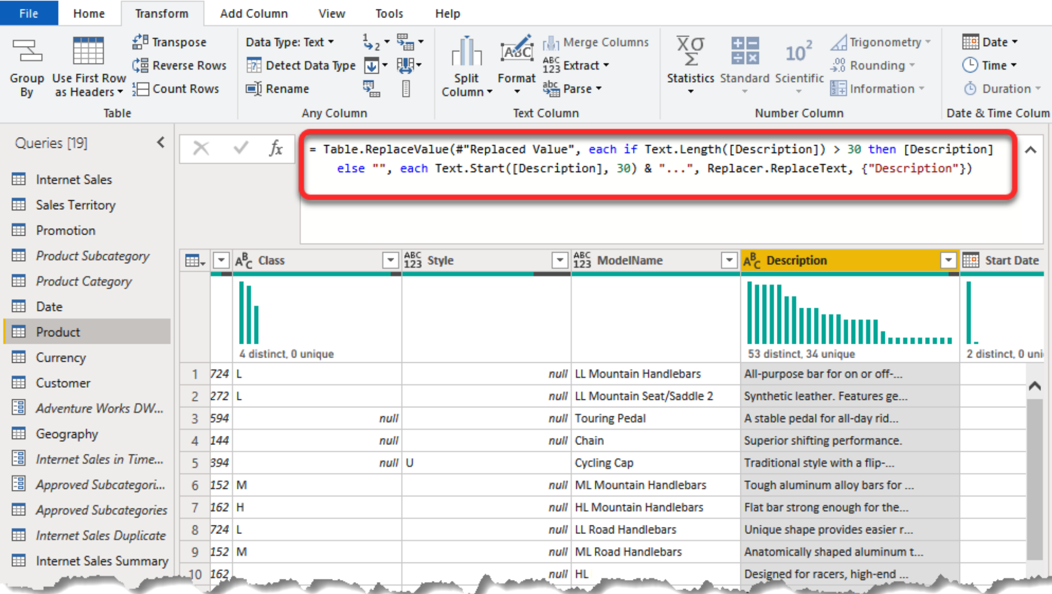


Figure 5.42 – Cutting off the values in the Description column that are longer than 30 characters

We may need to replace values based on values from another column or based on values from another query in some other scenarios. Replacing values is a widespread transformation step, and there are many scenarios that are virtually impossible to cover in this section.

Extracting numbers from text

Another common data preparation step is when we need to extract a number from text values. An excellent example is when we want to extract a flat number or a zip code from an address. Other examples include extracting the numeric part of a sales order number or cleaning fullnames of typos, such as when some names contain numbers. In our scenario, we want to add two new columns to the `Customer` table, as follows:

- Extract `Flat Number` as a new column from `AddressLine1`
- Extract the rest of the address, `Street Name`, as a new column

The `AddressLine1` column reveals that the flat number appears in different parts of the address; therefore, splitting by transitioning from digit to non-digit would not work:

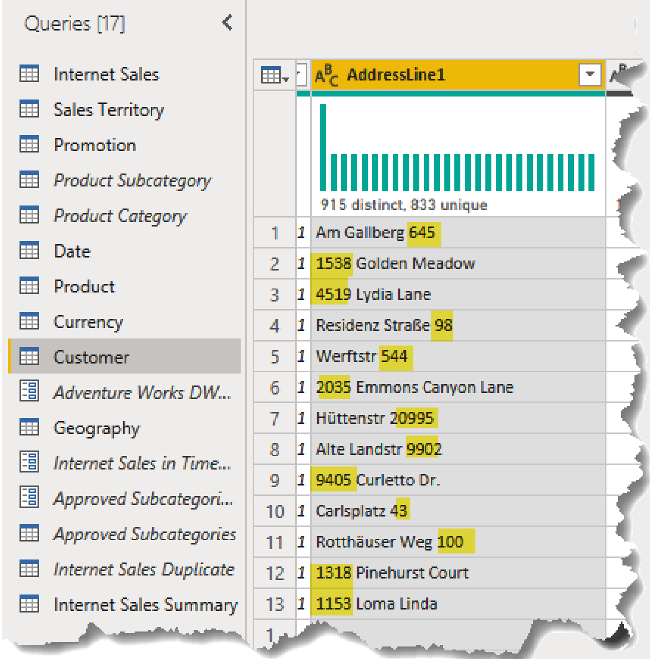


Figure 5.43 – Flat Number appears in different places in AddressLine1

To achieve our goal, we need to extract the numbers from text. To do so, we can use the `Text.Select(Text as nullable text, SelectChars as any)` function to get the job done. Follow these steps:

1. Click the **Custom Column** button from the **Add Column** tab of the ribbon.
2. Type in `Flat Number` as **New column name**.

3. Type in the following expression:

```
Text.Select([AddressLine1], {"0".."9"})
```

4. Click **OK**.

The following screenshot shows the preceding steps:

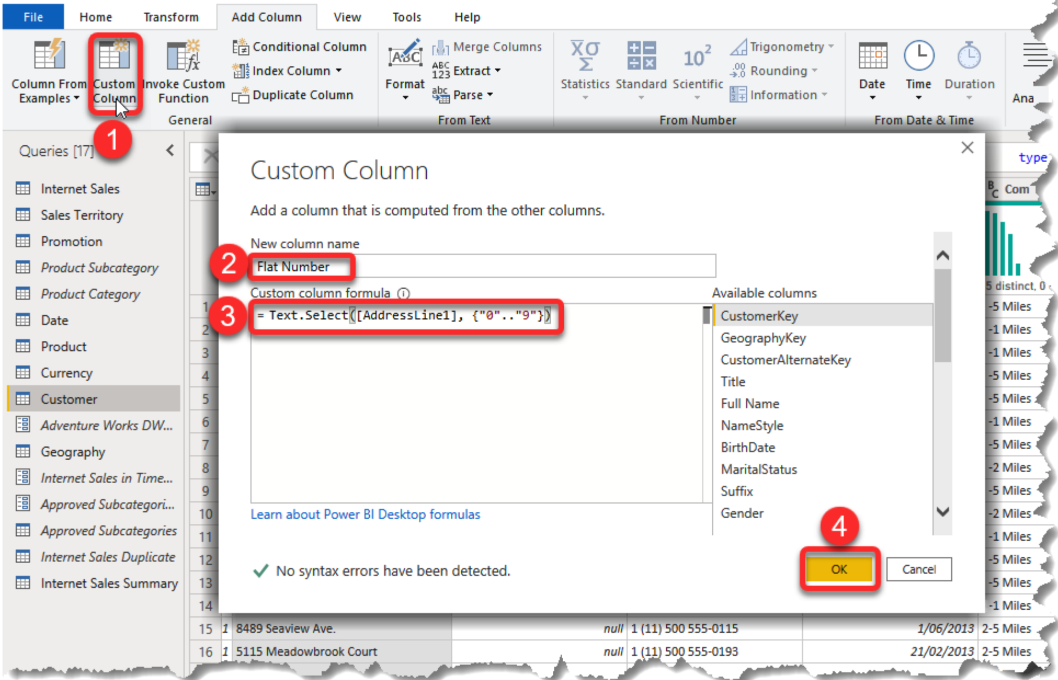


Figure 5.44 – Extracting Flat Number from Address as a new column

Now, we will use the same function with a different character list to extract the street name from `AddressLine1`, as follows:

1. Click the **Custom Column** button again to add a new column.
2. Type in `Street Name` as **New column name**.
3. Type in the following expression:

```
Text.Select([AddressLine1], {"a".."z", "A".."Z", " ", "."})
```

4. Click **OK**.

The preceding expression keeps all the letters, capital letters, the space character, and the dot character but drops anything else. The following screenshot shows the results:

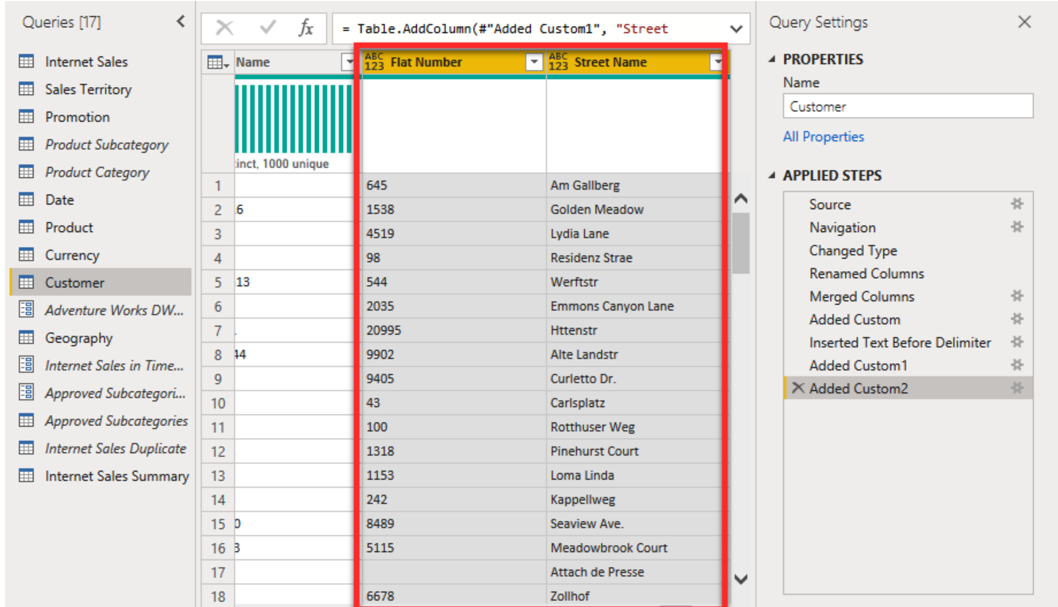


Figure 5.45 – The Flat Number and Street Name columns added to the Customer table

Dealing with Date, DateTime, and DateTimeZone

Generating `date`, `datetime`, and `datetimezone` values in Power Query is simple. We just need to use the three functions.

To generate `date` values, we can use the following command:

```
#date(year as number, month as number, day as number)
```

To generate `datetime` values, we can use the following command:

```
#datetime(year as number, month as number, day as number, hour
as number, minute as number, second as number)
```

To generate `datetimezone` values, we can use the following command:

```
#datetimezone(year as number, month as number, day as number,
hour as number, minute as number, second as number, offsetHours
as number, offsetMinutes as number)
```

The following code generates a record of the `Date`, `DateTime`, and `DateTimeZone` values:

```
let
    Source = [Date = #date(2020, 8, 9)
            , DateTime = #datetime(2020, 8, 9, 17, 0, 0)
            , DateTimeZone = #datetimezone(2020, 8, 9, 17, 0,
0, 12, 0)]
in
    Source
```

The results of the preceding code can be seen in the following screenshot:

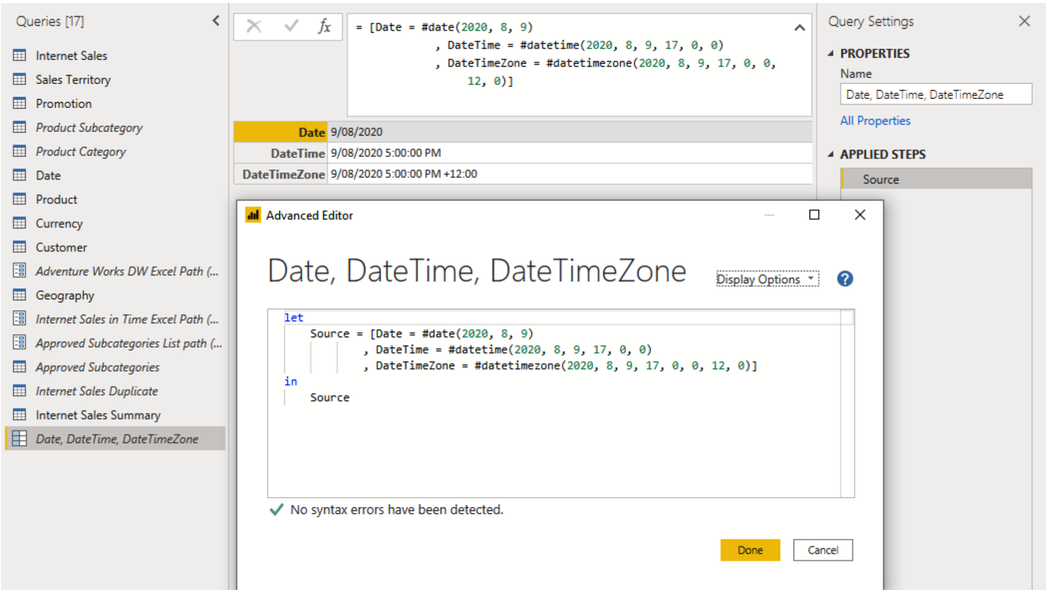


Figure 5.46 – Generating Date, DateTime, and DateTimeZone values

In most cases, the issue we're dealing with is not this simple. A common use case is when we have **Smart Date Keys**, and we want to generate the corresponding dates or vice versa.

> **Note**
>
> A **Smart Date Key** is an integer representation of a date value. Using a Smart Date Key is very common in data warehousing for saving storage and memory. So, the `20200809` integer value represents the `2020/08/09` date value. Therefore, if our source data is coming from a data warehouse, we will likely have Smart Date Keys in our tables.

In this scenario, we have the date values in the `Internet Sales` table, and we want to get the Smart Date Key of the `OrderDate` column. We want to add a new column that contains the corresponding integer values of the date values in the `OrderDate` column:

1. In the `Internet Sales` table, add a new column.
2. Name it `OrderDateKey`.
3. Use the following expression as our **Custom Column Formula**:

```
Int64.From(Date.ToText([OrderDate], "yyyyMMdd"))
```

4. Click **OK**.
5. Add `Int64.Type` as the optional operand of the `Table.AddColumn()` function from the expression bar.

The following screenshot shows the preceding steps:

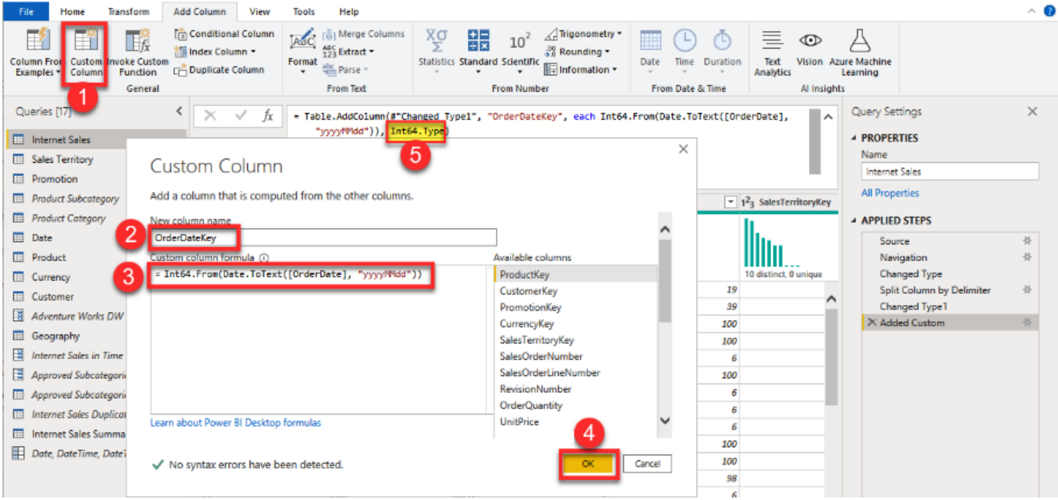


Figure 5.47 – Adding OrderDateKey (Smart Date Key) as a new column

Another scenario that can happen from time to time is when we need to represent the `Date` values in a different time zone. Let's go through this with a scenario.

The `Date` values in the `OrderDateTime` column in the `Internet Sales Date Time` table are stored in New Zealand local date-time. The business requires that we add a column to show `OrderDateTime` in **Universal Time Coordinate** (**UTC**).

To achieve this, we need to add a new custom column by using the `DateTimeZone.ToUtc(DateTimeZone as nullable datetimezone)` function, which must be nested into the `Table.AddColumn` function, as shown in the following expression:

```
Table.AddColumn(#"Changed Type", "OrderDateTimeUTC", each
DateTimeZone.ToUtc(DateTimeZone.From([OrderDateTime])),
DateTimeZone.Type)
```

The following screenshot shows the results of adding the new custom column:

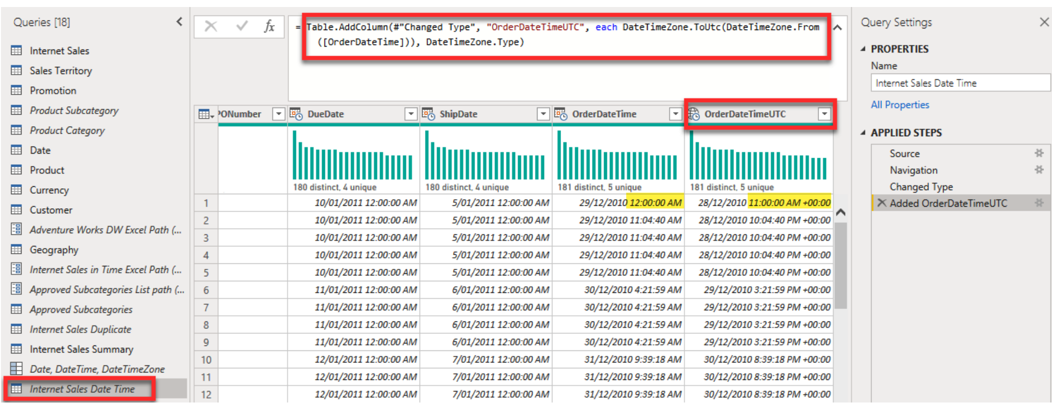


Figure 5.48 – OrderDateTimeUTC custom column added

Note the highlighted values in the preceding screenshot. The difference between the two columns is 13 hours. When we look closer at the data, we can see that **OrderDateTimeUTC** considered the daylight saving dates of my local machine. While I live in New Zealand, my local time difference between UTC time can be either 12 hours or 13 hours, depending on the date. If we use the `DateTimeZone.ToUtc()` function while not specifying the time zone, the value converts into the `DateTimeZone` data type and then `DateTimeZone.ToUtc()` turns the values into UTC.

What if we have the values of the `OrderDateTime` column stored in UTC, but the business needs to see the data in our local date-time? We can use the `DateTimeZone.ToLocal(dateTimeZone as nullable datetimezone)` function for this. So, if we need to add a new column to show the UTC values in local time, then it should look like the following expression:

```
Table.AddColumn(#"Added OrderDateTimeUTC",
"OrderDateTimeLocal", each DateTimeZone.
ToLocal([OrderDateTimeUTC]), type datetimezone)
```

The results are shown in the following screenshot:

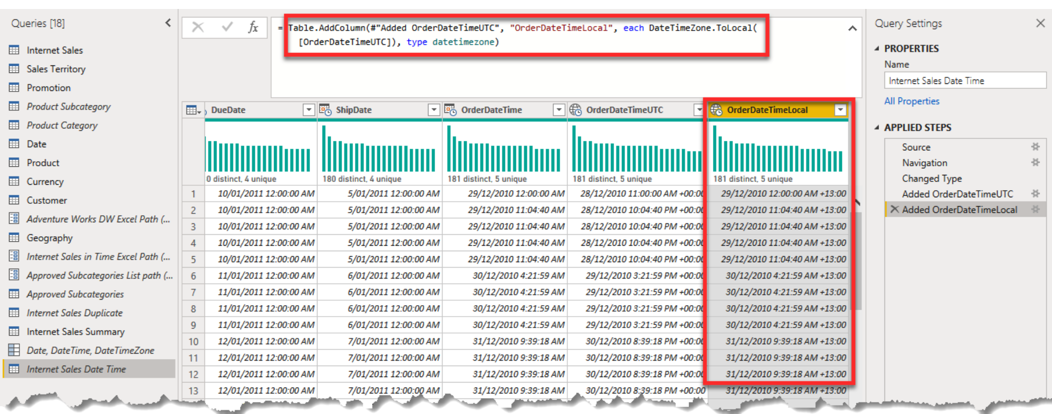


Figure 5.49 – Adding a new custom column to show OrderDateTime values in local time from UTC values

Summary

This chapter looked at common data preparation steps, which means we now know how data type conversion works and what can go wrong during this. We learned how to split a column, merge columns, add a custom column, and filter rows. We also learned how to use the Group By functionality in queries to create summarized tables. We also learned how to append queries and merge queries. Finally, we dealt with scenarios related to `Date`, `DateTime`, and `DateTimeZone`.

These skills give us firm ground to move on to the next chapter. In the next chapter, we will learn to prepare a Star Schema in Power Query Editor.

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:

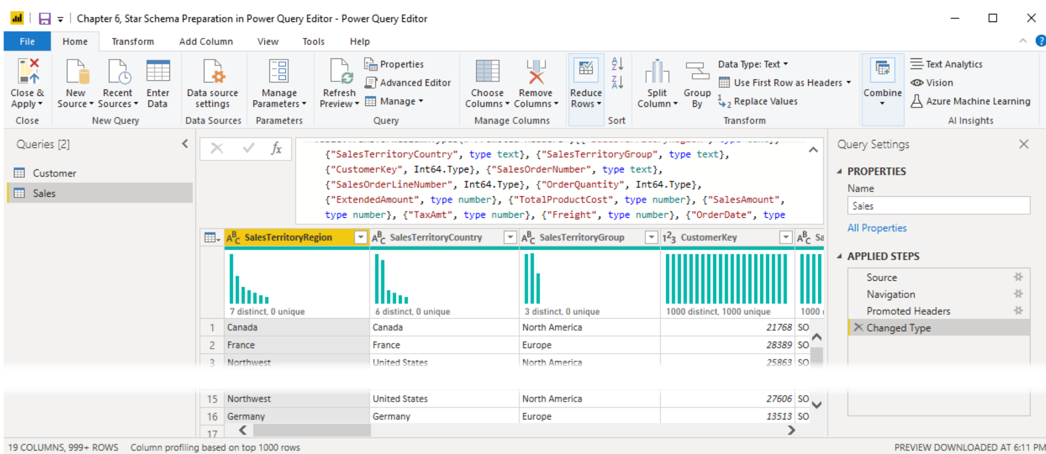


Figure 6.1 – Power Query Editor connected to the Chapter 6, Sales Data.xlsx sample data

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:

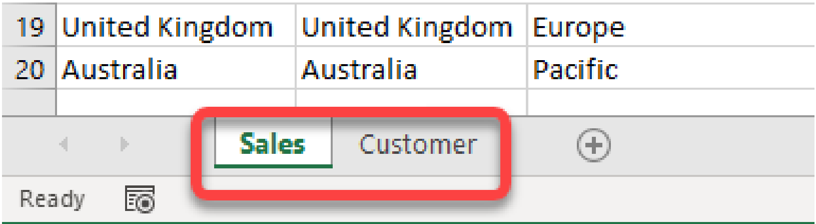


Figure 6.2 – The Excel sample file contains two sheets that translate into two base tables in Power Query Editor

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`:

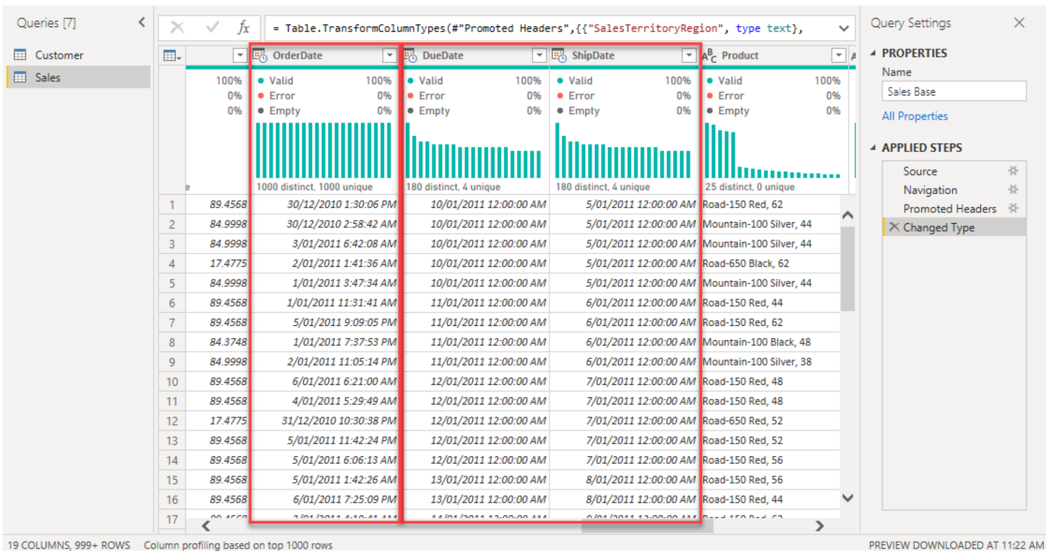


Figure 6.3 – Columns with the Date and DateTime data types in the Sales table

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:

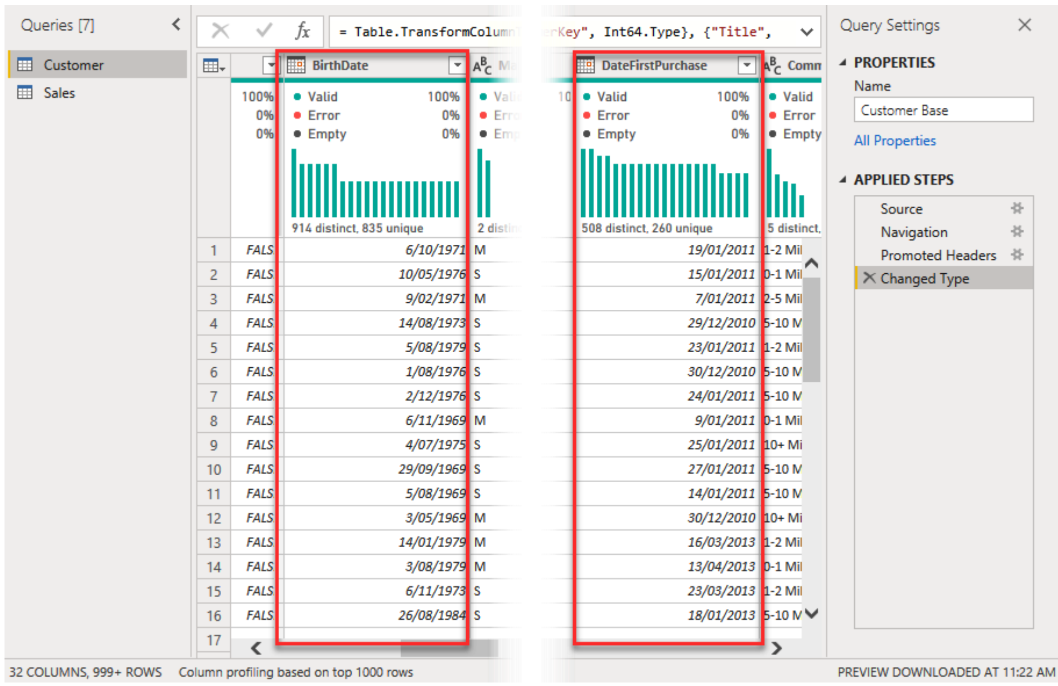


Figure 6.4 – The Date column in the Customer table

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:

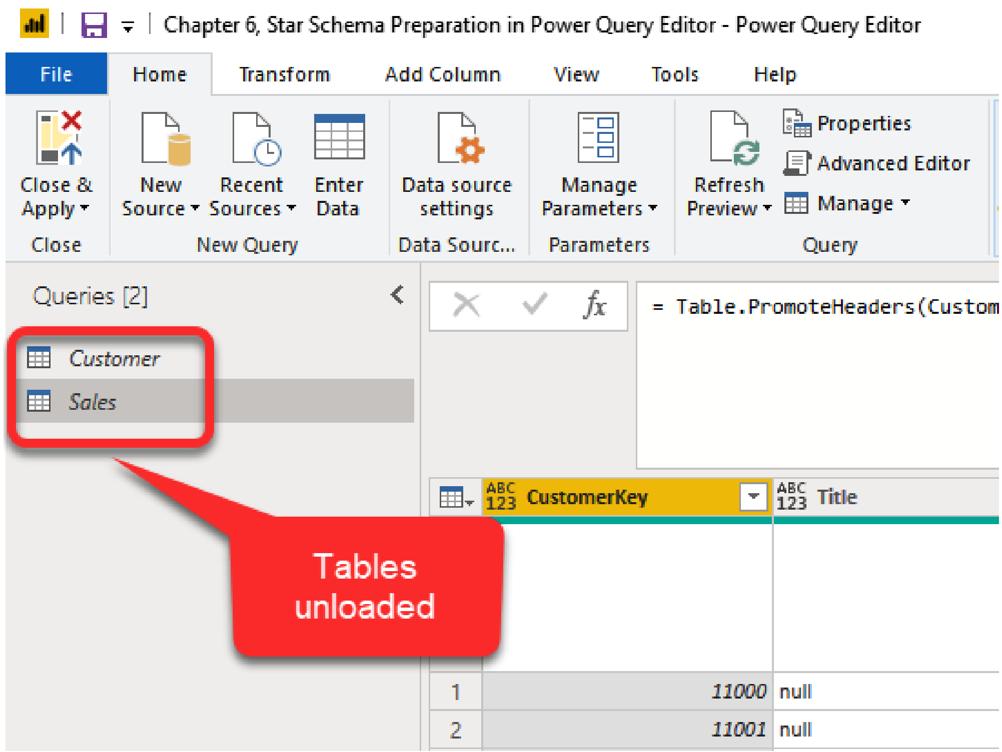


Figure 6.7 – Both the Sales and Customer tables unloaded

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

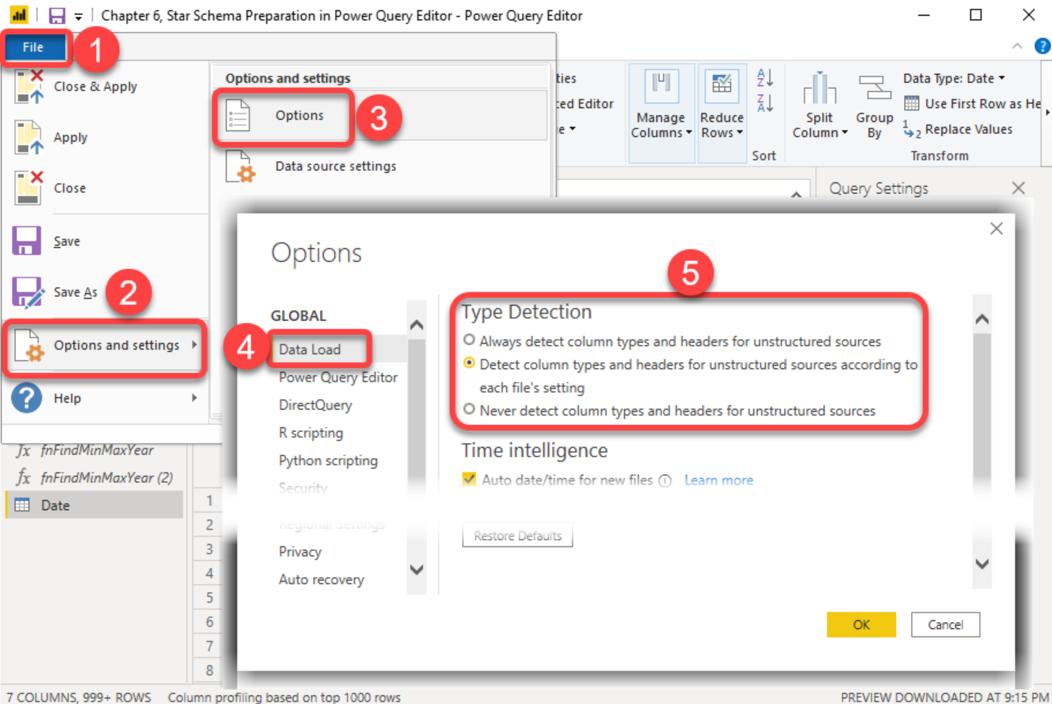


Figure 6.8 – Changing the Type Detection configuration at the Global level

6. Click **Data Load** under the **Current File** section.

7. Tick/untick the **Detect column types and headers for unstructured sources** option:

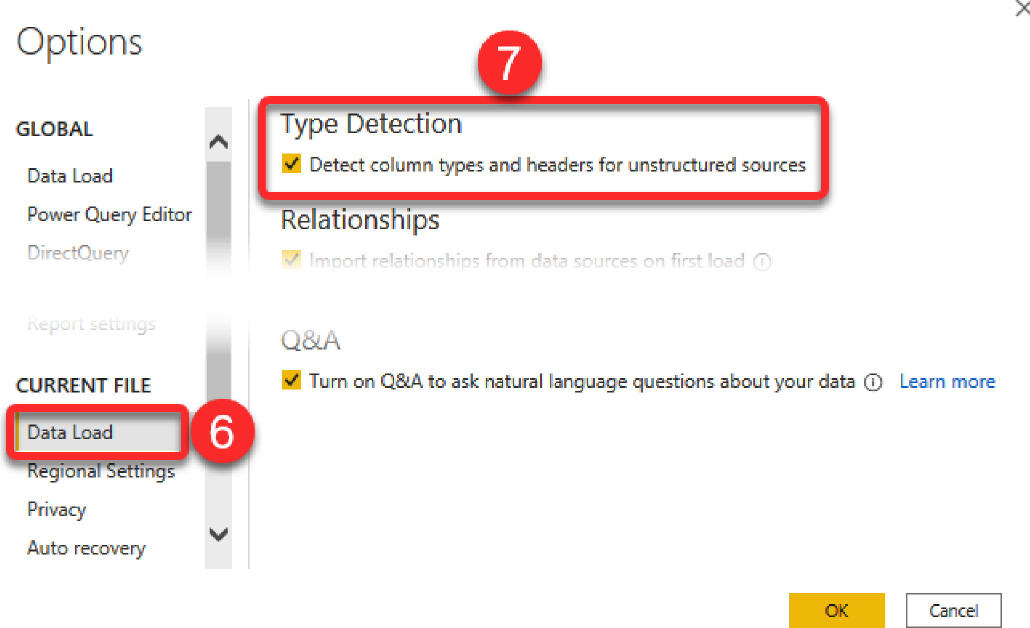


Figure 6.9 – Changing the Type Detection configuration at the Current File level

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:

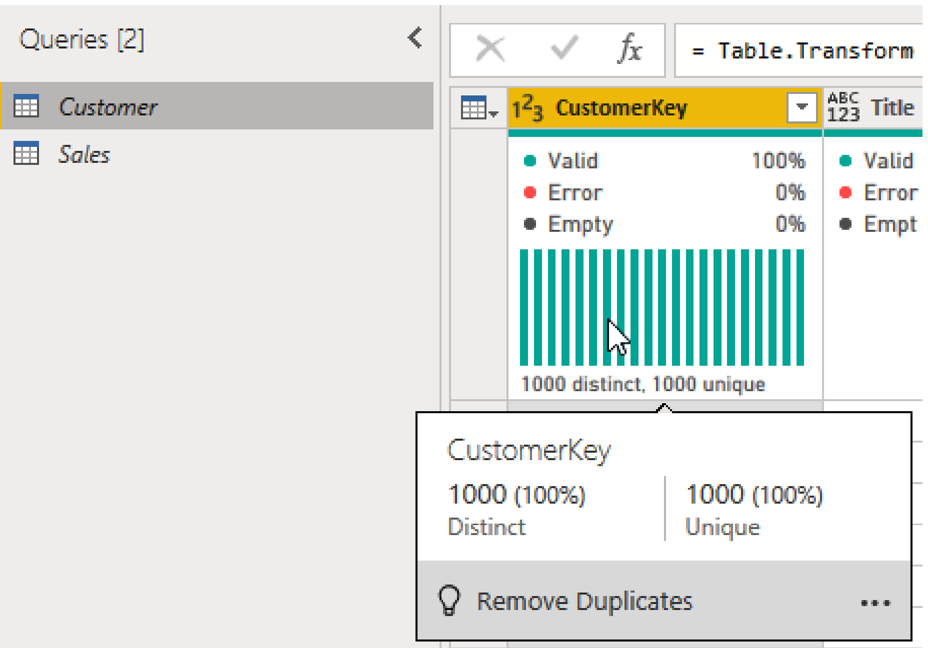


Figure 6.10 – Column Distribution shows that CustomerKey is the primary key for the Customer table

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:

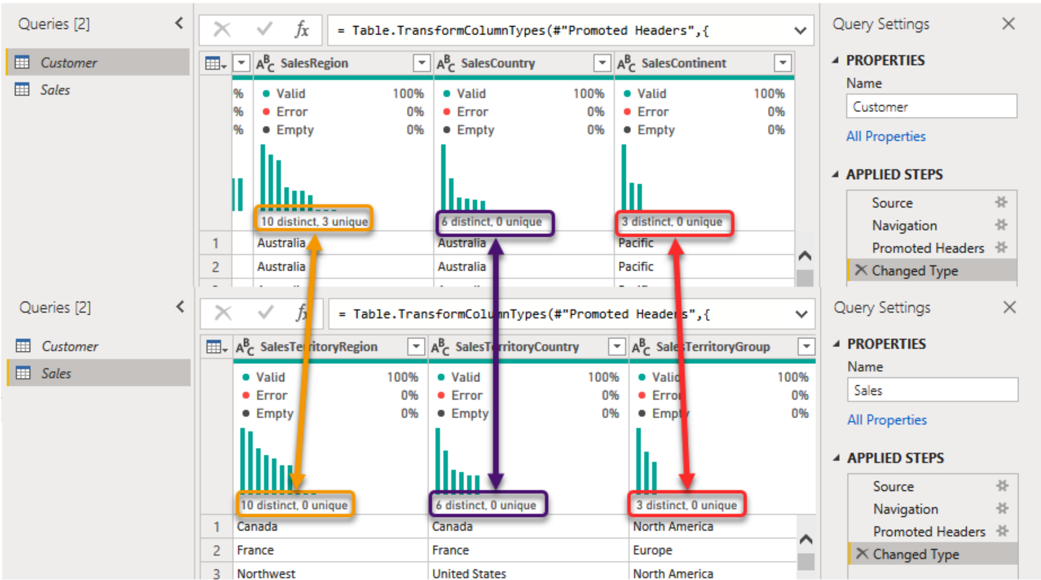


Figure 6.11 – Comparing Column Distribution for geography-related columns from the Customer table and the Sales table

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:

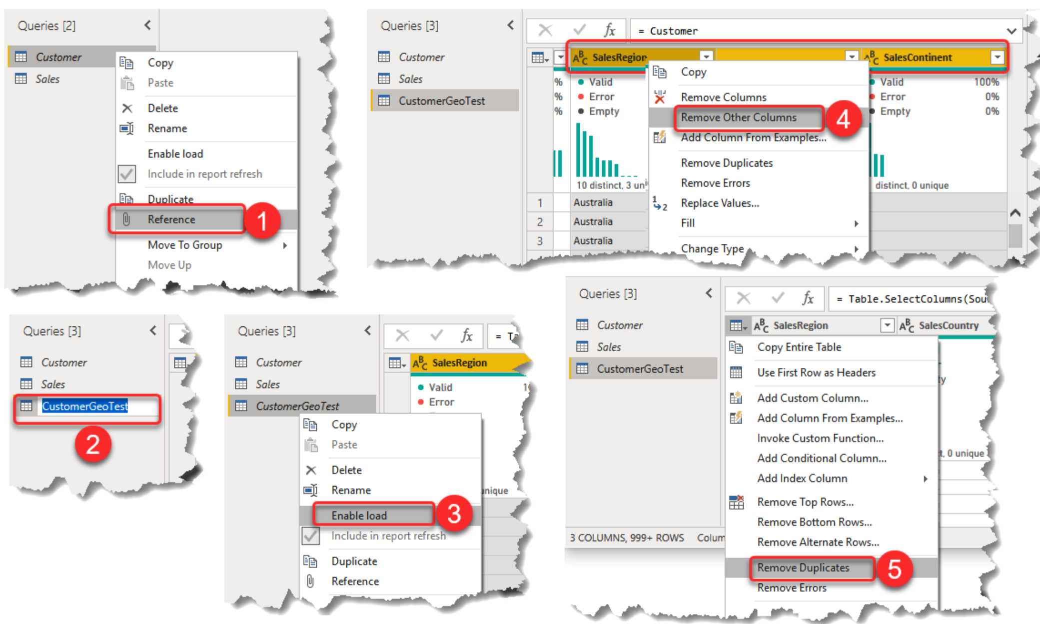


Figure 6.12 – Steps required to reference the Customer table and remove duplicates

The results of the preceding steps are as follows:

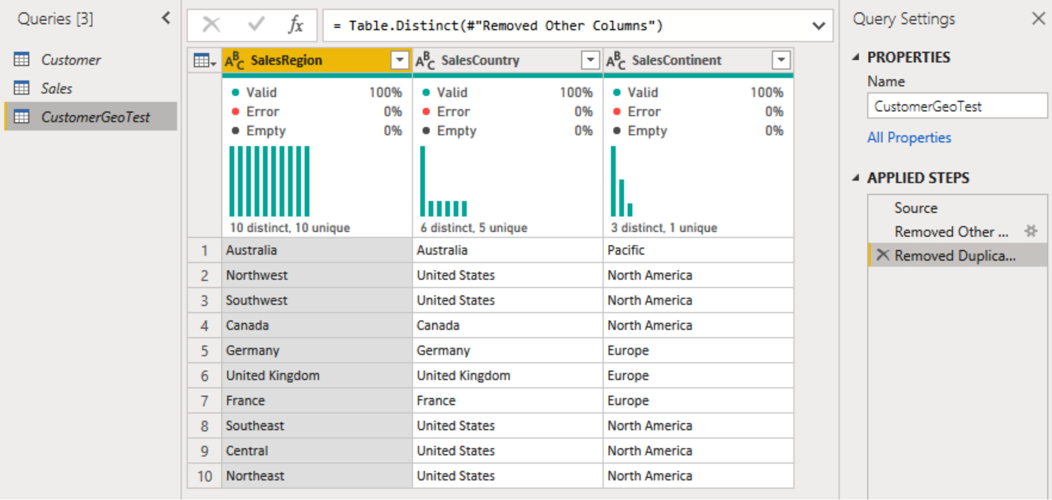


Figure 6.13 – The results of referencing the Customer table and removing duplicates from the geography-related columns

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:

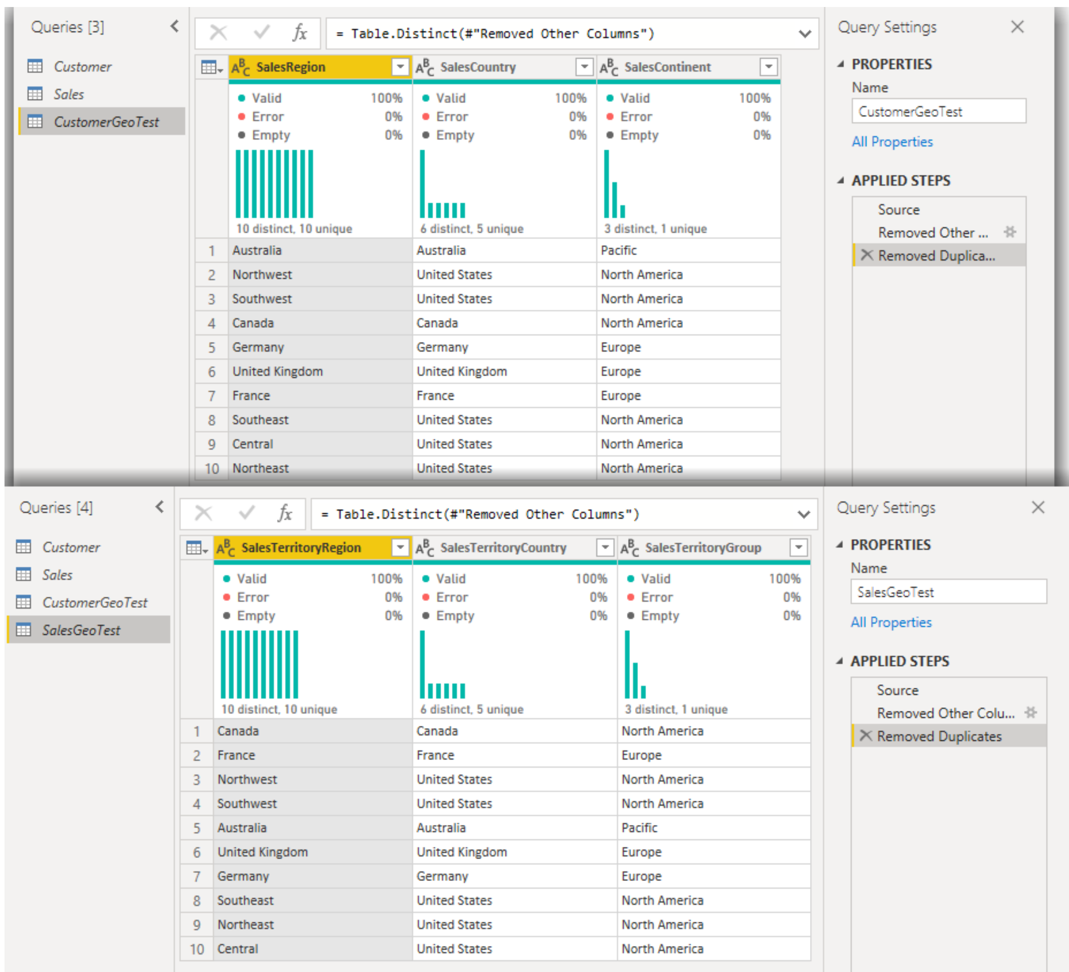


Figure 6.14 – Comparing the results of the CustomerGeoTest table and the SalesGeoTest table

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:

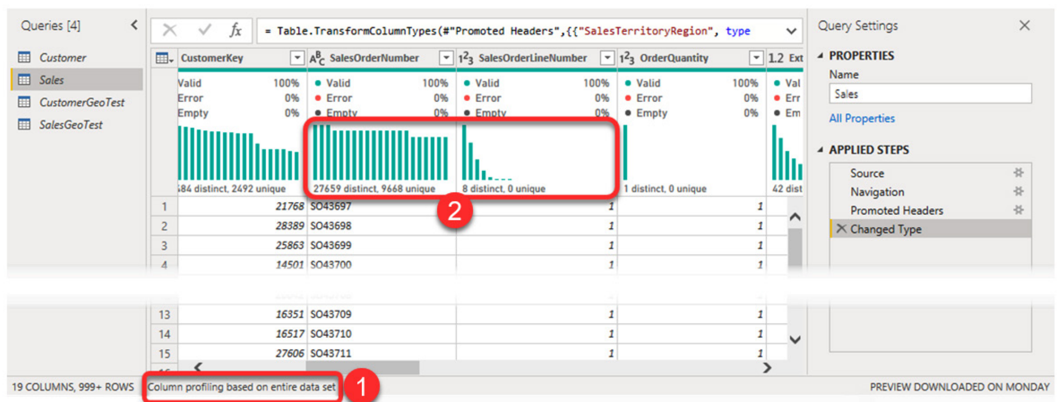


Figure 6.15 – Looking at the Column Distribution data shows that there are a lot of distinct values in SalesOrderNumber

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**:

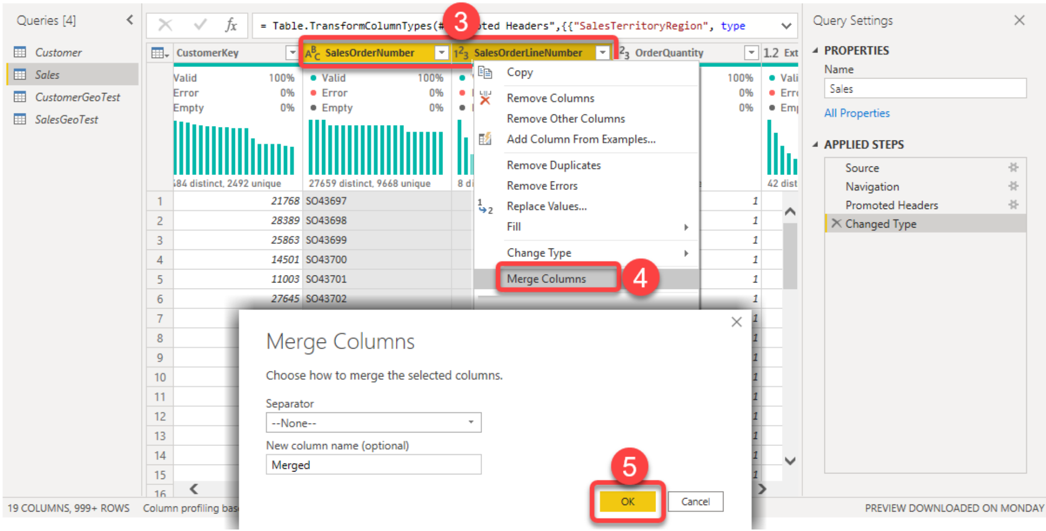


Figure 6.16 – Merging the SalesOrderNumber and SalesOrderLineNumber columns

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**:

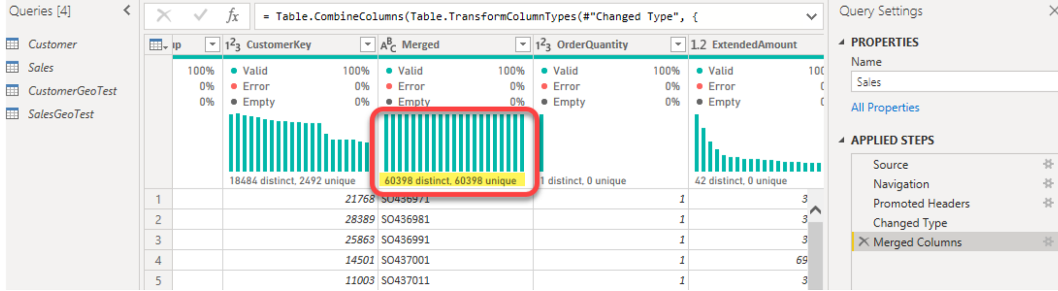


Figure 6.17 – Merged Column results

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`:

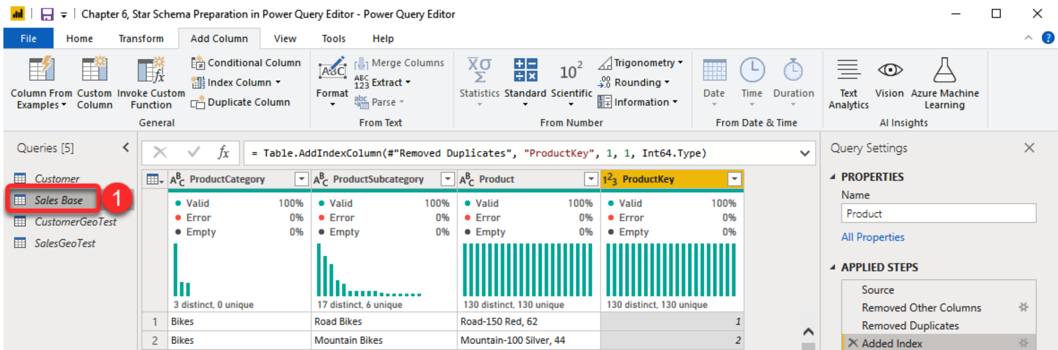


Figure 6.18 – Renaming the Sales table Sales Base

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

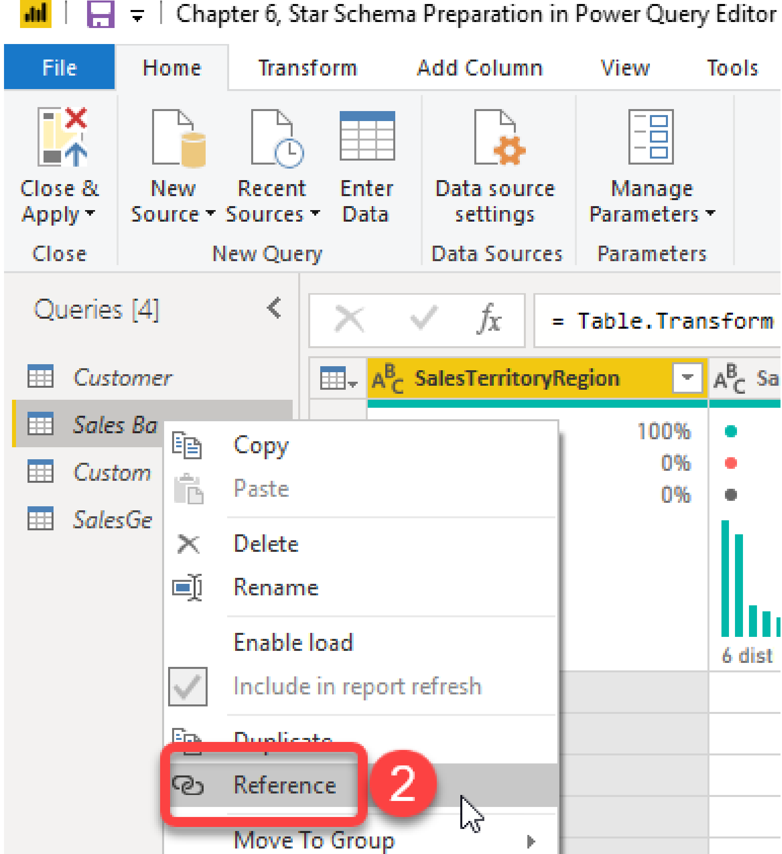


Figure 6.19 – Referencing the Sales Base table

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**:

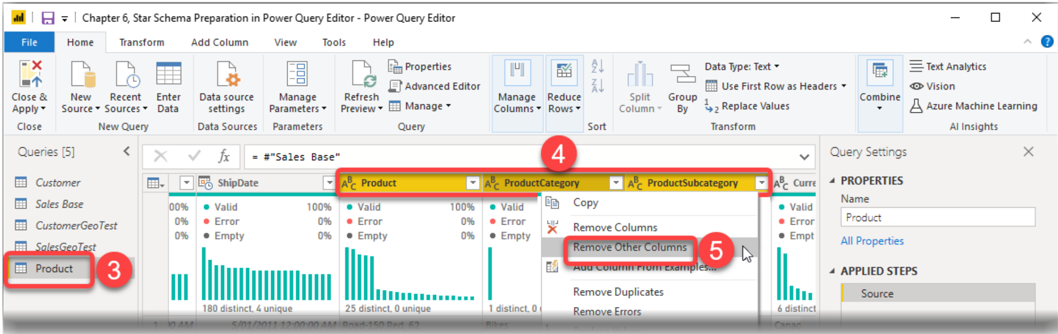


Figure 6.20 – Remove Other Columns option

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

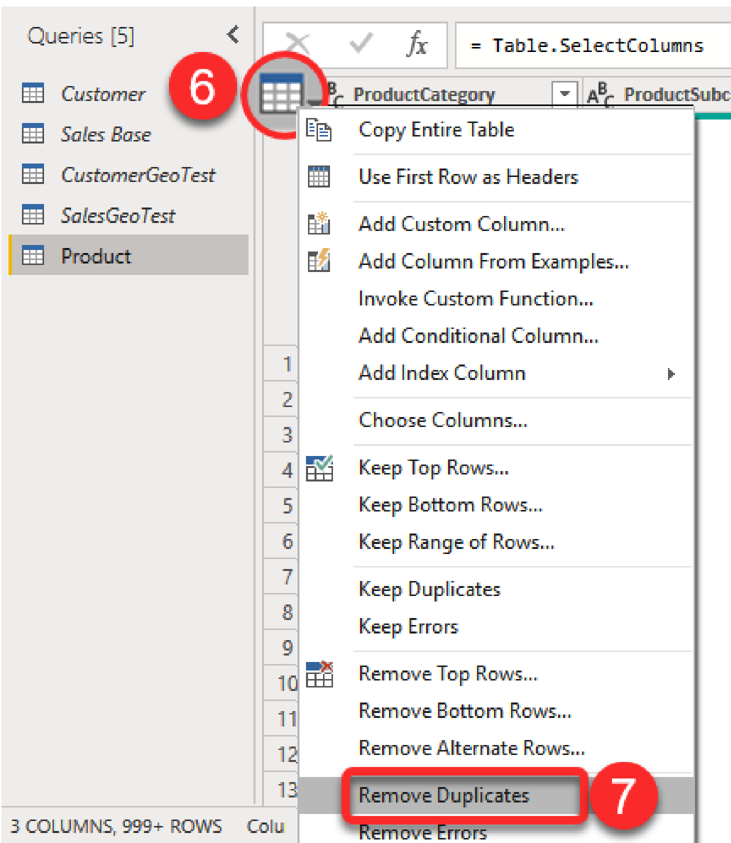


Figure 6.21 – Removing duplicates from all rows

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**:

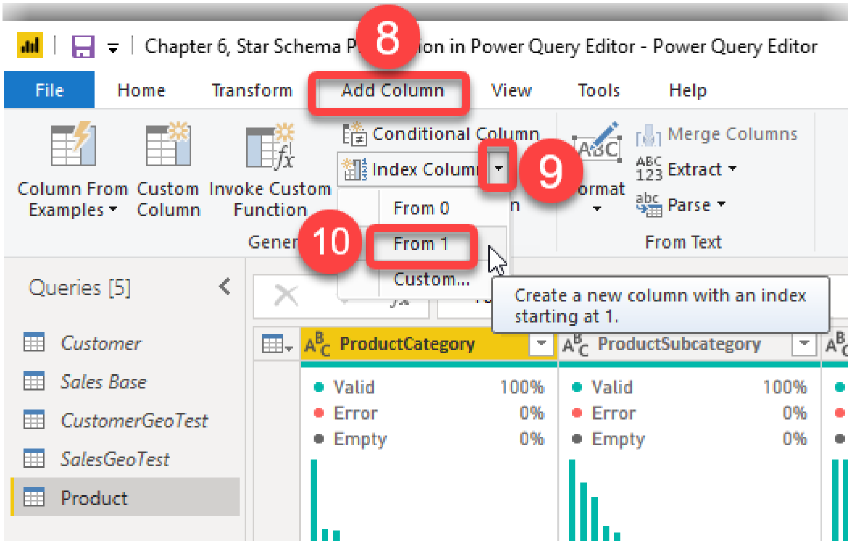


Figure 6.22 – Adding an Index column with an index starting from 1

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:

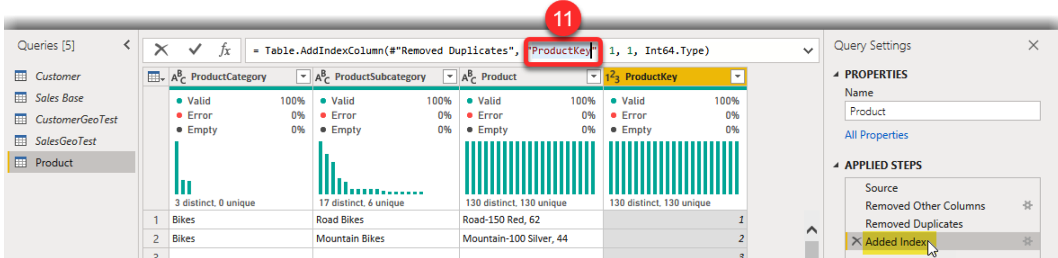


Figure 6.23 – Changing the default index column's name

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:

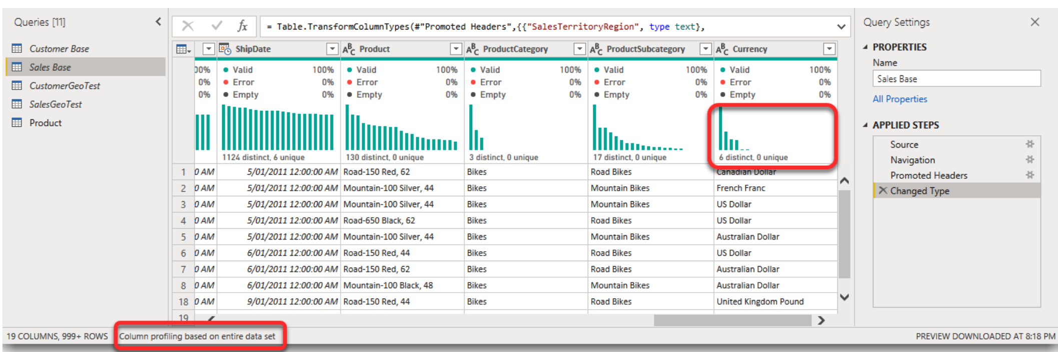


Figure 6.24 – Column distribution for the Currency column over the entire dataset

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:

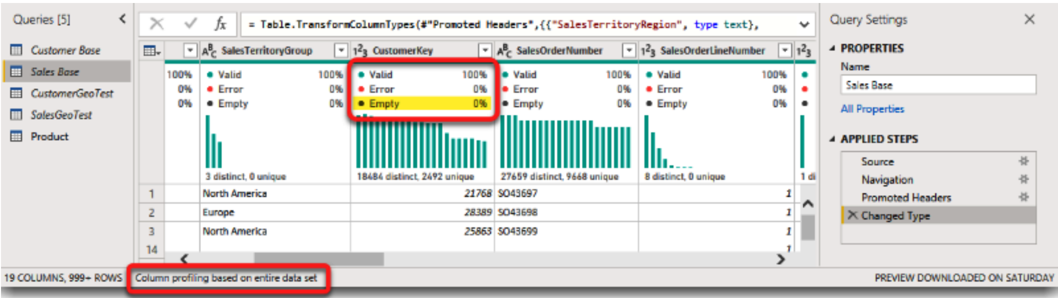


Figure 6.25 – The Column Quality information shows 0% Empty for CustomerKey

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:

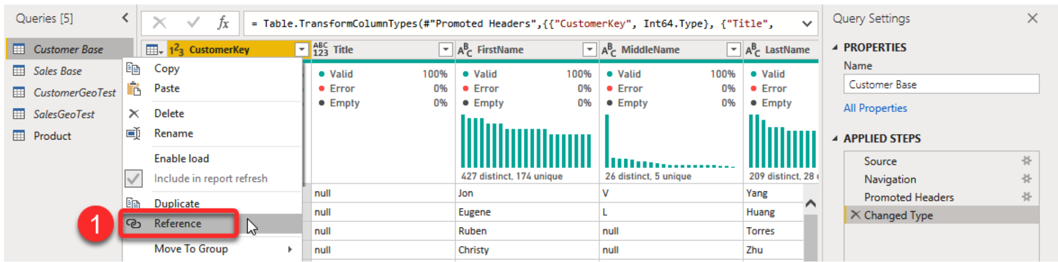


Figure 6.26 – Referencing the Customer Base table

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**:

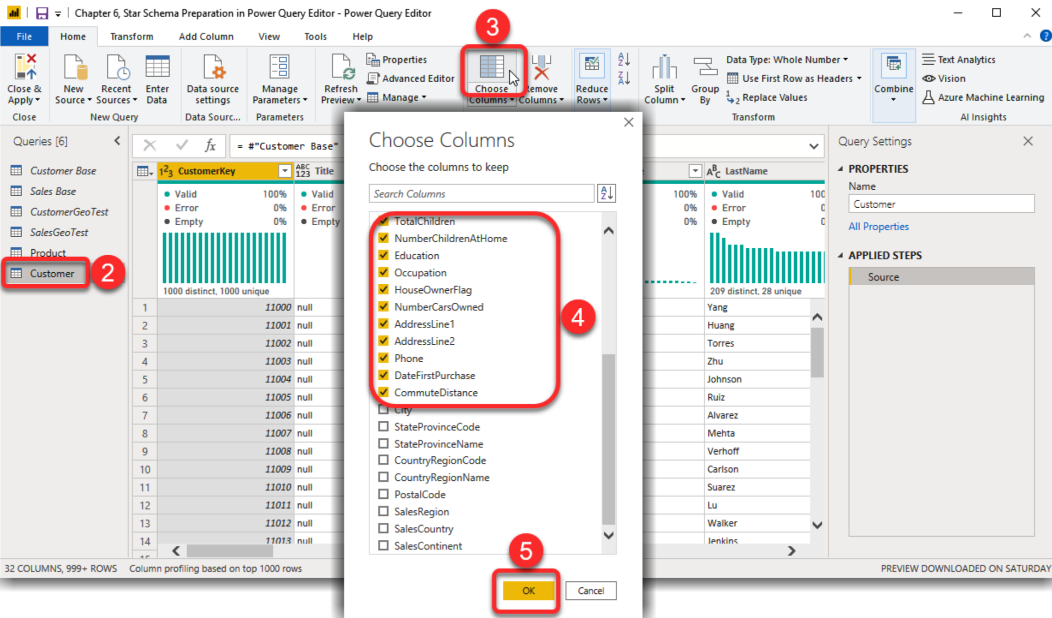


Figure 6.27 – Removing the unnecessary columns

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**:

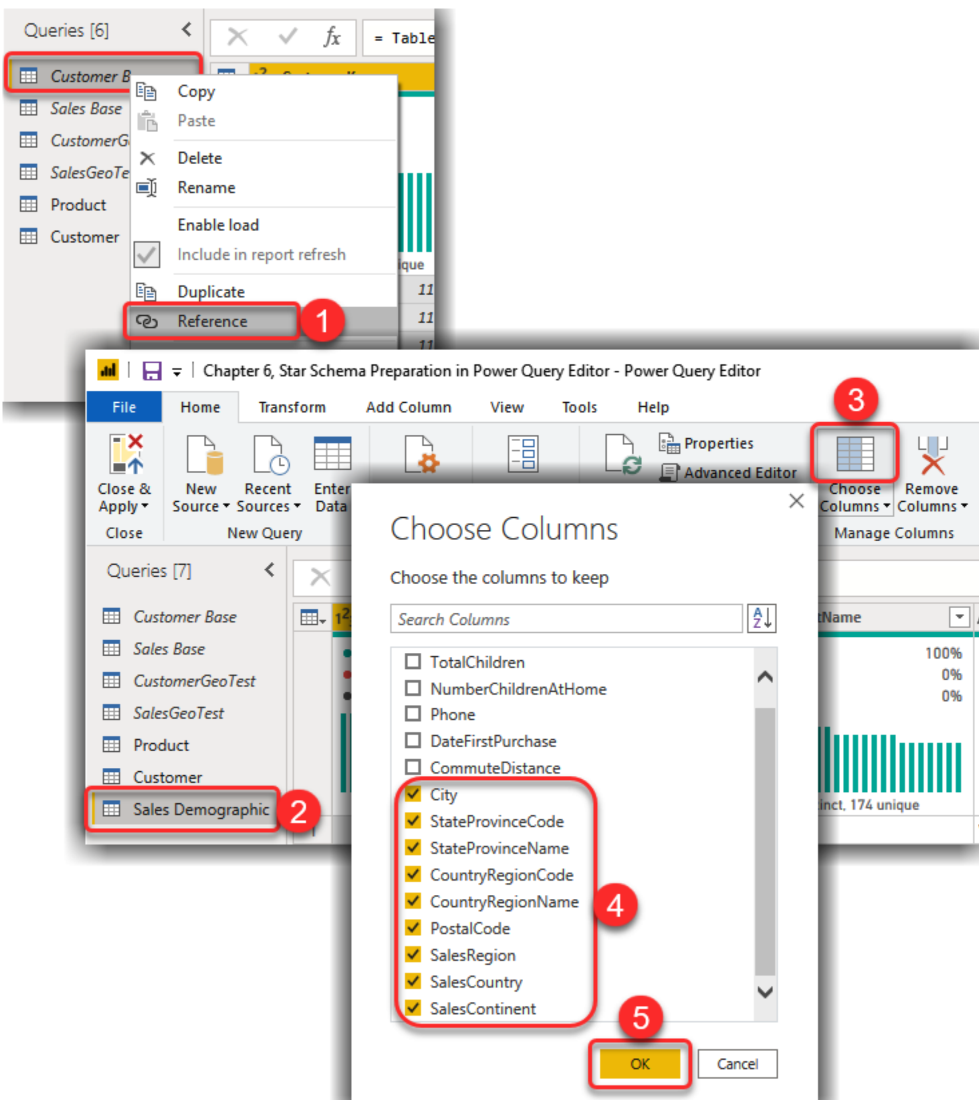


Figure 6.28 – Referencing the Customer Base table to create a Sales Demographic table and keep the relevant columns

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**:

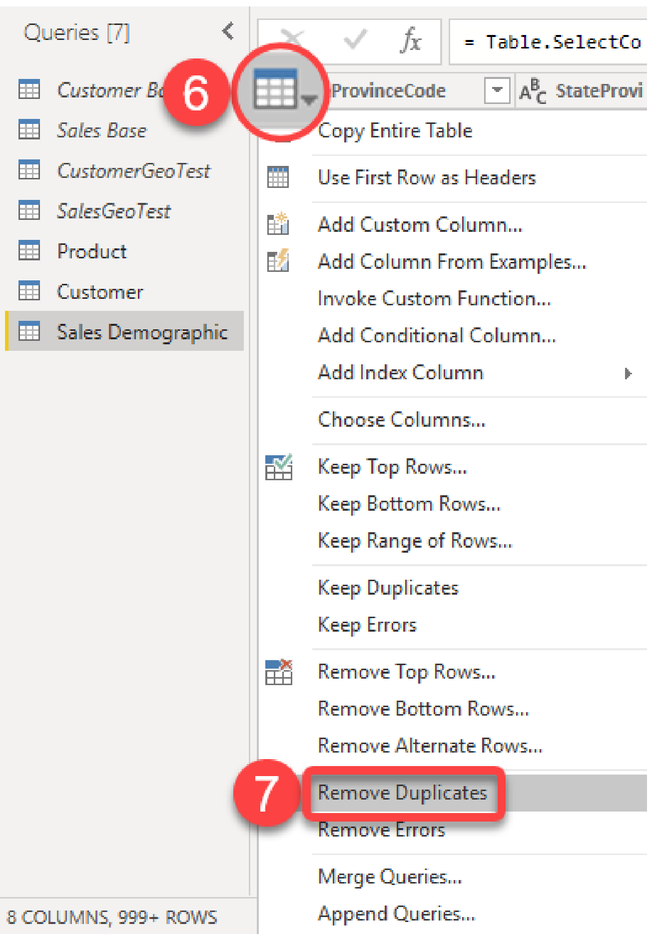


Figure 6.29 – Removing duplicates from the Sales Demographic table

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**:

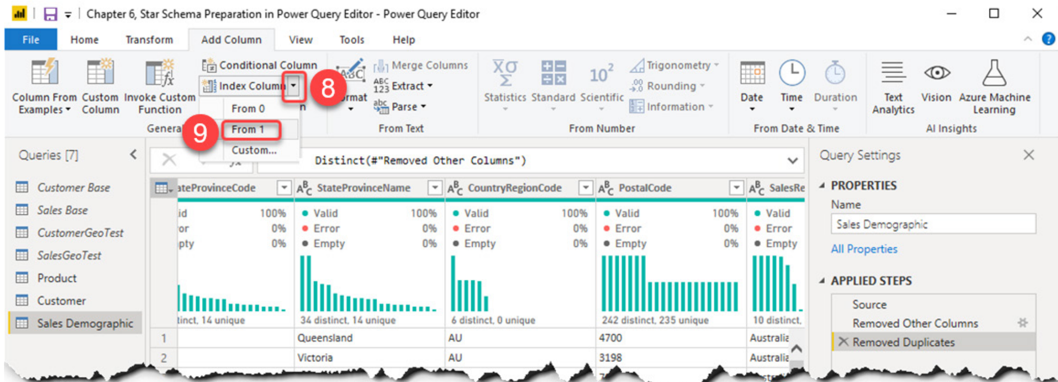


Figure 6.30 – Adding an Index column to the Sales Demographic table

10. Replace `Index` with `SalesDemographicKey` from formula bar.
11. Click the **Submit** button ✓:

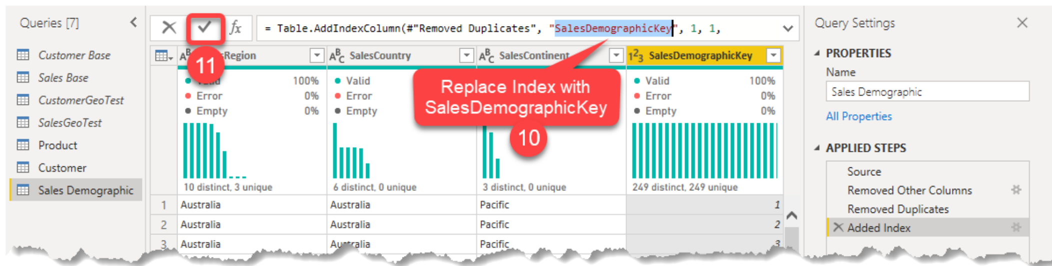


Figure 6.31 – Replacing Index with SalesDemographicKey

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**:

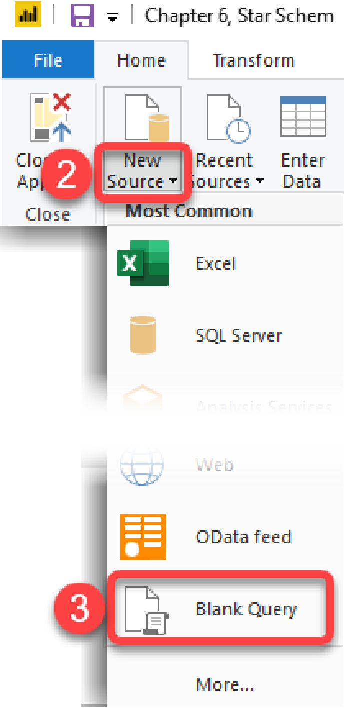


Figure 6.32 – Adding a Blank Query in Power Query Editor

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**:

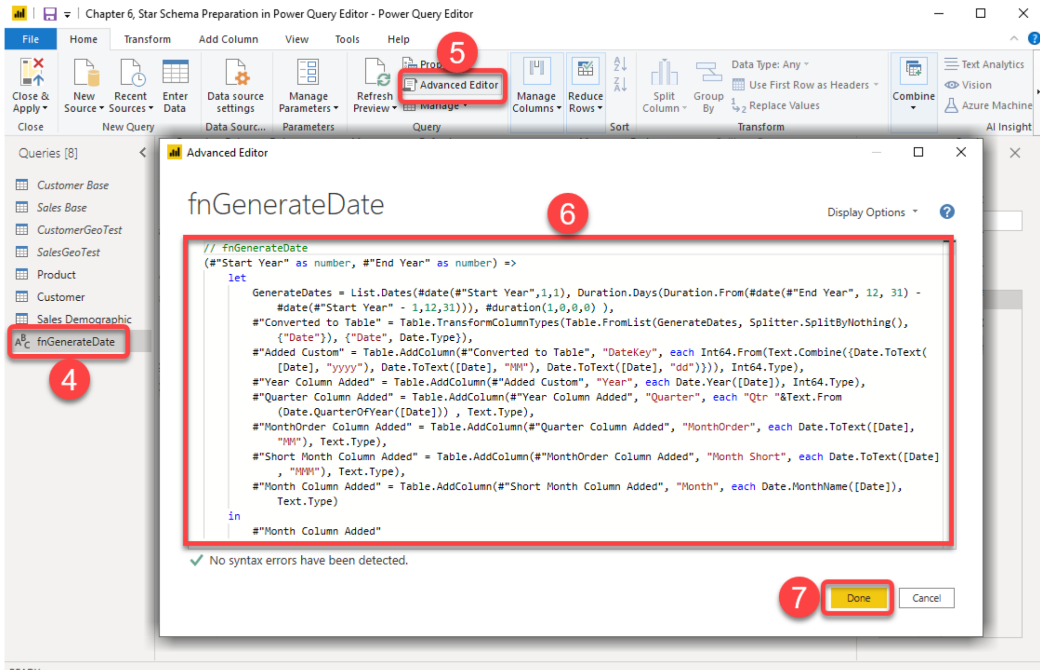


Figure 6.33 – Creating the fnGenerateDate custom function in Power Query Editor

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**:

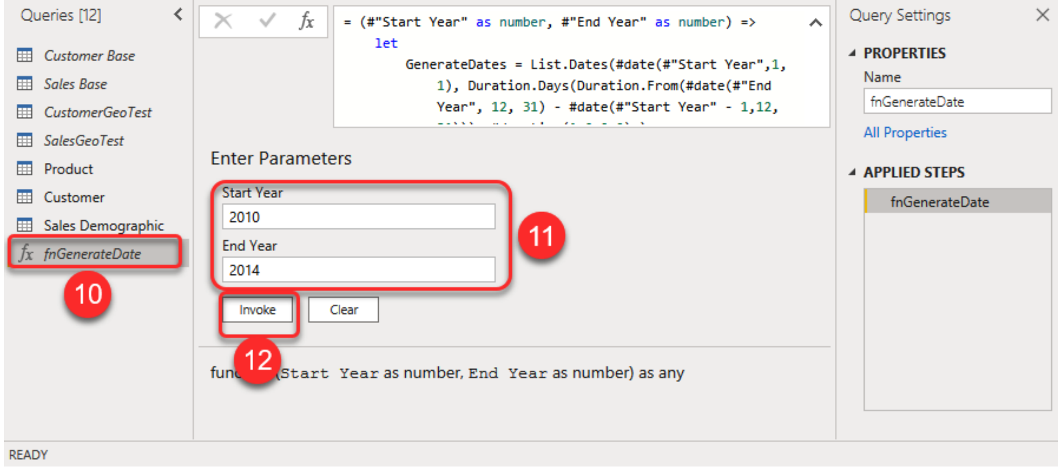


Figure 6.34 – Invoking the fnGenerateDate function

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:

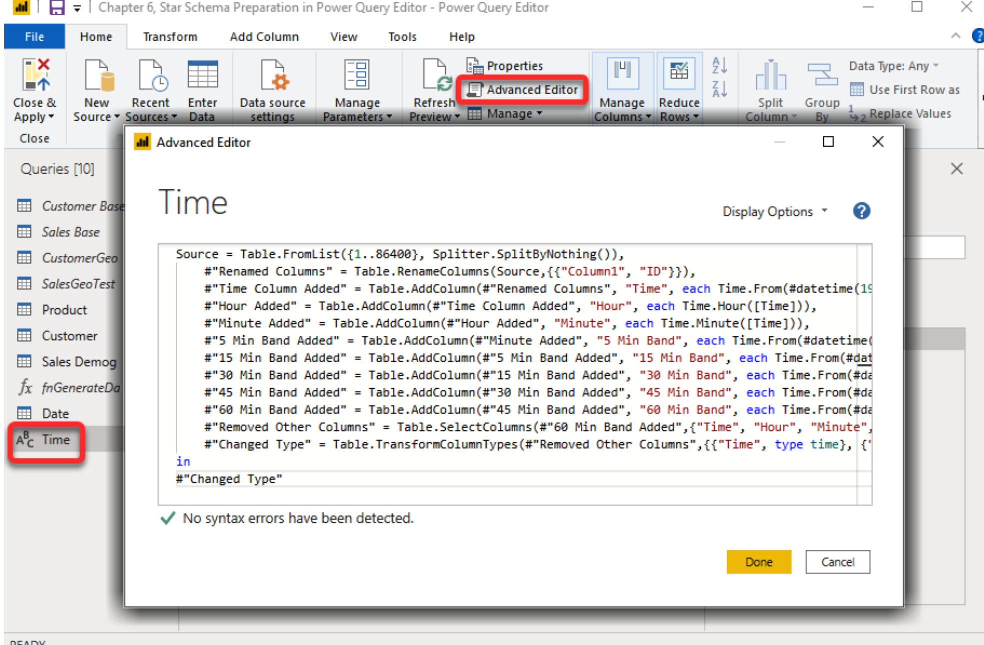


Figure 6.36 – Creating the Time dimension in Power Query Editor

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**:

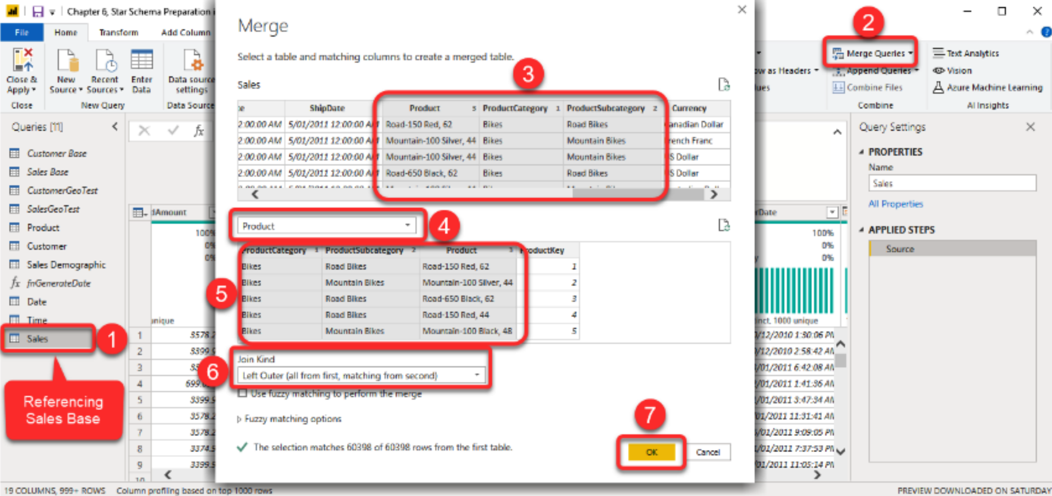


Figure 6.37 – Merging the Sales table with the Product table

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**:

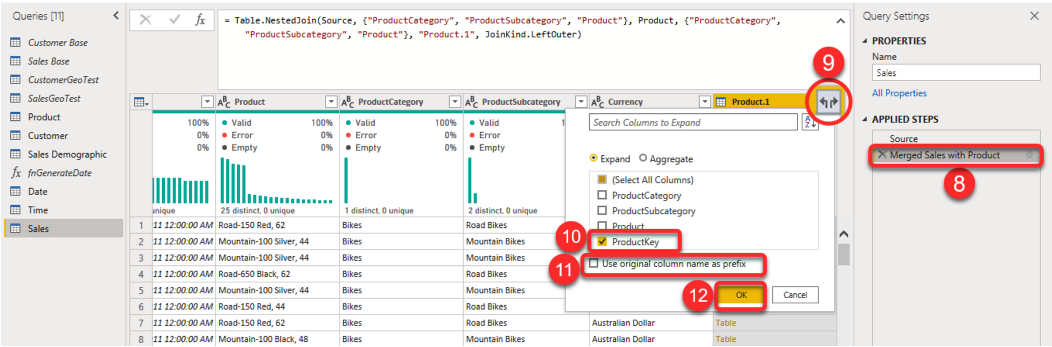


Figure 6.38 – Expanding the Product.1 structured column

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**:

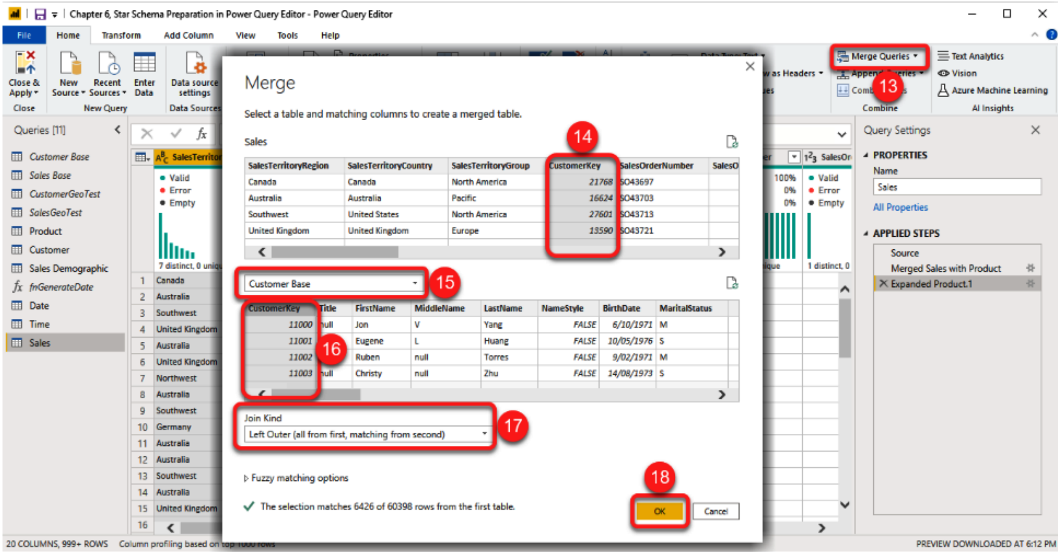


Figure 6.39 – Merging the Sales table with Customer Base

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**:

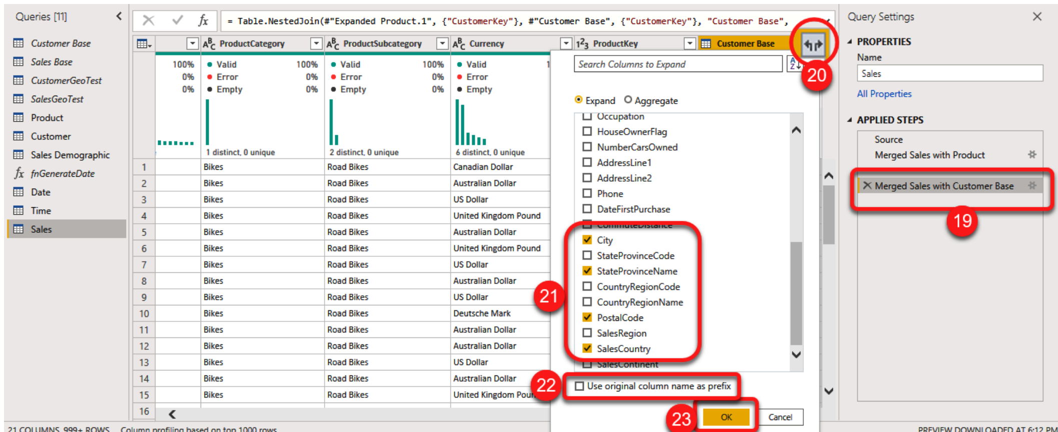


Figure 6.40 – Expanding the Customer Base structured column

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**:

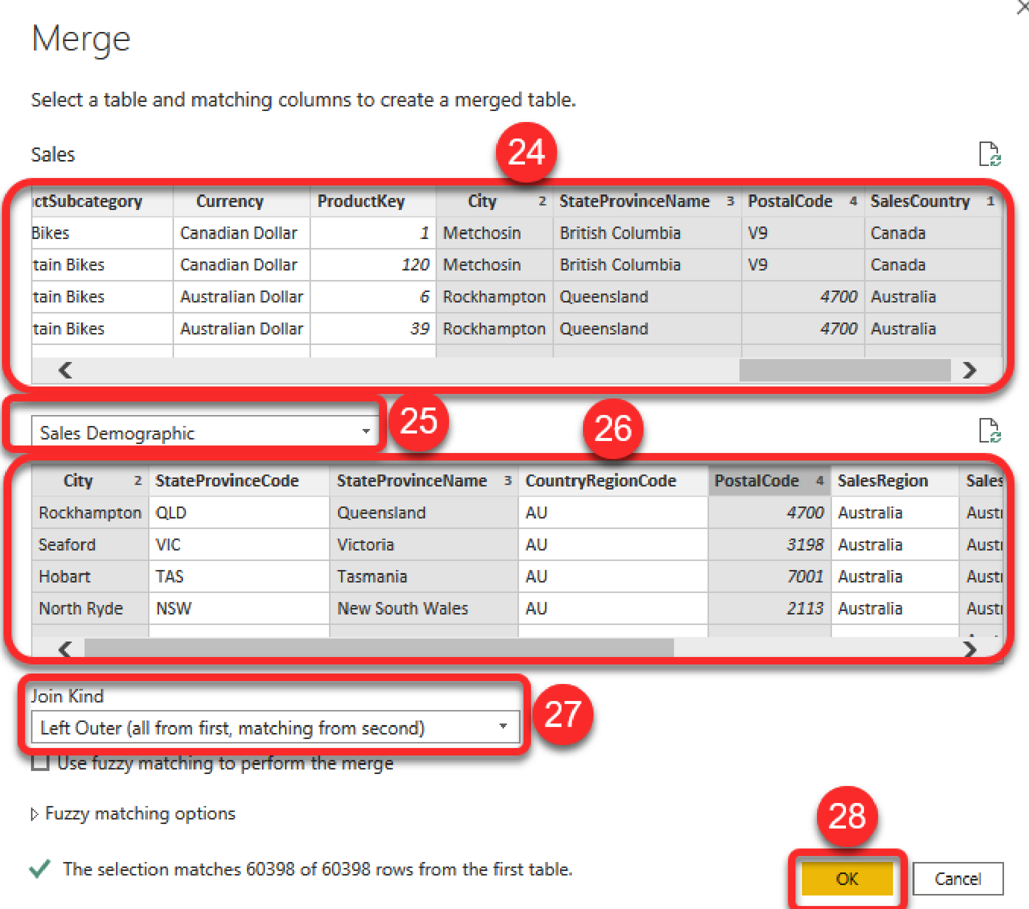


Figure 6.41 – Merging Sales with Sales Demographic

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**:

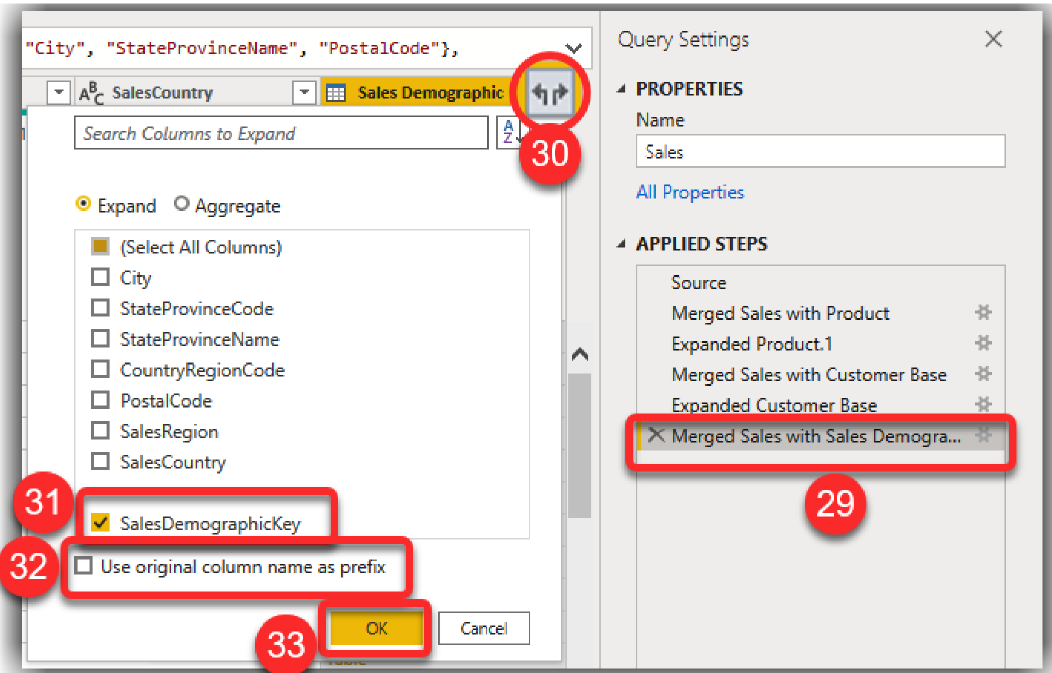


Figure 6.42 – Merging Sales with Sales Demographic

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**:

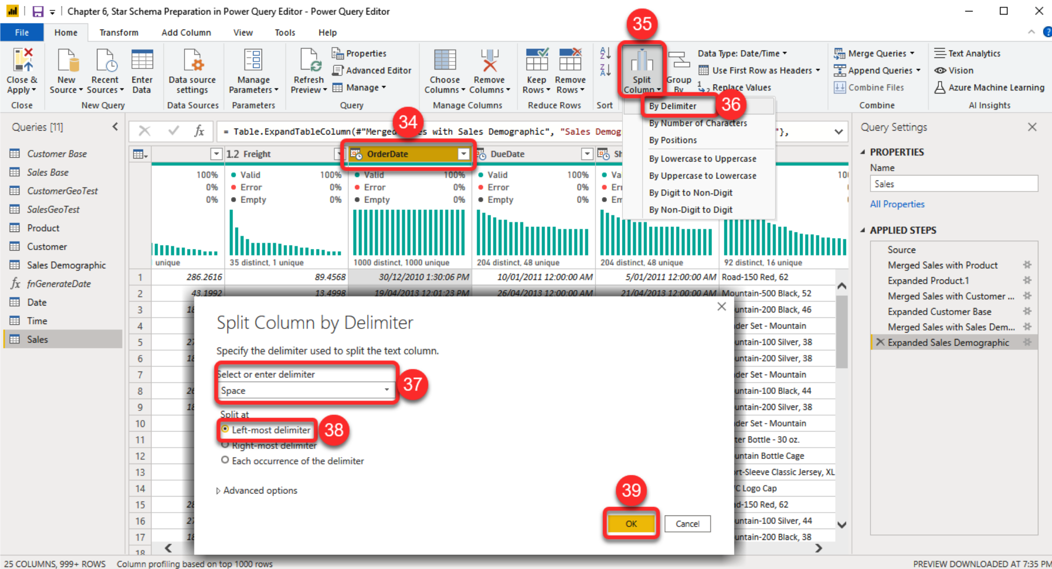


Figure 6.43 – Splitting OrderDate by delimiter

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:

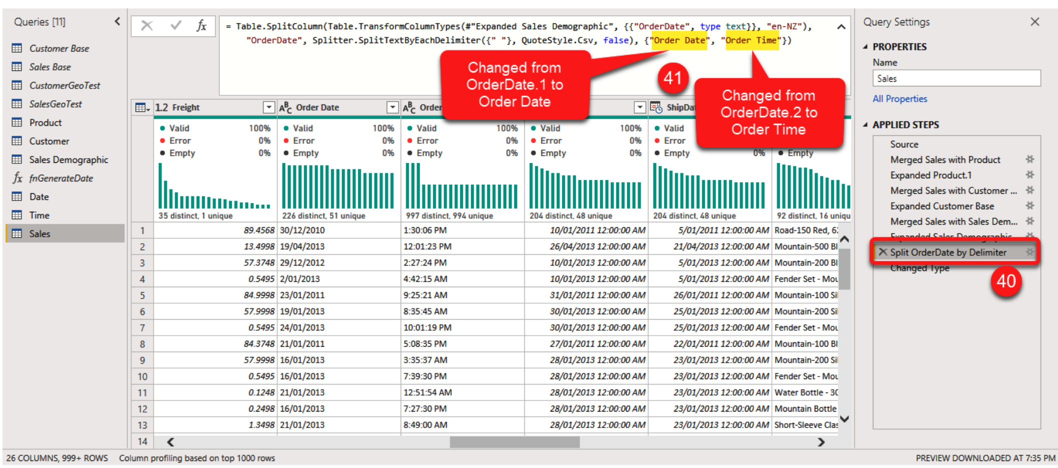


Figure 6.44 – Changing the split column names from the formula bar

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`:

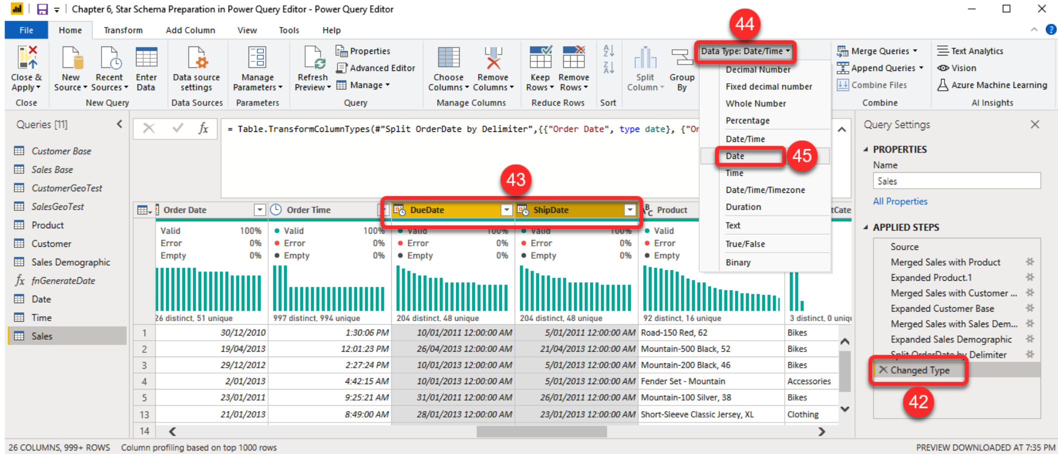


Figure 6.45 – Changing the DueDate and ShipDate data types

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**:

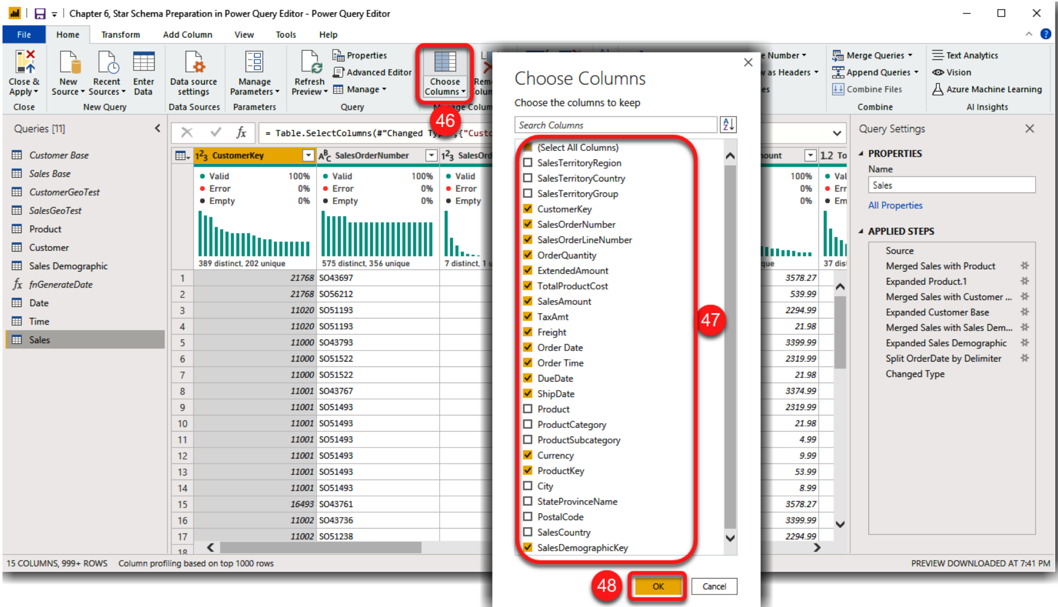


Figure 6.46 – Removing unnecessary columns from the Fact table

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.

7
Data Preparation Common Best Practices

In the previous chapter, we dealt with a flat data source, and we prepared the data in the star schema shape by identifying the dimensions and facts. Then we prepared the data to serve the star schema in the data model. In this chapter, we will look at common data preparation best practices that will help to achieve better-performing queries that are well organized and are cheaper to maintain by going through some general techniques and considerations in Power Query to avoid common pitfalls. We will look at query folding and discuss some relevant best practices. We will emphasize the importance of data conversion to avoid potential issues caused by inappropriate data conversions in the data model. We will also discuss some techniques to keep the query sizes optimized. Last but not least, we'll discuss some naming conventions that are essential for code consistency.

In this chapter, we'll discuss the following best practices:

- General data preparation considerations
- Data conversion
- Optimizing query sizes
- Naming conventions

General data preparation considerations

In this section, we'll provide more standard best practices for data preparation in Power Query. By following best practices, we are guaranteed to avoid issues down the road that are hard to identify and hence expensive to rectify.

Consider loading a proportion of data while connected to the OData data source

The **Open Data Protocol** (**OData**) was invented initially by Microsoft and is a commonly accepted method for creating and consuming REST APIs, and many systems are accessible via OData. When loading data via an OData connection into Power BI, it is essential to pay extra attention to the amount of data being loaded through the OData connection. In many cases, the underlying data model has wide tables with many columns containing metadata that is not necessarily needed.

A general rule of thumb with all kinds of data sources is only to keep relevant columns during data preparation. We need to pay even more attention to it when we are dealing with OData. I've seen Power BI reports bring production systems to their knees when the developer initially tried to load all data from wide tables with more than 200 columns. When we say **consider loading a proportion of data**, we are referring to loading data to relevant columns. In some cases, we may also need to filter the data to load the part of it that matters the most to the business.

At this point, we may need to involve the SMEs from the business too. The business sets the rules around the relevance of the data it would like to analyze. To be able to get an idea of how many tables are involved, and how many columns and rows they have, we can quickly experiment before we load the data from the OData data source within the Power Query Editor using the custom function that follows:

```
//fnODataFeedAnalysis
(ODataFeed as text) =>
let
```

```
Source = OData.Feed(ODataFeed),
        FilterTables = Table.SelectRows(Source, each Type.
Is(Value.Type([Data]), Table.Type) = true),
    #''TableColumnCount Added'' = Table.AddColumn(FilterTables,
''Table Column Count'', each Table.ColumnCount([Data]), Int64.
Type),
    #''TableCountRows Added'' = Table.
AddColumn(#''TableColumnCount Added'', ''Table Row Count'',
each Table.RowCount([Data]), Int64.Type),
    #''NumberOfDecimalColumns Added'' = Table.
AddColumn(#''TableCountRows Added'', ''Number of Decimal
Columns'', each List.Count(Table.ColumnsOfType([Data],
{Decimal.Type})), Int64.Type),
    #''ListOfDecimalColumns Added'' = Table.
AddColumn(#''NumberOfDecimalColumns Added'', ''List of Decimal
Columns'', each if [Number of Decimal Columns] = 0 then null
else Table.ColumnsOfType([Data], {Decimal.Type})),
    #''NumberOfTextColumns Added'' = Table.
AddColumn(#''ListOfDecimalColumns Added'', ''Number of Text
Columns'', each List.Count(Table.ColumnsOfType([Data], {Text.
Type})), Int64.Type),
    #''ListOfTextColumns Added'' = Table.
AddColumn(#''NumberOfTextColumns Added'', ''List of Text
Columns'', each if [Number of Text Columns] = 0 then null else
Table.ColumnsOfType([Data], {Text.Type})),
    #''Sorted Rows'' = Table.Sort(#''ListOfTextColumns
Added'',{{''Table Column Count'', Order.Descending}, {''Table
Row Count'', Order.Descending}}),
    #''Removed Other Columns'' = Table.SelectColumns(#''Sorted
Rows'',{''Name'', ''Table Column Count'', ''Table Row Count'',
''Number of Decimal Columns'', ''List of Decimal Columns'',
''Number of Text Columns'', ''List of Text Columns''})
in
    #''Removed Other Columns''
```

To invoke the preceding custom function, we need to pass the OData URL to the preceding function and it gives us a result set as follows:

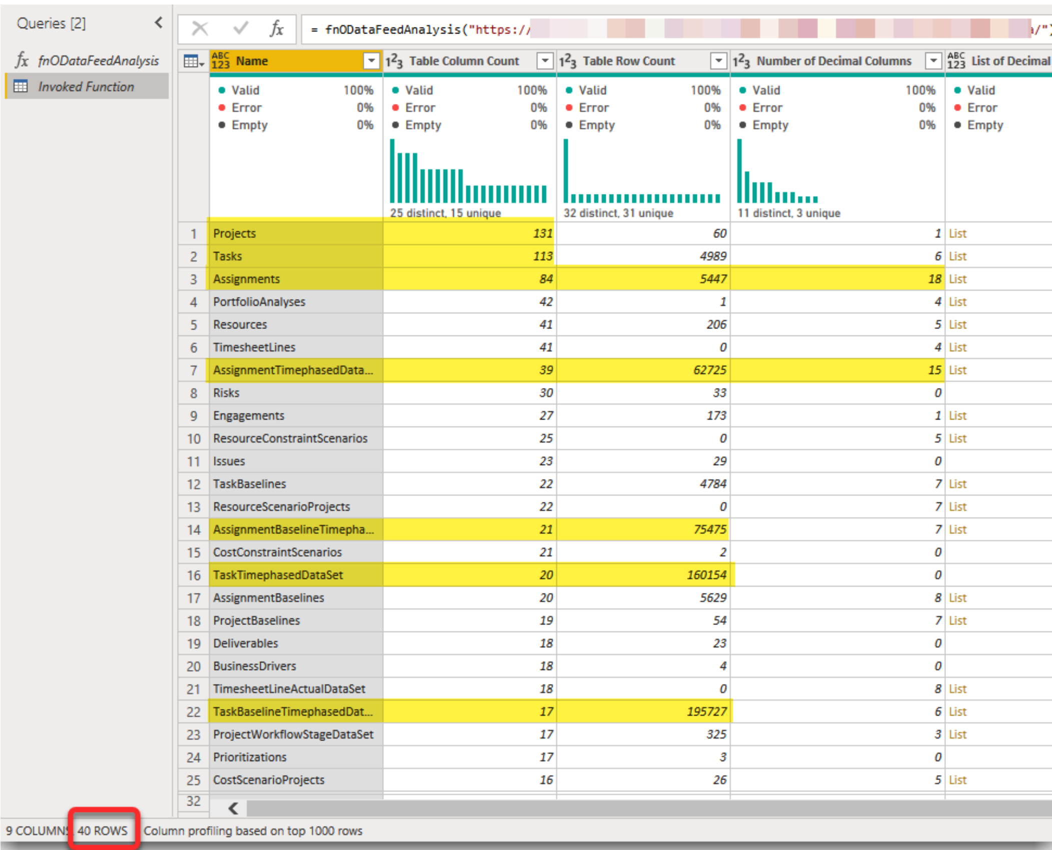


Figure 7.1 – Invoked fnODataFeedAnalysis custom function

> **Note**
>
> If you want to test the preceding custom function, you can use the Northwind test OData data source here: `https://services.odata.org/Northwind/Northwind.svc/`.

We invoked the `fnODataFeedAnalysis` custom function with the Microsoft Project Online OData URL. The function gets a list of all tables available in the OData data source and reveals the following information:

- **Name**: The names of the tables. As shown in *Figure 7.1*, the OData data source we connected to has 40 tables.

- **Table Column Count**: Shows the number of columns the data source has. This is quite handy; we can quickly identify which tables are wide and need more attention. As shown in *Figure 7.1*, the top three tables with the highest number of columns are **Projects** with 131 columns, **Tasks** with 113 columns, and **Assignments** with 84 columns.
- **Table Row Count**: Shows the number of rows each table has. As illustrated in the figure, the **TaskBaselineTimephasedDataSet** table with 195,727 rows, the **TaskTimephasedDataSet** table with 160,154 rows, and the **AssignmentBaselineTimephasedDataSet** table with 75,475 rows are the top three tallest tables in the data source.
- **Number of Decimal Columns**: Shows the count of columns with the `Decimal` datatype.
- **List of Decimal Columns**: Contains a list of columns with the `Decimal` datatype. We can click in each cell to see the column names.
- **Number of Text Columns**: Shows the count of columns with the `Text` datatype.
- **List of Text Columns**: Contains a list of columns with the `Text` datatype.

The latter six points are essential. They show the number of columns in the `Decimal`, `Text`, and `DateTimeZone` datatypes, which can consume too much memory if we do not handle them properly. By looking at the results of the preceding function, we can quickly find which tables will need more attention in the data preparation, such as the tables that are wide and tall.

Important notes on using the fnODataFeedAnalysis custom function

If a table in your data source has millions of rows, then the `#''TableCountRows Added''` step can take a long time to get the row count of the table. In that case, you may want to remove the `#''TableCountRows Added''` step from the preceding query and get the rest.

Some OData feeds result in values of type `record`. In those cases, we need to add some extra transformation steps to `fnODataFeedAnalysis`. But the current version of the function works with most OData feeds without any changes.

After we find the potentially problematic tables, we do the following:

- Eliminate all unnecessary columns.
- Filter the data to reduce the number of rows.
- We may consider changing the granularity of some tables by aggregating numeric values.
- The other crucial point is to treat the datatypes properly. We'll discuss it in a separate section in this chapter.

Appreciating case sensitivity in Power Query saves you from dealing with issues in data modeling

As explained before, Power Query is case sensitive. But case sensitivity is not just about the Power Query syntax. It is also essential to pay attention when working with data. In many cases, we have GUIDs as key columns (either a primary key or foreign key) when we mash up data from different data sources. If we want to compare the GUID values with different cases, then we'll get incorrect results. For instance, in Power Query the following values are *not* equal :

```
C54FF8C6-4E51-E711-80D4-00155D38270C
```

```
c54ff8c6-4e51-e711-80d4-00155d38270c
```

Therefore if we merge two tables joining the key columns, we get weird results. It is also the case if we load the data into the data model and create a relationship between two tables with key columns in different character cases. To fix this issue, we always keep both key columns in the same character case using either the `Text.Upper()` or `Text.Lower()` function in the Power Query Editor.

Be mindful of query folding and its impact on data refresh

Data modelers need to pay extra attention to query folding. Not only can query folding affect the performance of a data refresh but it can also hit resource utilization during the data refresh. Query folding is essential for the very same reason an incremental data refresh is. So if the refresh is taking too long due to the queries not being folded, then the incremental data refresh never happens. It is also crucial for the models in either DirectQuery or Dual storage mode as each transformation step must be folded. So, now that we know query folding is a vital topic, let's take a moment and see what it is all about.

Understanding query folding

Query folding is simply the Power Query engine's capability to translate the transformation steps to the native query language. Therefore, based on the Power Query engine's capability, a query in the Power Query Editor may be fully folded or partially folded. For instance, we connect to a SQL Server database and take some transformation steps. The Power Query engine tries to translate each transformation step to a corresponding function available in T-SQL. A query is fully folded when all query steps are translatable to T-SQL. If the Power Query engine cannot translate a step to T-SQL, then from that step onward the query is not folded anymore, so the query is partially folded.

Here is the point: when the Power Query engine can fully fold a query, it passes the generated native query to the source system and gets the results back. On the contrary, when a query is partially folded, the Power Query engine sends the part of the query that is folded back to the source system, gets the results back, and starts applying the unfolded transformation steps on our local machine by the Power Query engine itself. If the query is not folded at all, the Power Query engine itself must take care of all transformation steps.

DirectQuery and Dual storage modes and query folding

The concept of query folding in programming is referred to as server-side/client-side data processing. A fully folded query is processed on the server, which is much more efficient when the query is partially or fully processed client-side.

Therefore, the queries in either DirectQuery or Dual storage modes must be fully folded; in other words, they must be translatable to the native query supported by the data source system. Hence, when we use a Power Query function that cannot be folded over DirectQuery or Dual storage modes, we get the following warning, as also shown in *Figure 7.2*:

> **Note**
>
> This step results in a query that is not supported in DirectQuery mode. Switch all tables to Import mode.

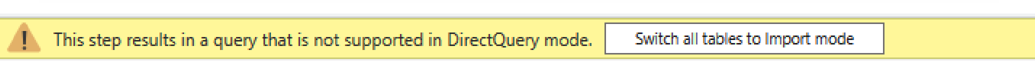


Figure 7.2 – Switch all tables to Import mode warning in DirectQuery storage mode

Data sources and query folding

Most of the data sources that have a query language support query folding, including the following:

- Relational databases that are supported in Power BI
- OData feeds
- SharePoint lists, which are basically OData feeds
- Microsoft Exchange
- Active Directory
- Microsoft Access

With that in mind, most file-based data sources, such as flat files, Excel files, blobs, and web data sources, do not support query folding.

Indications for query folding

Now that we know what query folding is, it would be good to determine when it happens and when it does not. The good news is that there are ways to indicate when a query is folded and when it is not. Depending on the storage mode and the transformation steps we take, the indication can be noticeable. We might need to take some steps to find out whether and when a query is folded.

As mentioned earlier, if the query's storage mode is DirectQuery or Dual, then the query must be fully foldable. Otherwise, we get a warning message to change the storage mode to Data Import, which indicates that the query is not foldable. But if the storage mode is already Data Import, then each step may or may not be folded. Generally speaking, all query steps translated to the data source's native language are foldable.

If we map query folding to the SQL language, then if the query and its steps are translatable to a simple `SELECT` statement including `SELECT`, `WHERE`, `GROUP BY`, all `JOIN` types, aliasing (renaming columns), and `UNION ALL` (on the same source), then the query is foldable. With that in mind, we can also check query folding by right-clicking on each applied step and seeing whether the **View Native Query** option is enabled in the context menu or not. If **View Native Query** is enabled, then the step and all previous steps are certainly folded. Otherwise, the step we are at (or some previous steps) most probably are not foldable. *Figure 7.3* shows a query on top of a SQL Server data source that is fully folded. We can click **View Native Query**, which is still enabled, to see the T-SQL translation of the current Power Query query:

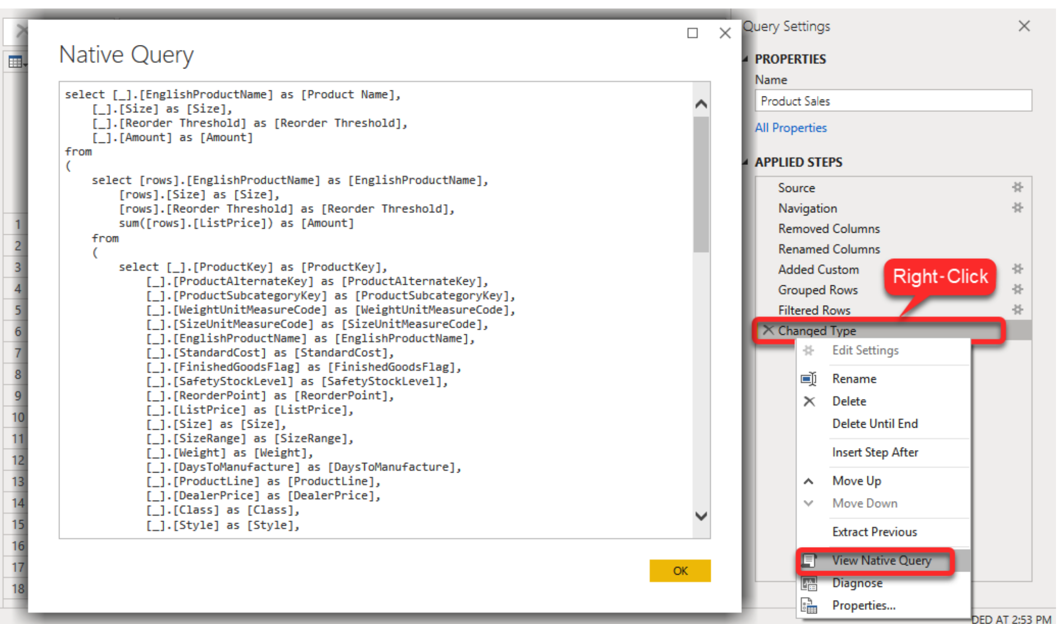


Figure 7.3 – View Native Query in the Power Query Editor

Query folding best practices

So far, we've discussed the importance of query folding and how we can determine whether a query is fully folded or not, so it is trivial now that the general rule of thumb is to always try to make a query fully folded. But there are more best practices around query folding. Let's have a look at them.

Use a SQL statement when connecting to a SQL server

When we connect to a SQL Server data source, we can write T-SQL statements such as simple `SELECT` statements or to execute stored procedures. The following screenshot shows the options available within the SQL Server connection:

Figure 7.4 – Using custom T-SQL statements when connecting to a SQL Server data source

This is a handy feature but we are conscious that when we write T-SQL statements, query folding is disabled. Therefore, the transformation steps are not folded, as shown in the following screenshot:

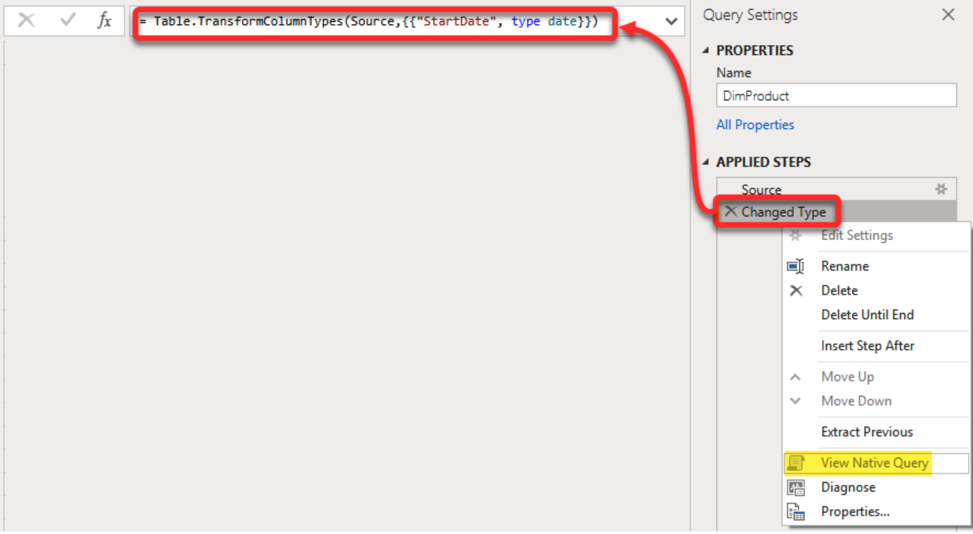


Figure 7.5 – Query folding is disabled for custom T-SQL queries when connecting to SQL Server data sources

So, it is best to take care of all transformation steps in our T-SQL statement. In that case, we will have only one step in the **APPLIED STEPS** pane in the query editor. *Figure 7.6* shows the T-SQL version of the query shown in *Figure 7.5*:

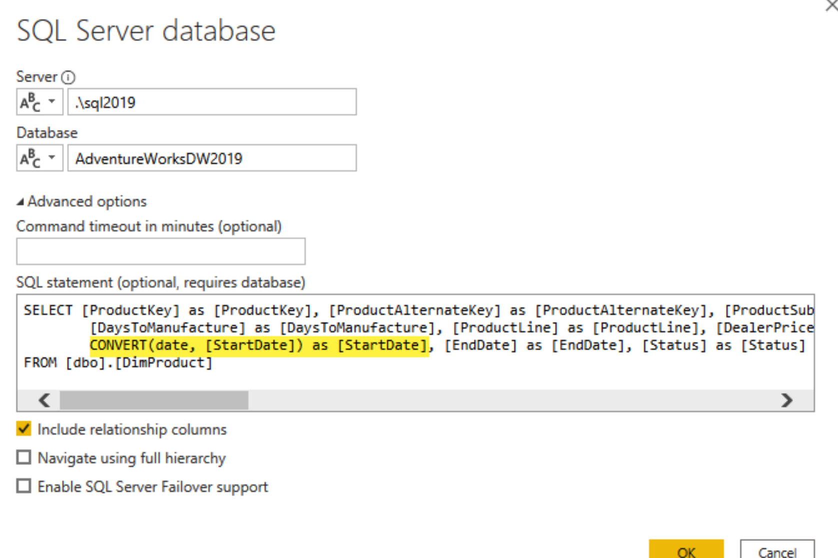


Figure 7.6 – Taking care of transformation steps in T-SQL statements

As a general rule of thumb, we never use `SELECT *`.

Push the data preparation to the source system when possible

It is advised to always push all the transformation steps to the source when possible. For instance, when we are connecting to a SQL Server data source, it is best to take care of all transformation steps on the SQL Server side by creating views, stored procedures, or tables that are populated by **Extract, Transform, and Load** (**ETL**) tools such as **SQL Server Integration Services** (**SSIS**) or Azure Data Factory.

Disabled View Native Query does not necessarily mean a transformation step is not folded

Investigating the foldability of a query or a transformation step is sometimes confusing. As mentioned earlier, an indication of a folded transformation step is to right-click a transformation step and see whether the **View Native Query** option is enabled within the context menu. But this is not true when we connect to relational databases. For instance, some transformation steps may disable **View Native Query** from the context menu. However, that step is folded back to the database. *Figure 7.7* shows a query connected to a SQL Server database. We added a simple **Keep First Rows** step to get the top 10 rows.

This transformation step disabled **View Native Query** from the context menu. But, we know that T-SQL has a `TOP` function; therefore, we expect the query to be foldable, but as *Figure 7.7* shows, **View Native Query** is disabled:

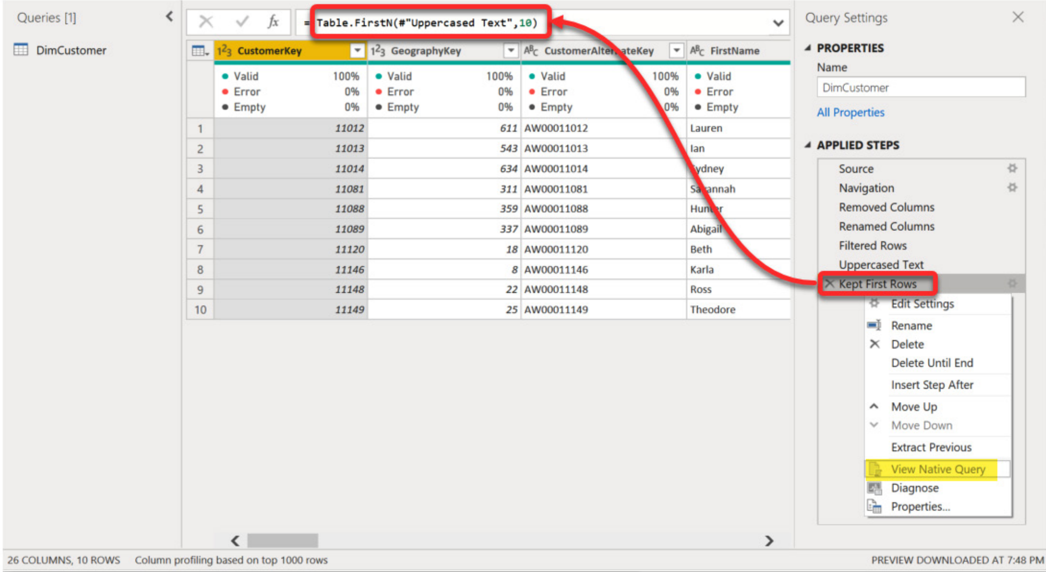


Figure 7.7 – View Native Query disabled

However, it does not mean that the query is not folded. We can use the **Power Query Diagnostics** tool to see whether the query is folded back to the server or not. In this case, we want to diagnose a step. We can right-click on the **Keep First Rows** step from the **APPLIED STEPS** pane, then click **Diagnose** as shown in *Figure 7.8*:

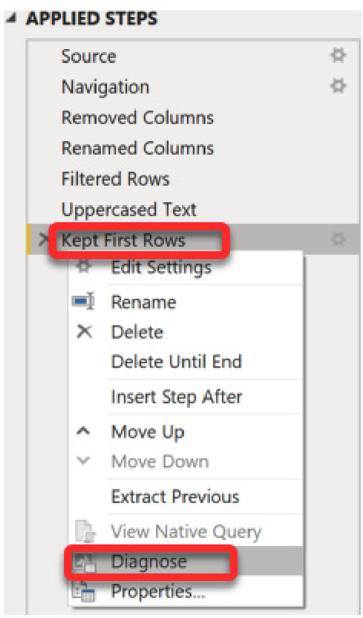


Figure 7.8 – Diagnose step in Power Query Editor

This creates a `Diagnostics` folder in the **Queries** pane, including a couple of diagnostic queries. As *Figure 7.9* shows, I have a **Detailed** query and an **Aggregated** query, which in our sample is named `DimCustomer_Kept First Rows_Aggregated`. For our sample, the **Aggregated** query gives us enough information. By clicking the `DimCustomer_Kept First Rows_Aggregated` query, we can see the diagnostic data:

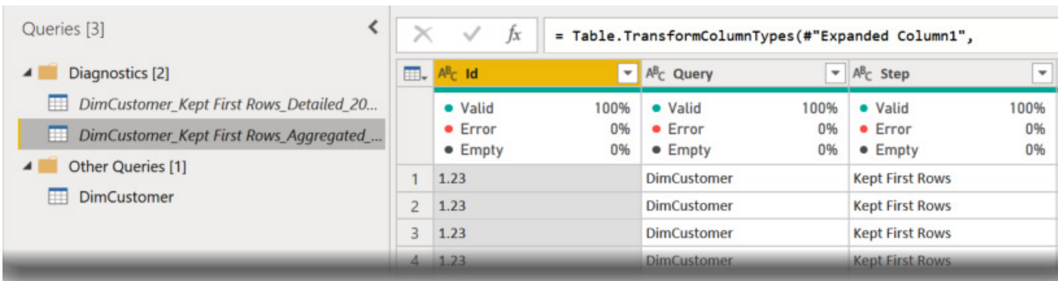


Figure 7.9 – Aggregated diagnostic query

The diagnostic query provides a lot of information. But we are only interested in the `Data Source Query` column's values, which show the actual T-SQL that Power Query sends back to SQL Server. The very last T-SQL query that appears in the `Data Source Query` column is the query we are after. As shown in *Figure 7.10*, the query indeed starts with `select top 10`, which means the **Keep First Rows** step is also folded back to SQL Server. This is quite important from a data modeling perspective, to ensure we are building efficient and performant data preparation within the Power Query layer:

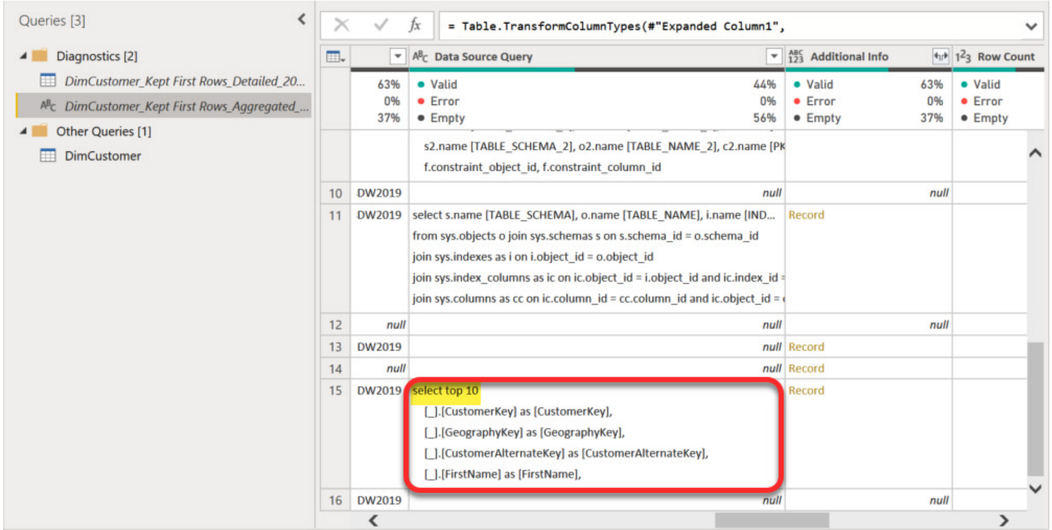


Figure 7.10 – Query folded while View Native Query is disabled

Organizing queries in Query Editor

One of the aspects of a good development model in the software development world is to keep our code and objects organized, and Power BI development is not an exception. While this best practice is not directly relevant to data modeling as such, from a support perspective, it is suggested to keep our queries as organized as possible. Organizing queries is simple. Just follow these steps:

1. Select multiple queries from the **Queries** pane.
2. Right-click then hover over **Move to Group,** then click **New Group**...
3. Enter a **name** for the group.
4. Enter a relevant **description** for the group.
5. Click **OK**.

The preceding steps are illustrated in *Figure 7.11*:

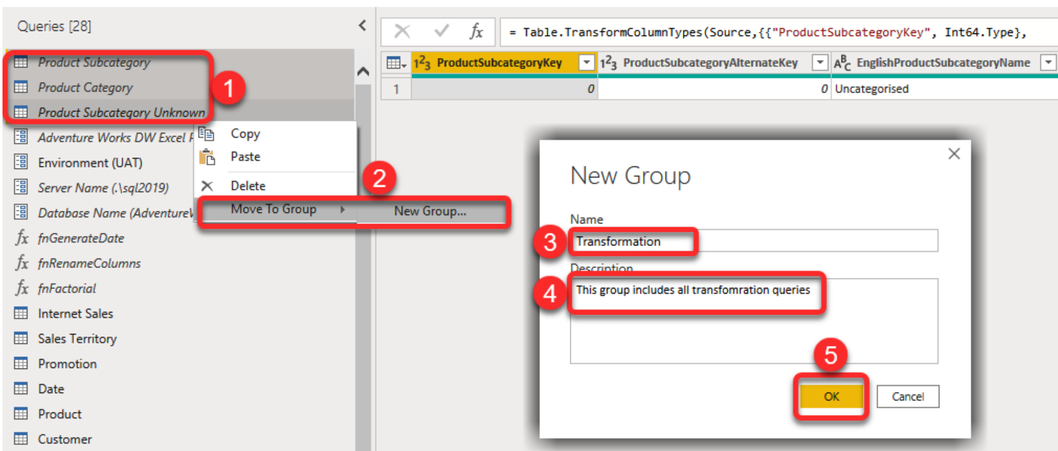


Figure 7.11 – Grouping queries in the Power Query Editor

After grouping all queries, we have organized the **Queries** pane. This is handy, especially with larger models with many queries, as the following screenshot shows:

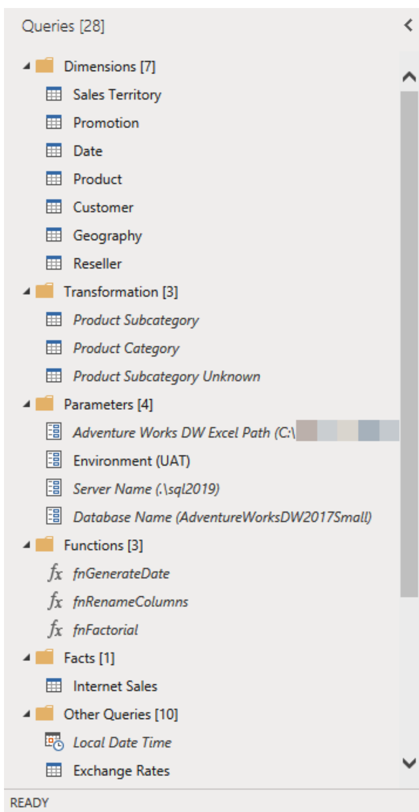


Figure 7.12 – Organized queries in the Power Query Editor

When support specialists look at an organized instance of the Power Query Editor like the preceding example, they can quickly understand how the queries relate.

datatype conversion

We previously discussed different Power Query types in *Chapter 3*, *Data Preparation in Power Query Editor*, in the *Introduction to Power Query (M)* section. We also discussed, in *Chapter 5*, *Common Data Preparation Steps*, that datatype conversion is one of the most common data preparation steps we take. In both chapters, we looked at different datatypes available in Power Query. So as a data modeler, it is crucial to understand the importance of datatype conversion. This section looks at some best practices about data conversion and how it can affect our data modeling.

Data conversion can affect data modeling

As mentioned, we already discussed the datatypes in Power Query in *Chapter 3*, *Data Preparation in Power Query Editor*, and *Chapter 5*, *Common Data Preparation Steps*. However, to emphasize the importance of understanding datatypes in Power Query, it is worth recalling it briefly in this section. In Power Query, we have only one numeric datatype, which is `number`. But wait, looking at the Power Query Editor, in the **Transform** tab, there is a **Data Type** drop-down button showing four numeric datatypes, as illustrated in *Figure 7.13*:

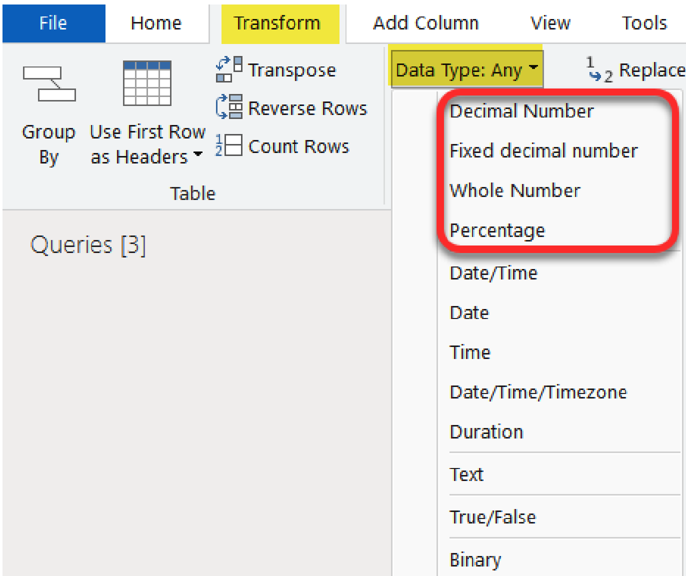


Figure 7.13 – datatype presentations in the Power Query Editor

In Power Query formula language, there is just one numeric type, which is `number`. We specify it in the Power Query syntax as `type number` or `Number.Type`. The datatypes shown in the preceding screenshot are indeed not actual datatypes. They are datatype presentations or datatype **facets**. But in the Power Query data mashup engine, they are all of type `number`. Let's look at an example to see what this means.

The following expression creates a table with different numeric values:

```
#table({''Value''}
                    , {
                        {100}
                        , {65565}
                        , {-100000}
                        , {-999.9999}
                        , {0.001}
                        , {10000000.0000001}
                        , {999999999999999999.999999999999999
99}
                    })
```

The following screenshot shows the results of the preceding expression:

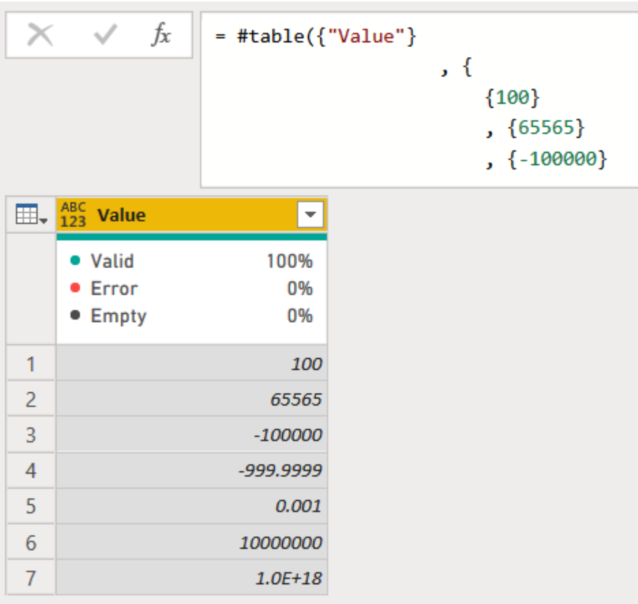


Figure 7.14 – Numeric values in Power Query

Now we add a new column that shows the datatype of each value. To do so, we can use the `Value.Type([Value])` function, giving us the type of each value of the `Value` column. The results are shown in the following screenshot:

```
= Table.AddColumn(Source, "Value Type", each
  Value.Type([Value]))
```

	Value	Value Type
	Valid 100% Error 0% Empty 0%	Valid 100% Error 0% Empty 0%
1	100	Type
2	65565	Type
3	-100000	Type
4	-999.9999	Type
5	0.001	Type
6	10000000	Type
7	1.0E+18	Type

Figure 7.15 – Getting a column's value types

To see the actual type, we have to click on each cell of the `Value Type` column, as shown in the following screenshot:

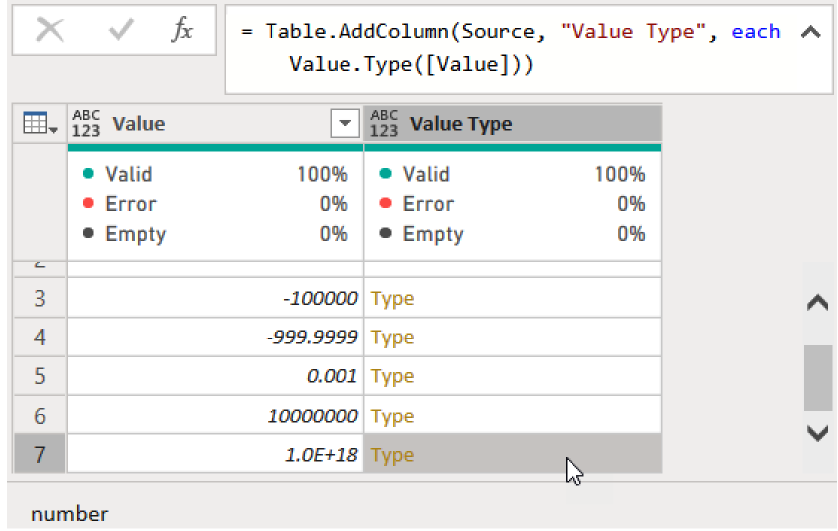


Figure 7.16 – Click on a cell to see its type

While it is not ideal to click on every single cell to be able to see the value's type, there is currently no function in Power Query that converts `Type to Text`. So, to be able to show the type as text in the table, we have to use a simple trick. There is a function in Power Query that gives a table's metadata.

The function is `Table.Schema(table as table)`. The output of the function is a table revealing informational data about a table, including `column Name`, `TypeName`, `Kind`, and so on. What we are after is to show `TypeName` for each value from the table shown in *Figure 7.15*. So, we only need to turn each value into a table. Luckily, we have a function for that in Power Query, the `Table.FromValue(value as any)` function. We then need to get the values of the `TypeName` column from the output of the `Table.Schema()` function.

So, let's add a new column to get textual values from `TypeName`. We name the new column `Datatypes`, and the expression is the following:

```
Table.AddColumn(#''Value Type Added''
                , ''Datatypes''
                , each Table.Schema(
                           Table.FromValue([Value])
                                ) [TypeName]{0}
                )
```

The results of the preceding expression are shown in *Figure 7.17*:

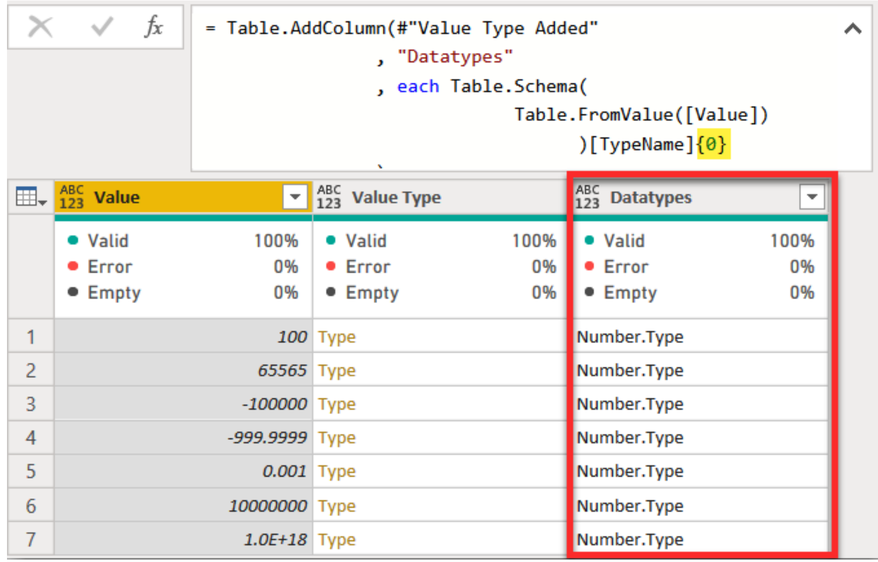


Figure 7.17 – Power Query has only one numeric type, which is Number.Type

As we can see, from a Power Query perspective, all types of numeric values are of type `Number.Type` and the way we present them in the Power Query Editor is with different **facets**, which does not make any difference in how the Power Query mashup engine treats those types. Here is the critical point: what happens after we load the data into the data model? Power BI uses the **xVelocity** in-memory data processing engine to process the data. The **xVelocity** engine uses *columnstore* indexing technology to compress the data, which works based on the cardinality of the column of the tables.

This brings us to a critical point: although the Power Query engine treats all the numeric values as the type `number`, they get compressed differently depending on their column cardinality after loading the values in the Power BI model. Therefore, it is important to set a correct type **facet** for each column.

The numeric values are one of the most common datatypes used in Power BI; let's continue with another example that shows the differences of the four different `type number` **facets**. Run the following expression in a new blank query in the Power Query Editor:

```
// Decimal Numbers with 6 Decimal
let
    Source = List.Generate(()=> 0.000001, each _ <= 10, each _
+ 0.000001 ),
    #''Converted to Table'' = Table.FromList(Source, Splitter.
SplitByNothing(), null, null, ExtraValues.Error),
    #''Renamed Columns'' = Table.RenameColumns(#''Converted to
Table'',{{''Column1'', ''Source''}}),
    #''Duplicated Source Column as Decimal'' = Table.
DuplicateColumn(#''Renamed Columns'', ''Source'', ''Decimal'',
Decimal.Type),
    #''Duplicated Source Column as Fixed Decimal'' = Table.
DuplicateColumn(#''Duplicated Source Column as Decimal'',
''Source'', ''Fixed Decimal'', Currency.Type),
    #''Duplicated Source Column as Percentage'' = Table.
DuplicateColumn(#''Duplicated Source Column as Fixed Decimal'',
''Source'', ''Percentage'', Percentage.Type)
in
    #''Duplicated Source Column as Percentage''
```

The preceding expressions create 10 million rows of decimal values between `0` and `10`. The resulting table has four columns containing the same data. The first column, **Source**, contains the values of type `any`, which translates to type `text`.

The remaining three columns are duplicated from the **Source** column with different `type number` facets, as follows:

- Decimal
- Fixed decimal
- Percentage

The following screenshot shows the resulting sample data of our expression in the Power Query Editor:

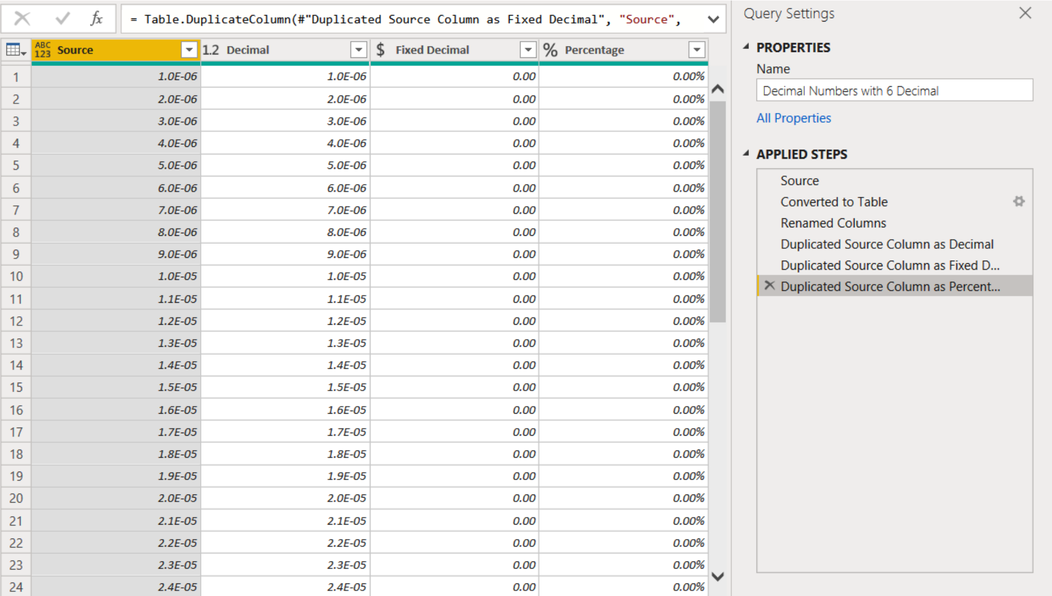


Figure 7.18 – Numeric values with different type number facets

Now click **Close & Apply** from the **Home** tab of the Power Query Editor to import the data into the data model. At this point, we need to use a third-party community tool, **DAX Studio**, which can be downloaded from the following link: `https://daxstudio.org/downloads/`.

After downloading and installing the tool, open it and connect to the current Power BI Desktop model, then follow these steps:

1. Click the **Advanced** tab.
2. Click the **View Metrics** button.

3. Click **Columns** from the **VertiPaq Analyzer Metrics** section.
4. Look at the **Cardinality**, **Col Size**, and **% Table** columns.

The results of the preceding steps are shown in *Figure 7.19*:

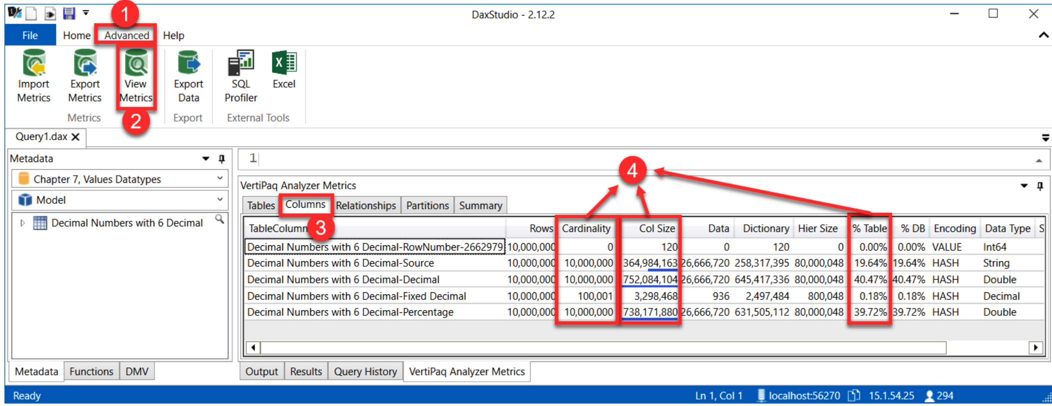


Figure 7.19 – VertiPaq Analyzer Metrics in DAX Studio

As we see, the `Decimal` column and `Percentage` consumed the largest part of the table. Their cardinality is also much higher than the `Fixed Decimal` column. So here it is: we would be better to always use the `Fixed Decimal` datatype (**facet**) for numeric values when possible.

> **Notes**
>
> By default, the `Fixed Decimal` values translate to the `Currency` datatype in DAX. So, if `Currency` is not the right format, we just need to change the column formatting.
>
> `Fixed Decimal`, as the name suggests, has a fixed four decimal points. Therefore, if the value has more decimal point digits after converting the `Fixed Decimal`, the numbers after the fourth decimal point will be cut off.

That is why the `Cardinality` column of the VertiPaq Analyzer Metrics in DAX Studio shows much lower cardinality for the `Fixed Decimal` column as the column values only keep up to four decimal points, not more.

So, the lesson to learn is to always take a datatype that makes sense to the business and is also efficient in our data model. It is advised that we use the VertiPaq Analyzer Metrics in DAX Studio to get a better understanding of the columns' datatypes in our data model.

As a data modeler, it is important to understand how the Power Query **types** and **facets** translate to DAX datatypes. Power Query and DAX are two different expression languages with their own datatypes. The following table shows the mapping between Power Query types and DAX datatypes:

Power Query Type Kind	Type Representation (Facet)	Mapping Datatype in DAX	xVelocity Internal Datatype
date	Date	Date/time	DateTime
datetime	Date/Time	Date/time	DateTime
datetimezone	Date/Time/Zone	Date/time	DateTime
duration	Duration	Decimal number	Double
logical	Logical	True/false	Boolean
number	Whole Number	Whole number	Int64
	Decimal Number	Decimal number	Double
	Fixed Decimal Number	Fixed decimal number	Decimal
	Percentage	Decimal number	Double
text	Text	Text	String
time	Time	Time	DateTime
any	Any	Text	String

Figure 7.20 – Power Query to DAX datatype mapping

Include the datatype conversion in a step when possible

One of the crucial points many Power BI developers and data modelers miss is that some of the most used Power Query functions have an optional operand to force the type of output of the function. By missing this point, we need to add at least one extra transformation step for datatype conversion, and the more transformation steps, the slower the data refresh will be. The following functions have an optional `columnType` operand that we can use to force the output datatype, which could potentially avoid adding extra steps for type conversion:

- `Table.AddColumn(table as table, newColumnName as text, columnGenerator as function, optional columnType as nullable type)`
- `Table.DuplicateColumn(table as table, columnName as text, newColumnName as text, optional columnType as nullable type)`
- `Table.AddIndexColumn(table as table, newColumnName as text, optional initialValue as nullable number, optional increment as nullable number, optional columnType as nullable type)`

For instance, in the following screenshot, I added four new columns in four steps. Then, I added another step to change the type of the new columns in a single step:

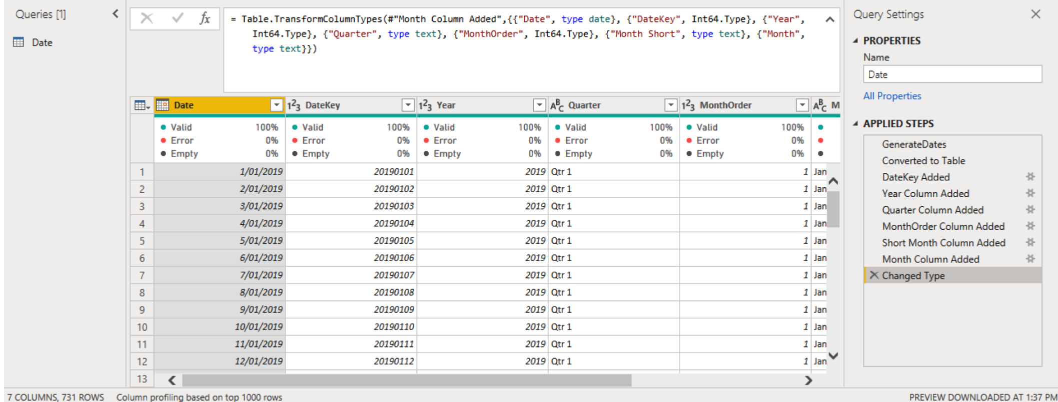


Figure 7.21 – Added new column without adding the column datatypes to each transformation step

I'll pick the `DateKey Added` step as an example to show how I wrote the expression. The `DateKey Added` expression looks like this:

```
Table.AddColumn(#''Converted to Table'', ''DateKey'', each
Int64.From(Text.Combine({Date.ToText([Date], ''yyyy''), Date.
ToText([Date], ''MM''), Date.ToText([Date], ''dd'')})))
```

The following screenshot shows that the type of the output of the `DateKey Added` expression is type `any`:

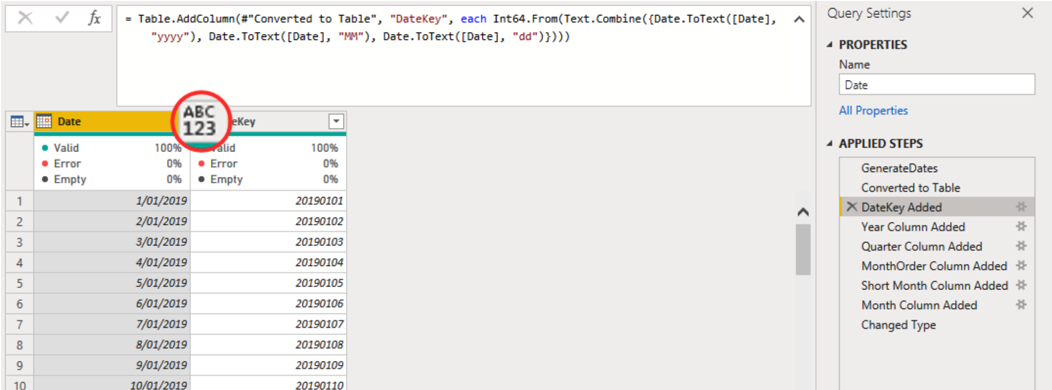


Figure 7.22 – The type of the DateKey column is type any

As explained earlier, the last operand of the `Table.AddColumn()` function in Power Query is `columnType`. So I can write the `DateKey Added` expression like the following expression instead:

```
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)
```

With the preceding expression, I explicitly mentioned the type of output. Therefore, by adding `columnType` to all other transformation steps, I can omit the **Changed Type** step.

Consider having only one datatype conversion step

It is a common habit a lot of Power BI developers and data modelers have that they convert the datatypes several times in a query during the data transformation. It is advised to avoid this habit and add only one datatype conversion step in your query when possible. The following screenshot shows a query in the Power Query Editor before and after consolidating all data conversion steps into one single step, while the results are the same:

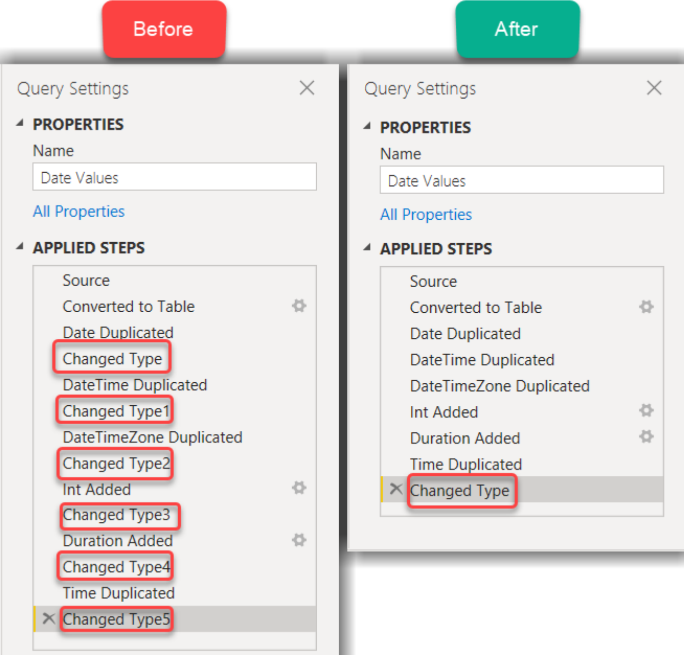


Figure 7.23 – Consolidating datatype conversion steps into one datatype conversion step

As you can see in the preceding screenshot, we can omit excessive use of the **Changed Type** step by having only one **Changed Type** step as the last transformation step.

Optimizing the size of queries

In this section, we'll look at some other data preparation best practices that can make our model a better model. Optimizing queries' sizes can reduce the data refresh time. A model with an optimized size performs better after we import the data into the data model. In the following sub-sections, we'll look at some techniques that help us in having more optimized queries.

Removing unnecessary columns and rows

In real-world scenarios, we might deal with large tables with lots of rows and columns. Some Power BI developers tend to import all columns and rows from the source, ending up having poor-performing reports. Power BI uses the **xVelocity** engine, which uses in-memory data processing for data analytics that works based on column cardinality. Therefore fewer columns directly translate to less memory consumption and as a result, a more performant data model. In many real-world cases, we need to discuss with the business to make sure the columns we will eliminate from the model will not be needed.

It is also a similar case when it comes to the number of rows. This is another common mistake a lot of Power BI developers/modelers make: they do not think about the dataset size limitation in the Power BI service. So, they create their reports importing all the data from the data source into their Power BI models. The file size gets bigger and bigger. Then, at a specific point in time, they get an error when trying to publish the report they built. The error complains about exceeding the size limit. As we discussed when we discussed the different Power BI licensing models in *Chapter 1*, *Introduction to Data Modeling in Power BI*, depending on the licensing tier we are at, we get different dataset size limitations:

- Free: 1 GB
- Pro: 1 GB
- Power BI Report Server: 2 GB
- Power BI Premium:
    - EM1/A1: 3 GB
    - EM2/A2: 5 GB
    - EM3/A3: 10 GB

- P1/A4: 25 GB
- P2/A5: 50 GB
- P3/A6: 100 GB
- P4: 200 GB
- P5: 400 GB

In many cases, we might not need to import the full history of data into the Power BI data model. So, it is advised to discuss this with the business and try to filter out the number of rows. Good data modelers always try to optimize the query size as much as possible.

Summarization (Group by)

There are many instances when we need to create reports to show summarized data. So why do we need to import all the data in its lowest grain into the data model? Well, an obvious answer is to support as many business requirements as possible, even the ones that might be required in the future. We are Power BI modelers, so we intend to create data models and use them to create reports. We can create centralized data models in Power BI that can be reused many times by different business departments. But creating centralized data models is not an easy task due to the reasons mentioned in the preceding section when we discussed the dataset size limits. So, there are many cases in which we may need to reduce the size of the data model.

Summarizing tables is one of the most effective ways of keeping the model size more optimized, therefore, we will have more performant data models. We should discuss data summarization with the business. Remember, by summarizing the data, we change the granularity of the data to a higher level.

In Power Query, we use the *Group by* feature to summarize tables. You can study the *Group by* functionality in *Chapter 5*, *Common Data Preparation Steps*, under *Common table manipulations*. We will revisit this technique in much more detail when discussing aggregations in *Chapter 10*, *Advanced Modeling Techniques*. But it was worth mentioning it as a data preparation best practice.

Disabling query load

In many cases, we do not have the luxury of proper ETL and a data warehouse in place, and we have to take care of the data transformation in Power Query. In such cases, it is a common practice to reference other queries. In many cases, we do not need to have the data within the referenced query in the data model. In those cases, the referenced query is indeed a transformation hub. Hence, we should consider disabling the query load from the Power Query Editor to avoid unnecessary data load. We discussed disabling the query load in *Chapter 3*, *Data Preparation in Power Query Editor*, in the *Query properties* section.

Naming conventions

It is essential to have naming conventions for Power BI developers and data modelers. It helps with solutions' consistency and makes the code more readable and more understandable for the support specialists. It also sets common ground that everyone across the organization interacting with our Power BI solutions can benefit from.

Data sources do not necessarily have the most user-friendly names. So, it is essential to follow a predefined naming convention during development, which will help the support specialists and contributors create new reports on top of an existing dataset. The following naming convention is suggested:

- Use camel case for object names including table names, column names, and parameter names.
- Replace underscores, dashes, hyphens, or dots between the words with space.
- Remove prefixes and suffixes from table names (such as `DimDate` becoming `Date` or `FactSales` becoming `Sales`).
- Use the shortest and most self-explanatory names for queries and transformation steps.
- Rename the default transformation steps to something more meaningful.
- For custom functions, use the `fn` prefix in the function name. Do not use any spaces in the name but keep the function name camel-cased, for example, `fnSampleCustomFunction`.
- When adding a new column supporting a specific calculation `and` it is going to be `hidden` from the end user, then use the `cal_` prefix in the column name so it can quickly be distinguished.

- Do not use acronyms in the object names unless it makes absolute sense to the business. In some businesses (such as aviation), acronyms are widely used and well understood by all end users. Therefore, in those cases, we always stick to the acronyms.
- Avoid using emojis in object names. We must not use emojis just because we can, and we think they look cool.
- Avoid using numbers in object names. In some cases, developers use a numeric value as a prefix for object names to sort. This is strongly against naming convention best practices.

You may have a more granular naming convention than the preceding list. However, if you do not have one, this can be a good starting point to think about the importance of having naming conventions in place.

Summary

In this chapter, we discussed some of the most critical best practices for data preparation. We cannot cover all the best practices in a chapter or two, but we tried to cover some of the most important ones in this chapter. We learned how to optimize query sizes in Power Query and discussed how case sensitivity in Power Query could affect our data model. Going ahead, we learned the importance of query folding and how we can identify whether a query is folded. We then looked at some datatype conversion best practices and how to reduce the number of steps by avoiding unnecessary steps. We then discussed the importance of having naming conventions.

In the next chapter, we will discuss data modeling components and building a star schema.

Section 3: Data Modeling

Everything you have learned so far comes together in this section, in which we will build a well-designed data model in Power BI. The section starts with data modeling components from a Power BI point of view. Then the concept of granularity is discussed. While in the previous chapters you prepared the building blocks of your data model, it is now time to physically build the model with real-world hands-on scenarios. This section also explains config tables and walks you through some scenarios in which you need to take advantage of the power of config tables. This section ends with naming conventions and data modeling best practices.

This section comprises the following chapters:

- *Chapter 8, Data Modeling Components*
- *Chapter 9, Star Schema and Data Modeling Common Best Practices*

8
Data Modeling Components

In the previous chapter, we learned about some critical data preparation best practices such as loading a proportion of data, removing unnecessary columns, and summarization to optimize our data model size. We also learned about query folding and how it can affect our data modeling in Power BI. We also looked at data type conversion and discussed the importance of selecting certain data types, in order to keep our data model more optimized when we import the data into the data model. This brings us to this chapter. All our data preparation efforts pay off by having a cleaner data model that is easier to maintain and performs well. In this chapter, we'll look more closely at data modeling in Power BI by covering the following topics:

- Data modeling in Power BI Desktop
- Understanding tables
- Understanding fields
- Using relationships

In this chapter, we'll work on the `Chapter 8, Data Modelling and Star Schema.pbix` sample file. It is a copy of the sample file that resulted from our work in *Chapter 6, Star Schema Preparation in Query Editor*.

Data modeling in Power BI Desktop

In Power BI Desktop, the central premise for data modeling is the **Model** tab in Power BI Desktop's main window. However, at the time of writing this book, we cannot take any expression-based activities from the **Model** tab, such as creating a new measure, calculated column, or calculated table. The following screenshot shows the **Model** tab in Power BI Desktop:

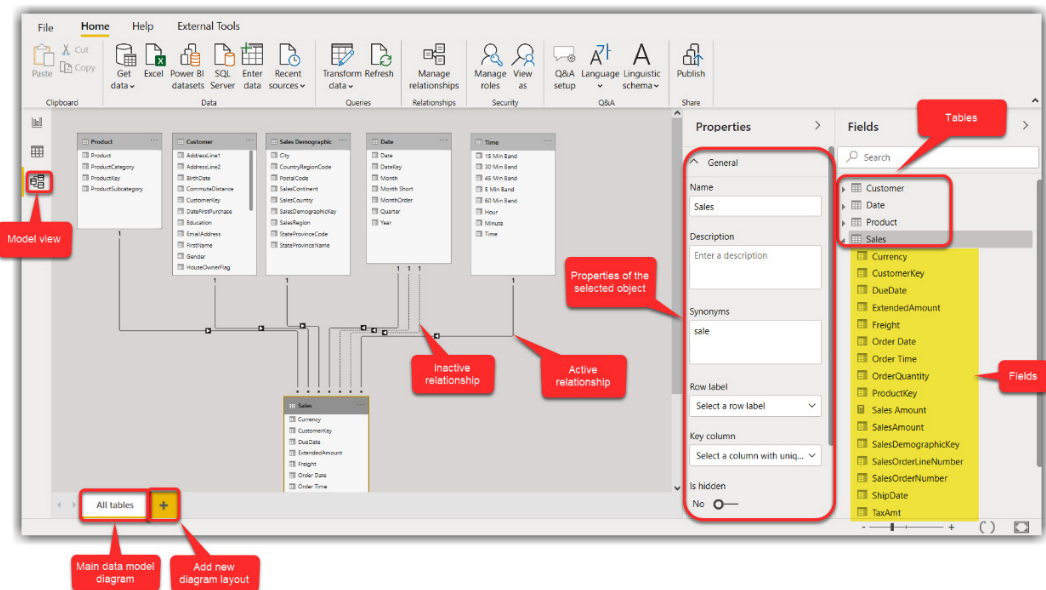


Figure 8.1 – Model view tab in Power BI Desktop

In the following few sections, we will discuss the modeling features currently available in Power BI Desktop. Then, we'll continue to build out the star schema that we prepared in *Chapter 6, Star Schema Preparation in Query Editor*.

Understanding tables

From a data modeling perspective, tables are objects that contain related data values by using columns and rows. In Power BI, each query with **Enable load** activated (within **Power Query Editor**) presents a table in the data model layer.

Table properties

By clicking on a table from the **Model** view in Power BI Desktop, tables properties show up in the **Properties** pane, as shown in the following screenshot:

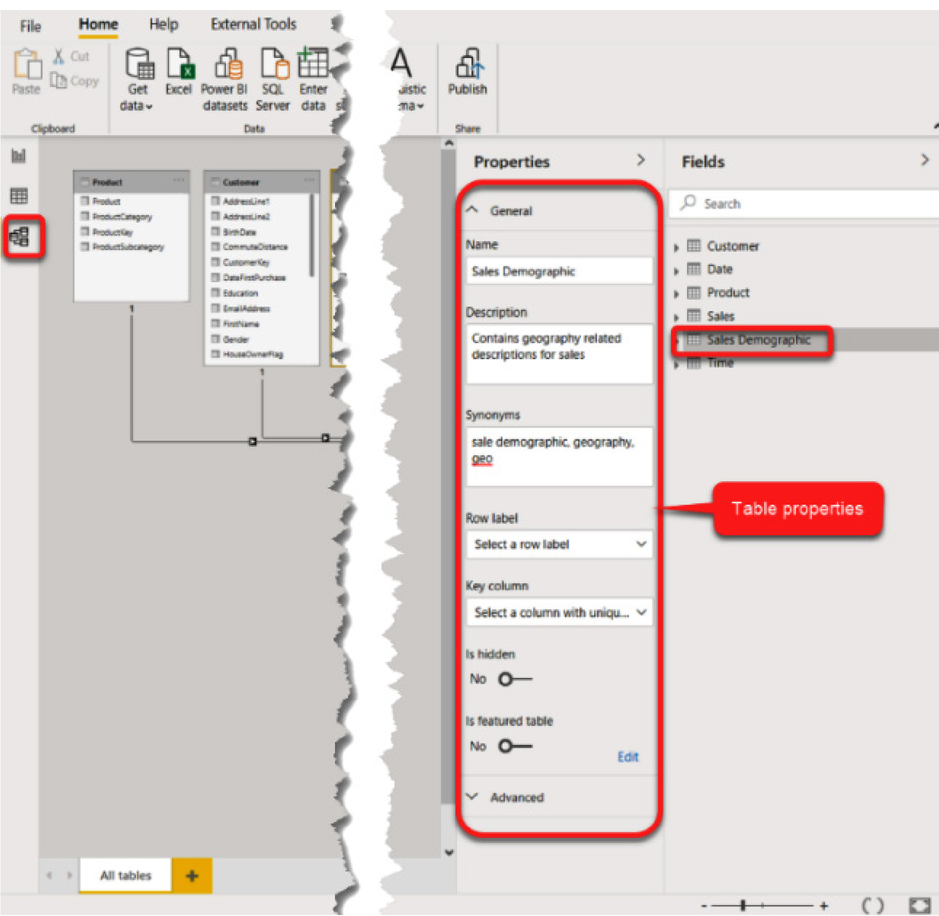


Figure 8.2 – Table properties pane in the Model view

Within the **Properties** pane, we can see the following settings:

- **General**: Includes the following general table properties:

  a. **Name**: The name of the table. We can rename a table from here.

  b. **Description**: We can write some explanations about the table here. These explanations then show up in the **Data** view, as well as in the **Report** view, as shown in the following screenshot:

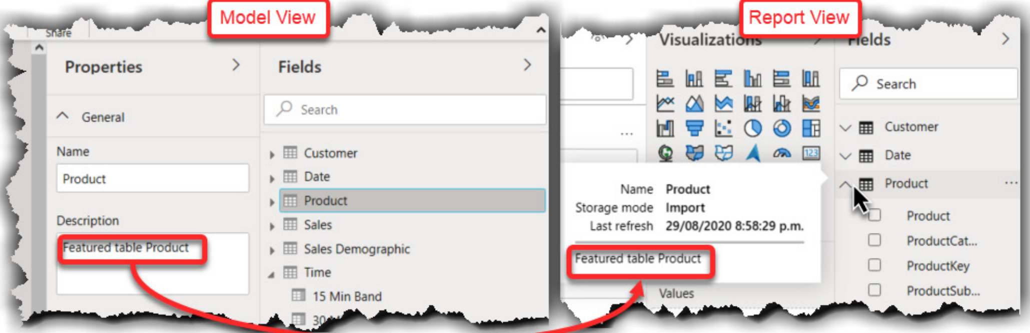


Figure 8.3 – The table description shows up in the Report view when hovering over a table

We can also use the description for documentation purposes.

  c. **Synonyms**: Adding synonyms helps explicitly with Power BI's **Q&A** feature. The users usually use a variety of terms to refer to the same thing. For instance, we have a table in the model named **Sales Demographic**. However, users may refer to it as **geography** or **geo**. So, we can add those phrases to the Synonyms box to help Q&A identify the Sales Demographic table if the user uses any of those phrases.

  d. **Row label**: At the time of writing this book, this setting affects the data model in two ways, as follows:

- **Q&A**: It helps Q&A create more helpful visuals. A **Row label** defines which column best describes a single row of data. For instance, in the `Product` table, the row label is usually the `Product` (or `Product Name`) column. Therefore, when the user asks `Sales by product`, Q&A treats `Product` as a column instead of a table. The following screenshot shows the results of asking the `sale by product as clustered column chart` question from Q&A:

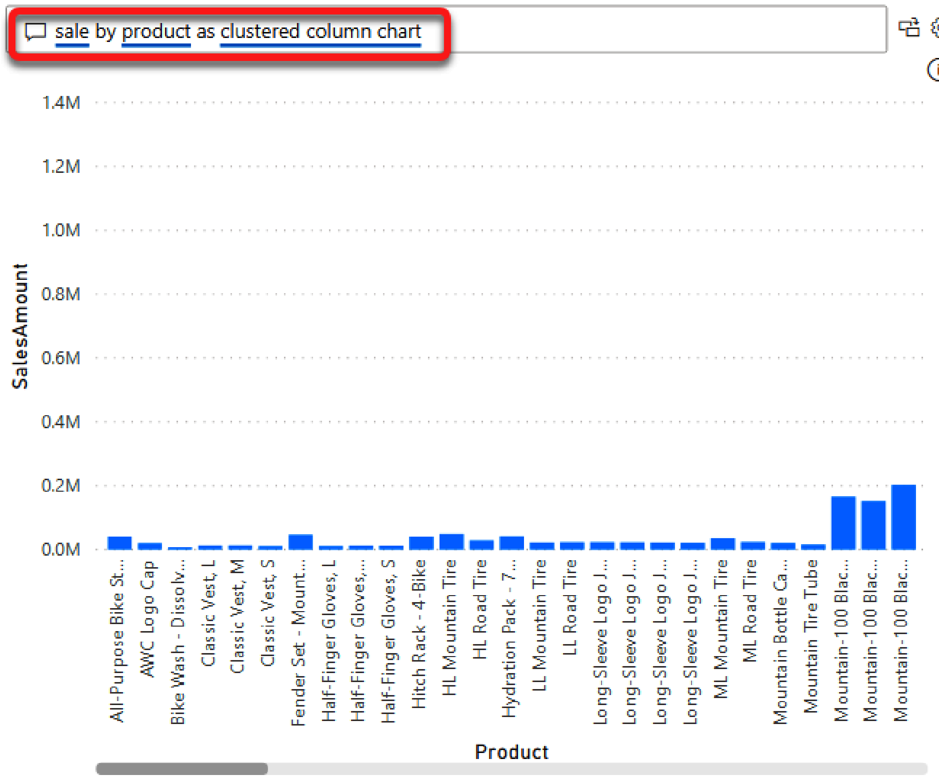


Figure 8.4 – Row label from the table properties helps Q&A provide better results

- **Featured table**: When we set the table as a featured table in Power BI Desktop, the column that was selected as **Row label** is used in Excel to identify the row quickly. More on this in the *Is featured table* bullet point.

  a. **Key column**: Provides a unique identifier for each row of the table. **Key column** will then be used in Excel to link a cell's value to a corresponding row in the table.

> **Note**
> At the time of writing this book, **Key column** is part of **featured table**, which is currently in public preview.

  b. **Is hidden**: By toggling this setting to **Yes** or **No**, we can make a table hide or unhide.

c. **Is featured table**: With this setting, we can make a table a Featured Table, which makes the table's data accessible via Excel. After publishing the data model to the Power BI service, only the specified users can access these featured tables. At the time of writing this book, the featured tables feature is in public preview, so it may look a bit different in the version of Power BI Desktop you're currently using. In the meanwhile, let's look at the **Is featured table** setting. We'll look at the Featured Table concept in more detail in the next section. To configure this setting, we must toggle it to **Yes**. To turn this feature off if it is already on, we must toggle it to **No**. To edit the **Is featured table** setting, we can click the **Edit** hyperlink, which opens the **Setup this featured table** window. In the following screenshot, we set the `Product` table as a Featured Table by putting a short description in the **Description** text box, selecting the `Product` column as **Row label**, and selecting the `ProductKey` column as **Key column**:

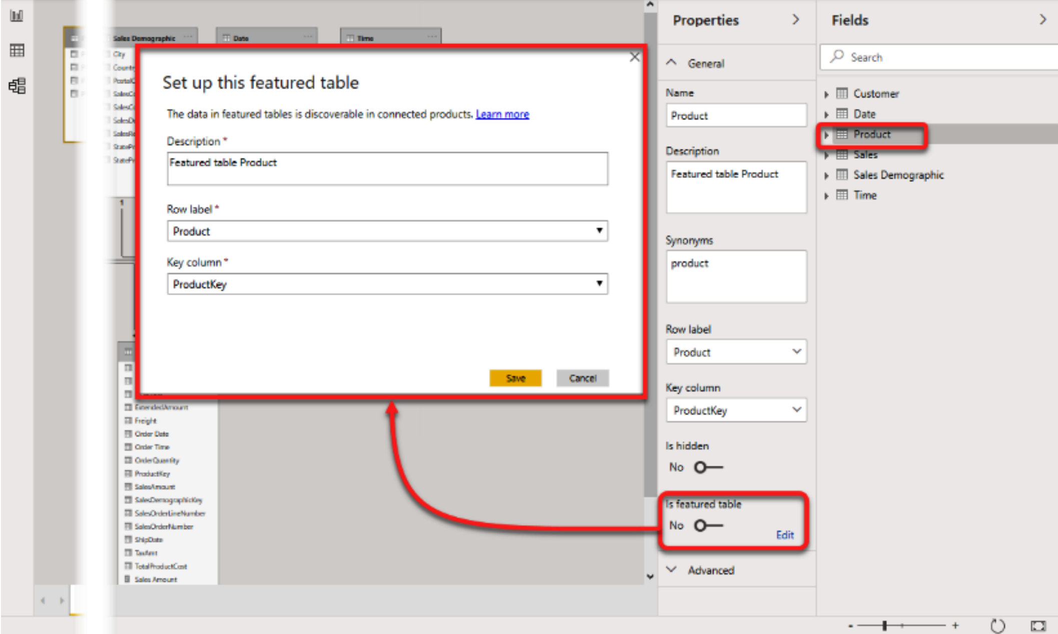


Figure 8.5 – Setting up Featured Table from the Model view

- **Advanced**: The tables currently have only one advanced setting, which is storage mode.

a. **Storage mode**: This shows the storage mode of the table. We already discussed the available storage modes in *Chapter 4*, *Getting Data from Various Sources*, in the *Storage modes* section.

Featured tables

The concept of featured tables comes from the reusability mindset where we take the prepared and already polished data across the organization in a secure way. The Power BI admins then have granular control of configuring or monitoring the featured table, as well as who can publish or update the featured tables or who, within the organization, can access the featured tables. We explained how to set the table as a featured table in the previous section. After setting the table as a featured table from the **Model** view, the data (including measures held by a table) will be available the **Data Types Gallery** in **Excel** after publishing the model to a modern **Workspace** in the Power BI service.

The following screenshot shows a new Excel file when the user types in a product name, and then selects that `Product` from **Data Types Gallery** from the **Data** tab in **Excel**:

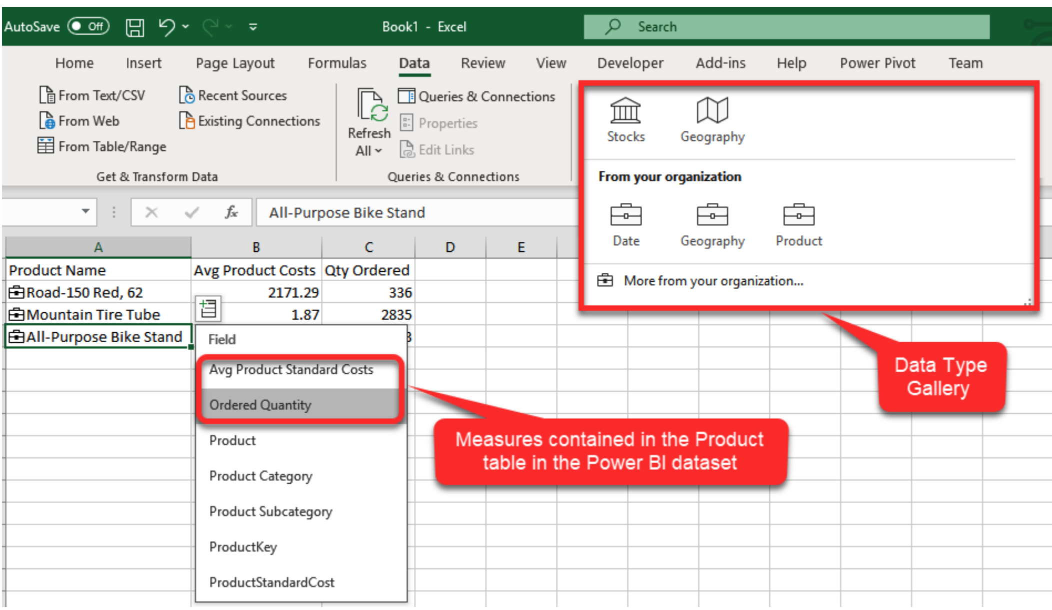


Figure 8.6 – Getting Product data in Excel from the Product table

There are many benefits of setting a table. Accessing and using featured tables in Excel is outside the scope of this book. You can learn more about using featured tables in Excel via the following link:

`https://docs.microsoft.com/en-us/power-bi/collaborate-share/service-excel-featured-tables?WT.mc_id=5003466.`

Calculated tables

In Power BI Desktop, we can create new tables using DAX expressions. Calculated tables are physical tables that are generated as a result of using table functions or constructors in DAX. Unlike **Virtual Tables**, which we already discussed in *Chapter 2*, *Data Analysis eXpressions and Data Modeling*, in the *Understanding virtual tables* section, calculated tables are much easier to work with as they are visually visible within the **Model** view in Power BI Desktop.

The following table shows the most common table functions in DAX that can be used to create a calculated table:

ADDCOLUMNS()	CALENDARAUTO()	FILTERS()	SELECTCOLUMNS()	VALUES()
ADDMISSINGITEMS()	CALCULATETABLE()	GENERATESERIES()	SUMMARIZE()	Table Constructor {}
ALL()	CROSSJOIN()	INTERSECT()	SUMMARIZECOLUMNS()	
ALLEXCEPT()	DATATABLE()	NATURALINNERJOIN()	TOPN()	
ALLSELECTED()	DISTINCT()	NATURALLEFTOUTERJOIN()	TREATAS()	
CALENDAR()	EXCEPT()	RELATEDTABLE()	UNION()	

Figure 8.7 – Most common table functions in DAX

We can create calculated tables in many scenarios, especially when the business needs to reuse the data contained in a calculated table in the future. Calculated tables become handy in many scenarios, such as creating summary tables based on the existing measures or creating `Date` or `Time` tables (when they do not exist in the data source). The data that's held in calculated tables is already available in the data model, so refreshing the calculated tables doesn't take much time.

Data refresh is not even available from the context menu when we right-click on a calculated table. The data is automatically populated when we refresh the underlying source tables. However, calculated tables, just like any other physical tables, consume some storage space. Hence, after creating calculated tables, the data model's size in Power BI Desktop increases. Consequently, when we publish the model to the Power BI service, the dataset's size increases.

Creating a new calculated table is not currently available via the **Model** view in Power BI Desktop. However, we can find the **Table tools** tab from the ribbon in the **Data** view of Power BI Desktop, as shown in the following screenshot:

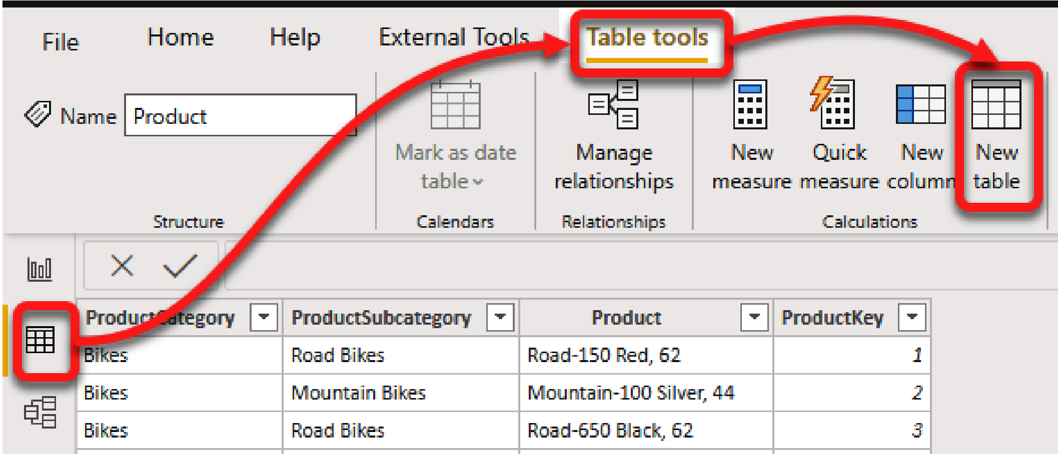


Figure 8.8 – Creating a new calculated table from the Data view in Power BI Desktop

We can also create a new calculated table from the **Modeling** tab of the **Report** view, as shown in the following screenshot:

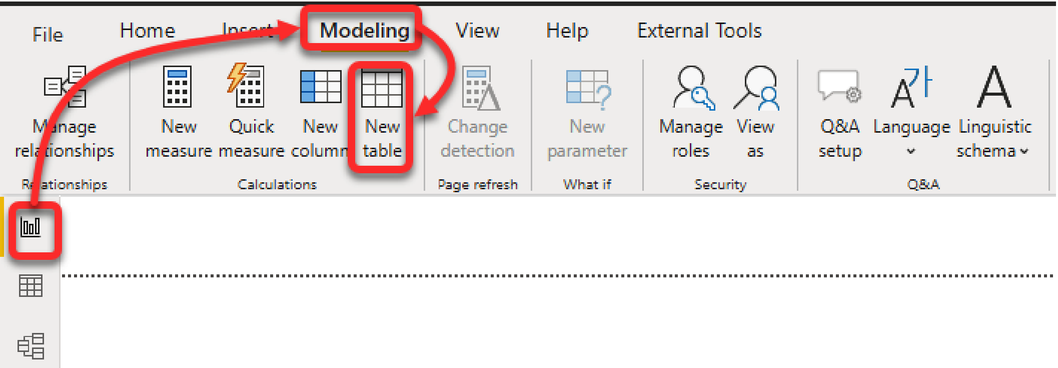


Figure 8.9 – Creating a new calculated table from the Report view in Power BI Desktop

Let's look at the `Chapter 8, Data Modelling and Star Schema.pbix` sample file. The data model looks as follows:

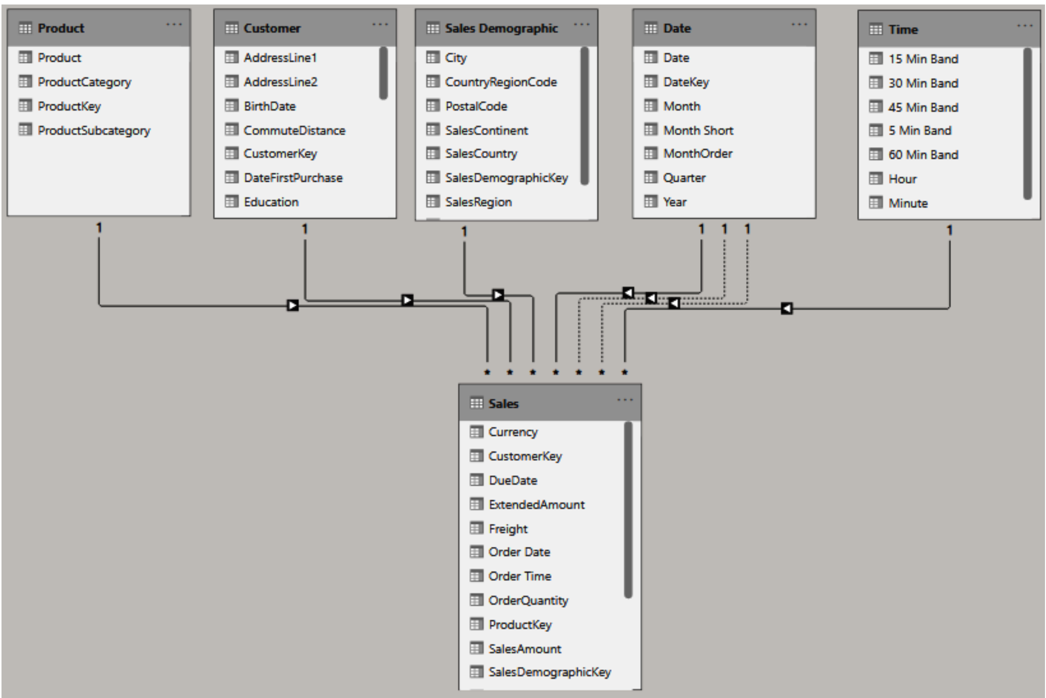


Figure 8.10 – Sample Sales data model

With the preceding data model, we can analyze sales for many different business entities, such as analyzing `Sales` by `Product`, by `Customer`, by `Sales Demographics`, by `Date`, and by `Time`. Now, the business has a new requirement: getting the number of products sold to all customers with a yearly income greater than $100,000 at the year-month level. We can show these requirements in the data visualization layer. Still, the business would like to have the data as a summary table in order to analyze the data at higher levels, and then reuse the data in different data visualizations in the future. The important points to note in this scenario are as follows:

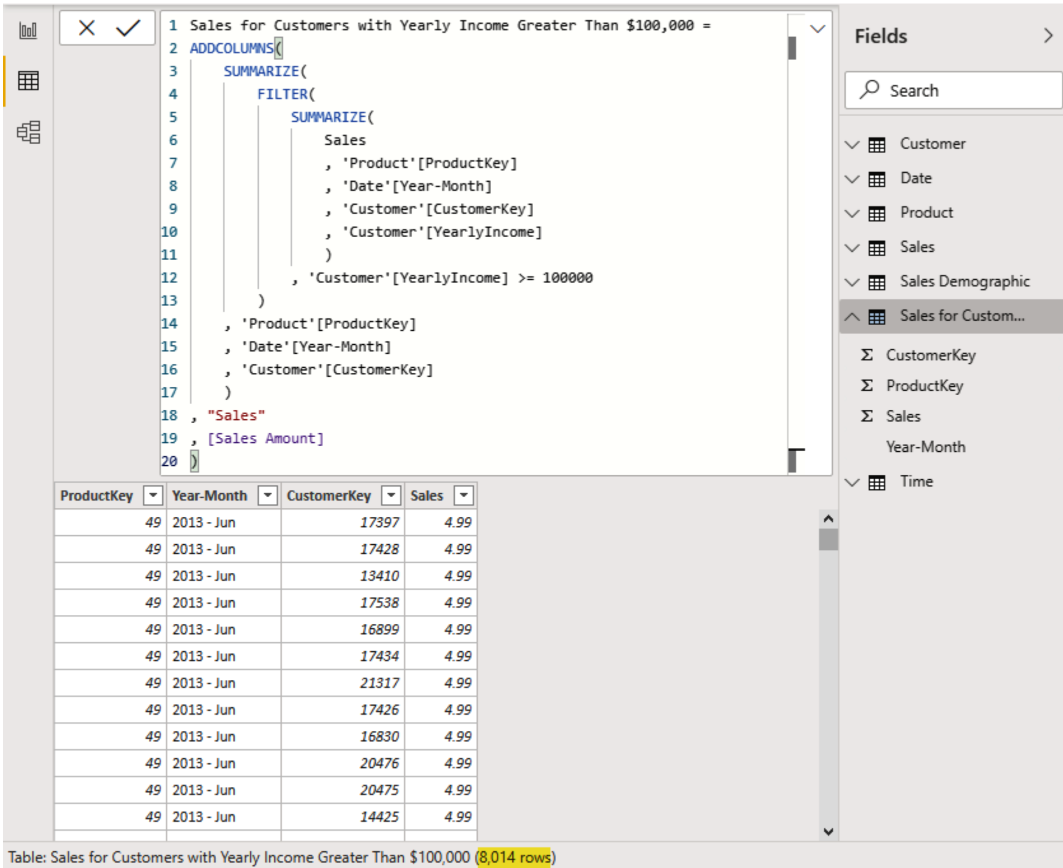
- The business is only after the products sold. Therefore, for each product from the `Product` table, there must be a `Sales Amount` within the `Sales` table.
- From the sold products, the business is only interested in the ones that the customers with a yearly income greater than `$100,000` had bought.

- The results must be at the year-month level.
- The business requires the results as summary data so that they can analyze the data at a higher granularity.

We can use the following DAX expression to cater to that:

```
Sales for Customers with Yearly Income Greater Than $100,000 =
ADDCOLUMNS(
    SUMMARIZE(
        FILTER(
            SUMMARIZE(
                Sales
                , 'Product'[ProductKey]
                , 'Date'[Year-Month]
                , 'Customer'[CustomerKey]
                , 'Customer'[YearlyIncome]
                )
            , 'Customer'[YearlyIncome] >= 100000
        )
    , 'Product'[ProductKey]
    , 'Date'[Year-Month]
    , 'Customer'[CustomerKey]
    )
, "Sales"
, [Sales Amount]
)
```

The following screenshot shows the results:




```
1 Sales for Customers with Yearly Income Greater Than $100,000 =
2 ADDCOLUMNS(
3     SUMMARIZE(
4         FILTER(
5             SUMMARIZE(
6                 Sales
7                 , 'Product'[ProductKey]
8                 , 'Date'[Year-Month]
9                 , 'Customer'[CustomerKey]
10                , 'Customer'[YearlyIncome]
11                )
12            , 'Customer'[YearlyIncome] >= 100000
13        )
14    , 'Product'[ProductKey]
15    , 'Date'[Year-Month]
16    , 'Customer'[CustomerKey]
17    )
18 , "Sales"
19 , [Sales Amount]
20 )
```

ProductKey	Year-Month	CustomerKey	Sales
49	2013 - Jun	17397	4.99
49	2013 - Jun	17428	4.99
49	2013 - Jun	13410	4.99
49	2013 - Jun	17538	4.99
49	2013 - Jun	16899	4.99
49	2013 - Jun	17434	4.99
49	2013 - Jun	21317	4.99
49	2013 - Jun	17426	4.99
49	2013 - Jun	16830	4.99
49	2013 - Jun	20476	4.99
49	2013 - Jun	20475	4.99
49	2013 - Jun	14425	4.99

Figure 8.11 – Calculated table showing sold products to customers with yearly income greater than $100,000

As the preceding screenshot shows, the calculated table has four columns: `ProductKey`, `Year-month`, `CustomerKey`, and `Sales`.

When we create a calculated table derived from other tables in the data model, we might get a circular dependency error if we were not careful about the functions we used to create the calculated table. In that case, we cannot create any relationships between the calculated table and any tables it is derived from, unless we change our choice of functions in the DAX expressions.

> **Note**
>
> It is advised not to use calculated tables for any data transformation activities. We always tend to move the data transformation logics to the source system when possible. Otherwise, we take care of the transformation in the Power Query layer.

Now, we can create the relationships between the new calculated table, the `Product` table, and the `Customer` table, as shown in the following screenshot:

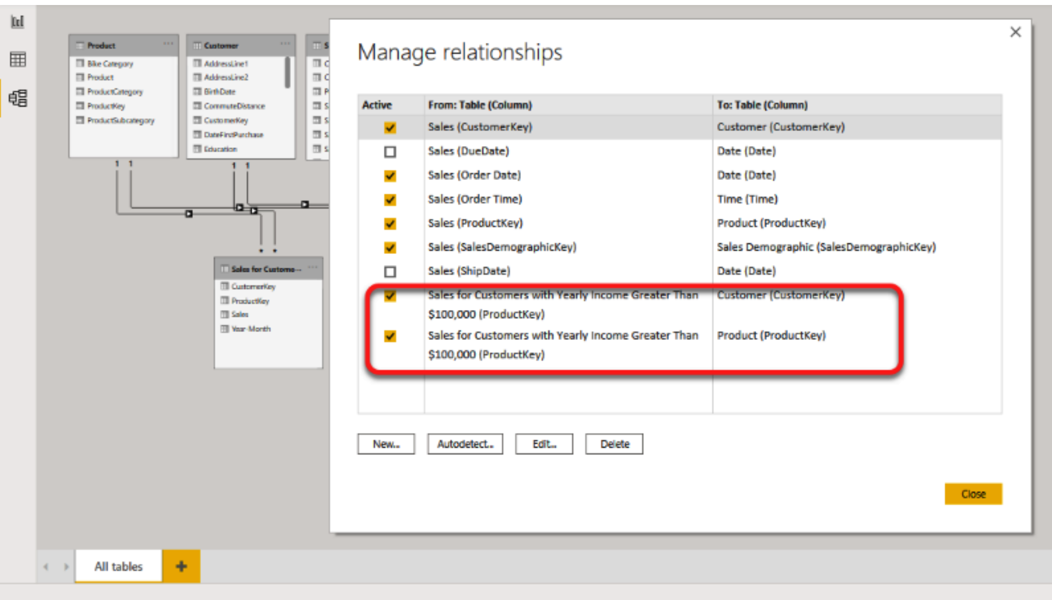


Figure 8.12 – Creating relationships between the calculated table and the dimension tables

The business can now analyze the sales data based on the attributes of both the `Product` and `Customer` tables, but only for the customers with an annual income greater than $100,000. This exercise can be an exciting analysis if you wish to target a specific group of customers and understand what products they spend their money on.

Understanding fields

Fields in Power BI include columns and measures. When we talk about fields, we are generally talking about something that applies to both columns and measures. For instance, when we talk about fields' data types, we refer to the correct data types for both columns and measures. The **Fields** term is used within Power BI Desktop in different views, so there is a **Fields** pane in the **Report** view, the **Data** view, and the **Model** view.

Data types

When we import data into the model, the model converts that data, in columns, into one of the **Tabular Model** data types. When we then use the model data in our calculations, the data is converted into a **DAX** data type for the duration of running the calculation. The model data types are different from Power Query data types. For instance, in Power Query, we have `DateTimeZone`. However, the `DateTimeZone` data type does not exist in the data model, so it converts into `DateTime` when it loads into the model. The following table shows the different data types supported in the model, as well as DAX:

Data type in model	Data type in DAX	Description
Whole Number	Int64	Integer numbers between -9,223,372,036,854,775,808 (-2^63) and 9,223,372,036,854,775,807 (2^63-1).
Decimal Number	Double	Real numbers between negative values between -1.79E+308 and -2.23E-308 and positive values between 2.23E-308 and 1.79E+308. The number of digits after the decimal point is limited to 17 digits.
True/false	Boolean	Either a True or False value.
Text	String	A Unicode character data string in text format.
Date	DateTime	Shows the date part of DateTime values in the accepted date format. The default starting date in DAX is December 30, 1899, however the official start date considered in calculations is March 1, 1900.
Time	DateTime	Shows the time part of a DateTime values. If the date part is not specified in the source, then December 1, 1899 is considered as the date part.
Date/time	DateTime	Shows full DateTime values in accepted date/time format.
Fixed Decimal	Decimal	Currency data type allowing values between -922,337,203,685,477.5808 to 922,337,203,685,477.5807 with a fixed four digits after the decimal point.
N/A	Blank	A blank is a data type in DAX similar to null in SQL. We can create a blank by using the BLANK() function in DAX. To test for blanks we can use the ISBLANK() function.

Figure 8.13 – Data types in DAX

In Power BI Desktop, the model data types are visible under the **Column tools** tab from the **Data** view or the **Report** view. The following screenshot shows the data types from the **Column tools** tab within the **Data** view:

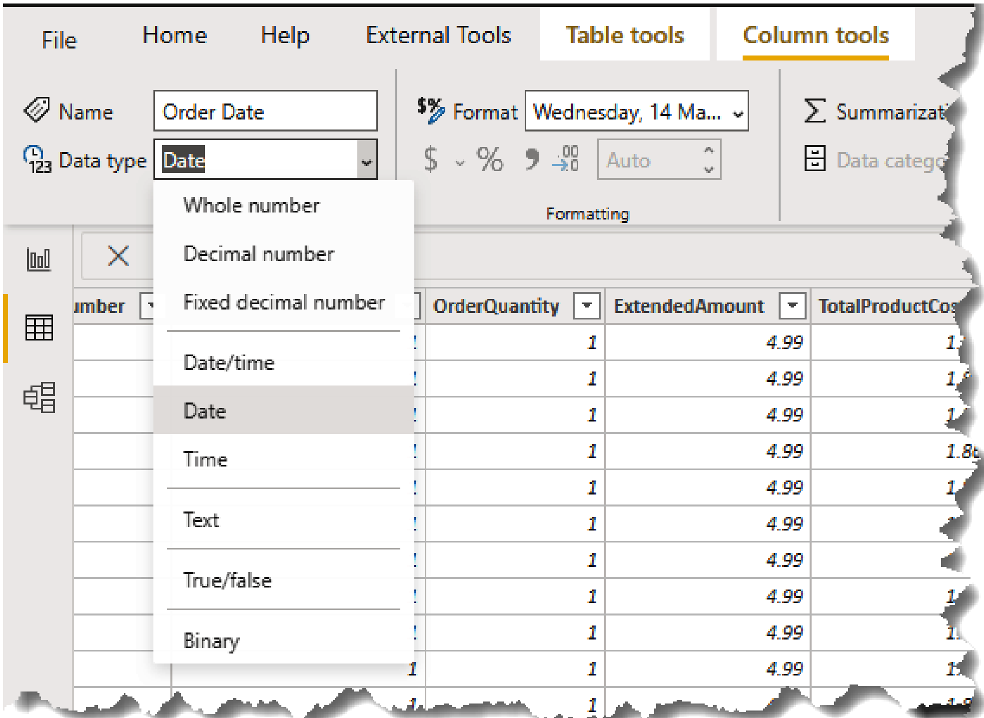


Figure 8.14 – Data types in Power BI Desktop

Notes

While the `Binary` data type is available in the **Data types** dropdown, Power BI does not support it in the data model. Hence, if we try to convert a column's data type into binary, we get an error message.

Power BI Desktop eliminates binary columns by default. Therefore, we should remove them in **Power Query Editor**.

Power Query supports the `Binary` type; therefore, it is best to convert the binary values into one of the supported tabular model data types.

We can implicitly define the data types for columns only. A measure's data types are automatically determined by the functions we use in DAX expressions.

When we use table functions in DAX, the result is a `Table` data type. We use this data type to create either virtual tables or calculated tables.

While Power Query supports the `DateTimeZone` type, it converts into `DateTime` without adjusting the zone when it loads into the data model. Therefore, we must take care of the zone adjustments in **Power Query Editor** before loading the data.

Power Query supports the `Duration` type, but when the data loads into the model, the duration values are converted into `Decimal` values.

We can `Add` and `Subtract` numeric values to/from `DateTime` values without raising any errors; for instance, `DATE(2010, 1, 1) + 0.04167 = 1/01/2010 1:00:00 AM`.

Custom formatting

Formatting is the way we garnish values. We only format values to make them more user-friendly and readable. Changing the value's formatting does not change its data type. Hence, it does not affect memory consumption, nor performance. Some data types support custom formatting. The following table shows custom formatting for various supporting data types:

Datatype	Symbol	Meaning	Example
DateTime	d	Day number of the month without leading zero.	FORMAT(CONVERT("2020/12/31 3:45:55", DATETIME) , "dddd, mmm d yyyy h:mm:ss am/pm") returns Thursday, Dec 31 2020 3:45:55 A.M.
	dd	Day number of the month with leading zero.	
	w	Week number of the month without leading zero.	
	ww	Week number of the month with leading zero.	
	m	Month number of the year without leading zero.	
	mm	Month number of the year with leading zero.	
	mmm	Abbreviation of the month name.	
	mmmm	Full mouth name.	
	yy	Last two digit of year number.	
	yyyy	Four-digit year number.	
	h	Hour number of the day without leading zero (24-hour format without appended am/pm, 12-hour format with appended am/pm).	
	hh	Hour number of day with leading zero (24-hour format without appended am/pm, 12-hour format with appended am/pm).	
	n	Minute number of the hour without leading zero.	
	nn	Minute number of the hour with leading zero.	
	m	Minute number of the hour without leading zero if comes after h or hh.	
	mm	Minute number of the hour with leading zero if comes after h or hh.	
	s	Second number of the hour without leading zero.	
	ss	Second number of the hour without leading zero.	
	A.M./P.M.	Force the output to display the time in 12-hour format.	
Numeric	0	Displays a digit or zero. If the number has fewer digits as the number of zeros in the format string, it puts leading or trailing zeros. If the number has more digits, it rounds the number to as the number of zeros in the format string.	FORMAT(0, "0,0") returns 00 FORMAT(0.1153, "0.00") returns 0.12 FORMAT(35.1113, "0,00.00") returns 035.11
	#	It works like the 0, but it displays a digit or nothing.	FORMAT(0, "#,#") returns nothing FORMAT(0.1153, "#.##") returns .12 FORMAT(35.1113, "#,##.##") returns 35.11
	%	Displays the value in percentage by multiplying it by 100 and shows the % symbol where it appears in the format string.	FORMAT(0.351156, "#,##.##0 %") returns 35.116 %
	$	Displays the dollar sign where it appears in the format string.	FORMAT(-35.1156, "$#,##.##0") returns $35.116
	;	Section separator in the format string. There are four sections allowed in a format string if the semicolon mentioned. If the format string has one section only then the formatting applies to all values, otherwise the first section applies to positive values, the second to negative values, the third to zeros, and the fourth to Null values.	FORMAT(0.015, "$#,#.#0;;\Z\e\r\o") returns $.02 FORMAT(-0.015, "$#,0.#0;($#,0.#0);\Z\e\r\o") returns ($0.02) FORMAT(-0.015, "$#,0.0;($#,0.0);\Z\e\r\o") returns Zero
	()+-	Displays literal characters. This is normally used to show positive and negative values.	FORMAT(-0.05, "$#,0.0;($#,0.0)") returns ($0.1)
	\	The backslash is the escape character in the format string. So, it displays the next character as literal. To show the backslash itself we use \\ (backslash, backslash).	FORMAT(-0.01, "$#,0.0;($#,0.0);\n\i\l") returns nil

Figure 8.15 – Custom formatting options

Columns

In Power BI, tables are created by a set of columns. These columns are either physical columns that come directly from the data source or are made by the developer or Power BI Desktop itself. In this section, we will look more closely at the different types of columns and column properties within the **Model** view in Power BI Desktop.

Calculated columns

Calculated columns are the columns that do not live in the data source. We can create a calculated column in the data model using DAX expressions.

Power BI uses the xVelocity engine, which is Microsoft's proprietary in-memory data processing engine. The xVelocity engine uses ColumnStore indexing technology, which highly compresses the column's data based on the data's cardinality within that column. When we refresh a table's data, the engine compresses all columns' data and loads it into memory at once. This process applies to the physical columns that already exist in tables.

Power BI computes the calculated columns after loading the table (which the calculated columns belong to) into the model. When we refresh a table, the new data loads into the model, so the calculated columns' values are no longer valid. Therefore, the engine must recompute all the calculated columns. Moreover, the engine sequentially computes the calculated columns in a table. Thus, the calculated columns are not optimized and compressed as well as the physical columns are.

Grouping and binning columns

In Power BI Desktop, we can create a grouping column on top of any columns. However, we can only create binning columns for the columns with numeric data types. Grouping and binning are two ways to group the values of a column manually. Grouping and binning come in handy when we need to group our data.

The grouping and binning features are not currently available in the **Model** view, so to create a new grouping or binning column, we need to switch to either the **Report** view or the **Data** view. Then, we must right-click on the desired column and select the **New group** option from the context menu. For instance, if we look at the `Product` table from our sample file, we can see, under the `ProductCategory` column, that the categories are `Accessories`, `Bikes`, and `Clothing`. Now, the business only needs to analyze their sales over two categories; that is, `Bikes` and `Other`. There are many ways to answer this query, such as by creating a new column within Power Query Editor. However, we want to look at the grouping feature by going through the following steps:

1. Click the **Data** view tab.
2. Right-click the `ProductCategory` column.
3. Click **New group**.
4. Enter `Bike Category` for **Name**.
5. Click **Bikes** from the **Ungrouped values** list.
6. Click the **Group** button.
7. Tick the **Include Other group** option.
8. Click **OK**.

The following screenshot illustrates the preceding steps:

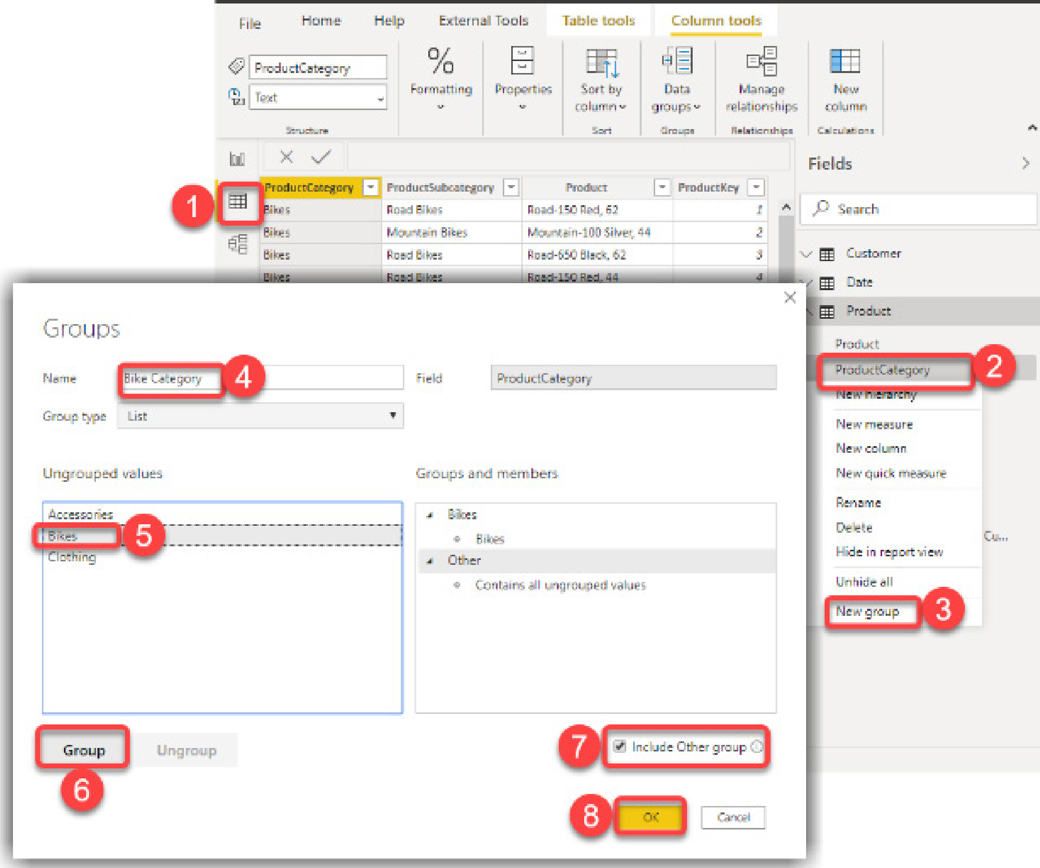


Figure 8.16 – Creating a new Data Group

The preceding steps create a new **Data Group** column, as shown in the following screenshot:

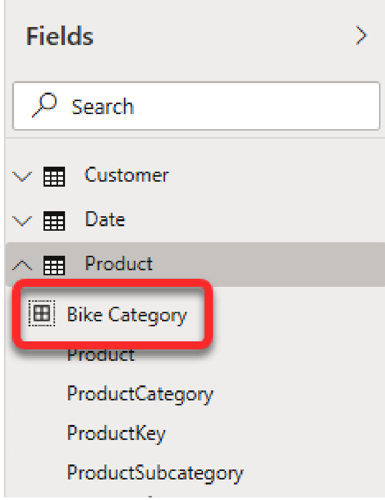


Figure 8.17 – A new Data Group has been created

We can use this new **Data Group** in our data visualizations just like any other columns.

Just like how we created a **Data Group** using the grouping, we can use the binning option for numeric columns.

An excellent example of binning the data in our sample is when the business must group the `SalesAmount` values from the `Sales` table when the bin (group) size for `SalesAmount` is `$1,000`.

The following steps show how to bin the values of `SalesAmount`:

1. Right-click the `SalesAmount` column from the `Sales` table.
2. Type in `Sales Amount Bins` for **Name**.
3. Stick to **Bin** via the **Group type** dropdown.
4. Make sure **Bin Type** is set to **Size of bins**.
5. Enter `1000` for **Bin size**.
6. Click **OK**:

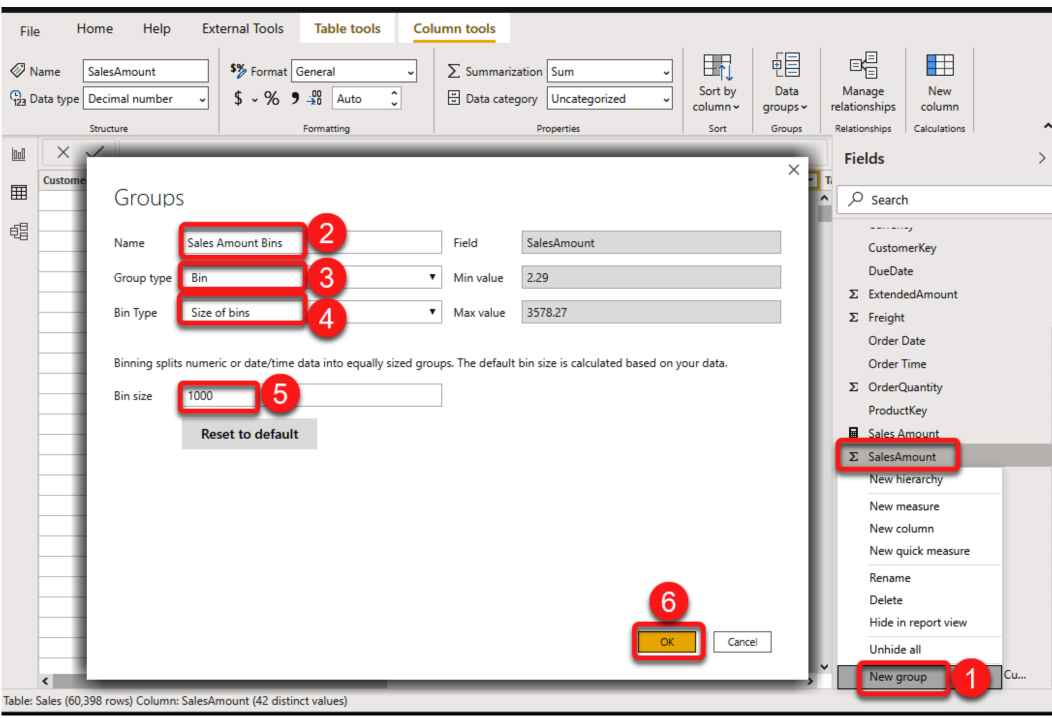


Figure 8.18 – Creating Data Groups by binning the data

The preceding steps create a new numeric **Data Group** column with no summarization.

The following screenshot shows a simple visualization showing how grouping and binning can help us create storytelling visuals:

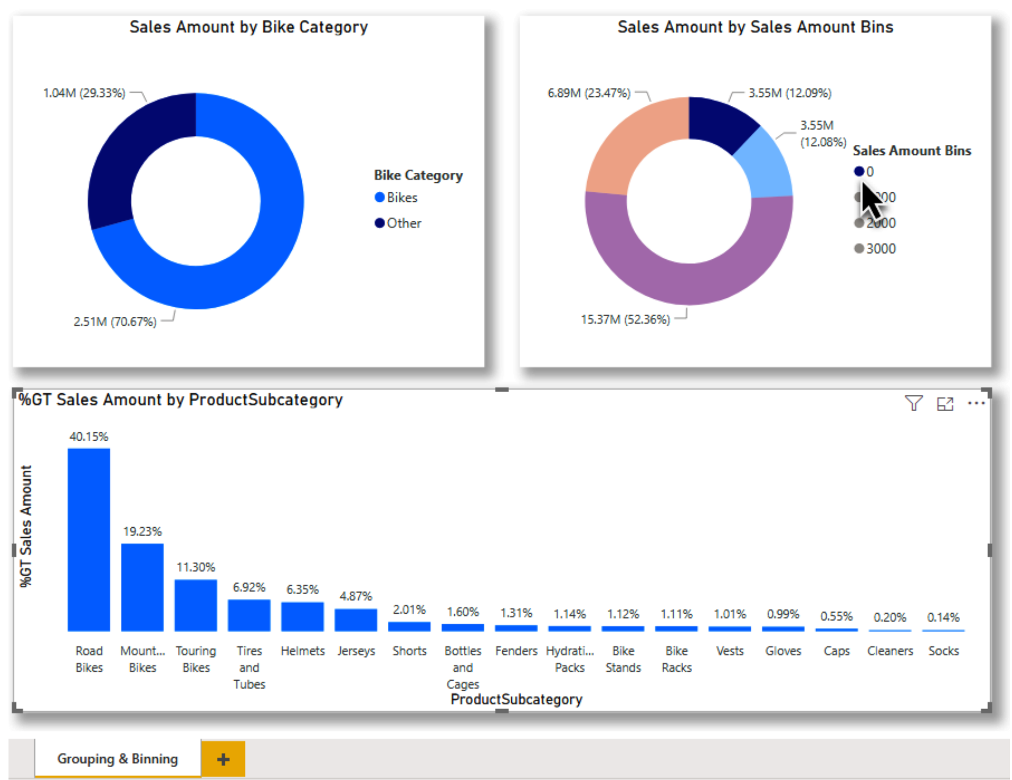


Figure 8.19 – Using Data Groups for data visualization

In the preceding screenshot, the user was interested in seeing sales analysis for all sales items smaller than **$1,000**. She clicked on the **0** bin from the right doughnut chart. Here is how this simple activity reveals interesting information about our sales:

- In the right doughnut chart, we can see that **12.09%** of our total sales are from items cheaper than **$1,000**.
- The left doughnut chart shows that from that **12.09%** of total sales, **70.67%** comes from selling bikes, while **29.33%** comes from other categories.
- The column chart at the bottom of the report also reveals some exciting information. For example, we have **Road Bikes**, **Mountain Bikes**, and **Touring Bikes** that are cheaper than **$1,000**. From the under **$1,000** items (**12.09%** of our total sales), our buyers bought under **$1,000** worth of road bikes, which is **40.15%** of our under **$1,000** deals.

Column properties

The column properties are available within the **Model** view. We can see and change column properties by clicking on a column in the **Model** view via the **Properties** pane. Depending on the data type of the selected column, we will see slightly different properties in the properties pane. For instance, the following screenshot shows the column properties of the `Date` column from the `Date` table:

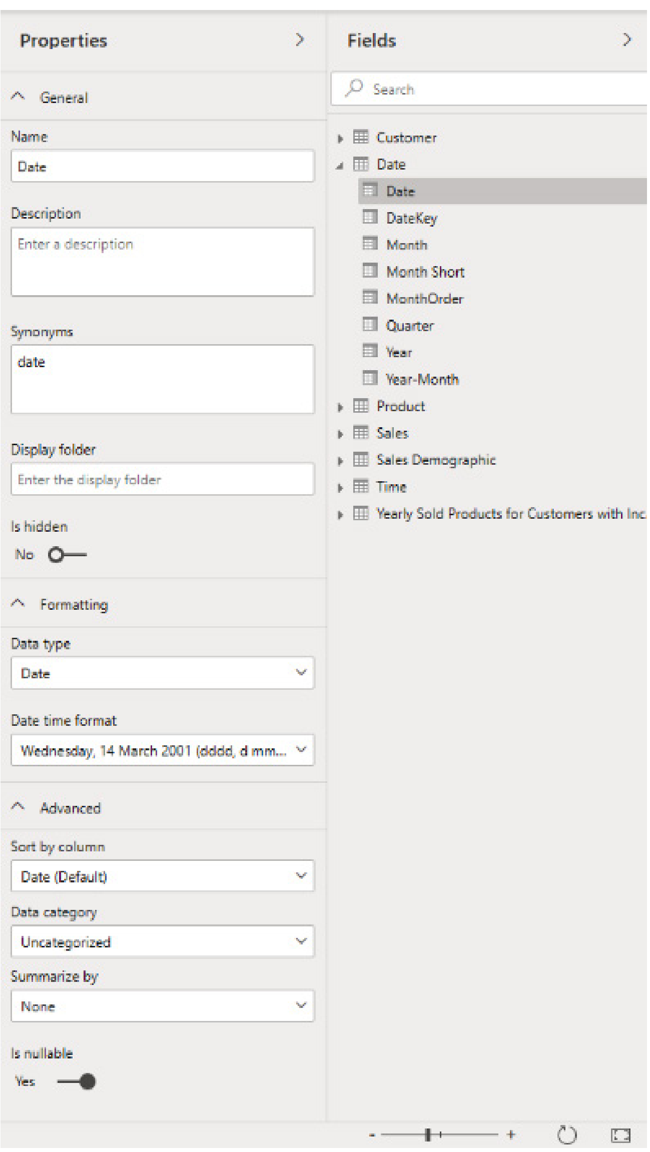


Figure 8.20 – Column properties in the Model view

We can change a column's properties in bulk by pressing the *Ctrl* key on our keyboard and then clicking the columns via the **Properties** pane. Then, we can change the properties of the selected columns in bulk from the **Properties** pane.

The column properties include the following:

- **General**: This includes generic column properties, such as the following:

  a. **Name**: Contains the name of the column. We can rename a column by changing the value of its **Name** property.

  b. **Description**: Here, we can write a brief description of the column. The description of a column shows up when we hover over the column from the **Fields** pane in Power BI Desktop, as well as in the Power BI service when the report is in **Edit** mode.

  c. **Synonyms**: We can enter some synonyms to help Q&A show more relevant information.

  d. **Display folder**: The display folder is also available for measures. We can organize our model by grouping relevant fields into display folders.

We can group all the key columns in a folder by following these steps:

1. Search for `key` in the search box in the **Model** view.
2. Select multiple columns by pressing the *Ctrl* key on your keyboard and clicking each column.
3. Enter **Key Columns** in the **Display folder** box of the **Properties** pane.

The following screenshot illustrates the preceding steps:

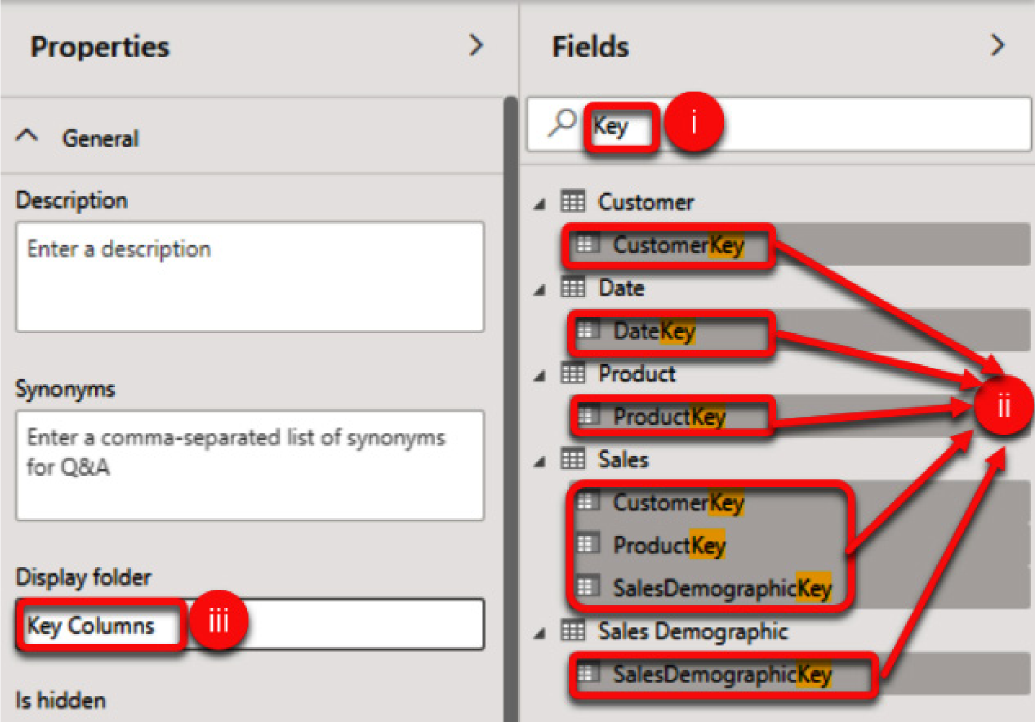


Figure 8.21 – Grouping columns with the Display folder option

The preceding steps result in creating a **Key Columns** display folder in each table. The following screenshot shows the new display folder:

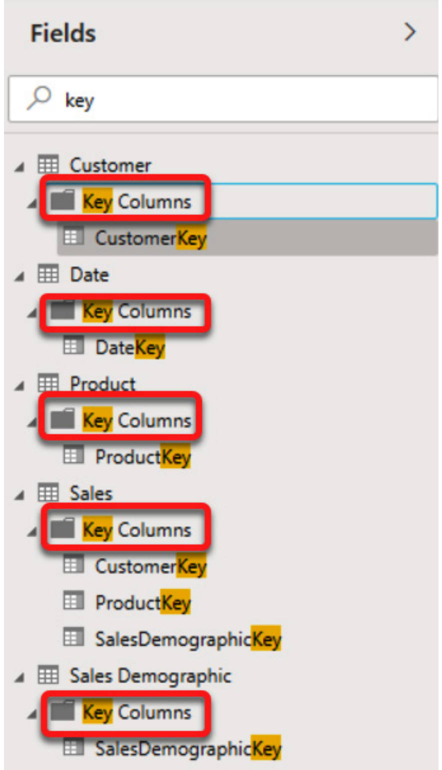


Figure 8.22 – Display folders created in tables

We can also create nested folders by following the **Parent Folder\Child Folder** pattern. The following screenshot shows an example of creating nested display folders:

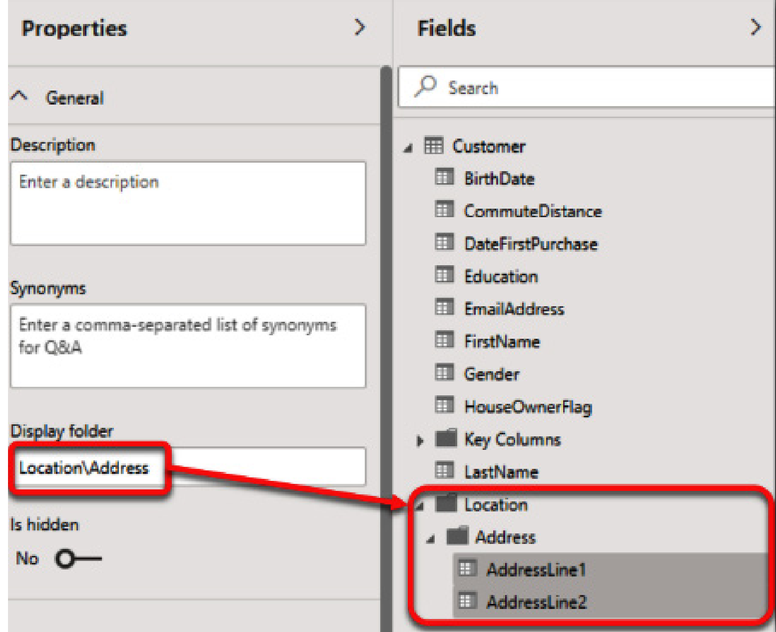


Figure 8.23 – Creating nested display folders

Here are some properties of **Fields**:

- **Is hidden**: We can hide a field by toggling this property to **Yes** or **No**.
- **Formatting**: This includes properties related to formatting a column. The formatting section varied based on the data type of the selected column, as follows:

  a. **Data type**: We can change the data type of the selected columns by selecting a desired data type from the dropdown list.

  b. **Format**: The format varies based on the data type, so we can pick a format that best suits our use case. We looked at custom formatting in detail in this chapter in the *Custom formatting* section.

- **Advanced**: This includes more advanced column properties, as follows:

  a. **Sort by column**: We can set this property to sort a column by another column; for example, we can sort the `Month` column by the `MonthOrder` column from the `Date` table.

  b. **Data category**: We can state the data category for the selected column by setting this property. This property tells Power BI how to treat the values in data visualization. For instance, we can set this property to `City` for the `City` column from the `Sales Demographic` table.

c. **Summarize by**: This shows how we want to implicitly aggregate a column when we use it in our visuals. For instance, we might want to aggregate `ProductKey` to `COUNT` or `ProductKeys`. We can do this in DAX, but this is another way to do so. We can set this property to `COUNT` for the `ProductKey` column from the `Product` table. When we use it in our visuals, it automatically shows the count of `ProductKeys`. The following screenshot shows the card visual that we directly used for our `ProductKey` after setting its **Summarize by** property to `COUNT`:

Figure 8.24 – Count of the ProductKey column by setting its Summarize by property

- **Is nullable**: We can set this property for the columns that are not supposed to have any null values, such as our primary keys. However, bear in mind that if we toggle this property to **No**, then if, in the future, there is a null value in the column, we get an error message while refreshing the data.

Hierarchies

Hierarchies are abstract objects in our model that show how different columns relate to each other from a hierarchical viewpoint. We can create hierarchies in the data model from the **Report** view, from the **Data** view, or the **Model** view. To create a **Calendar hierarchy** in the `Date` table from the **Model** view, follow these steps:

1. Right-click the **Year** column.
2. Select the **Create hierarchy** option from the context menu.
3. From the **Properties** pane, change the **Name** value of the hierarchy to **Calendar Hierarchy**.
4. From the **Hierarchy** section, select the **Month** and **Date** columns from the dropdown list.
5. Click the **Apply Level Changes** button.

The following figure shows the preceding steps:

Figure 8.25 – Creating a hierarchy from the Model view

As shown in the preceding image, some other properties are available for a hierarchy that are very similar to column hierarchies, so we will skip explaining them again.

Measures

In data modeling, measures are calculations we create to help us with our data analysis. The results of these equations of measure always change when we interact with the measures. This interaction with the measures can happen from the data visualization layer, when we use the measures in visuals, or within the data model, when we use the measure to create other tabular objects such as calculated tables. We can create measures either from the **Report** view or from the **Data** view. Once we've created these measures, we can set measures properties from the **Model** view. The properties for these measures are very similar to the column properties, so we will skip explaining them again. In Power BI, there are two types of measures: implicit measures and explicit measures. Let's look at them in more detail.

Implicit measures

Implicit measures are abstract measures that are created when we use a column in a visual. Power BI automatically detects these implicit measures based on the column's data type when it is a numeric data type. We can quickly recognize these implicit measures by their (Σ) icon from the **Fields** pane. If we set the **Summarize by** property of a numeric column (implicit measure), then we can use that column in a visual. This visual, by default, uses the selected aggregation within the **Summarize by** property. This means that if we set the **Summarize by** property of a column to `COUNT` when we use the column in a visual, the visual automatically uses the count of that column as a measure. In other words, implicit measures are the measures that we never create using DAX expressions. They are generated when we use them directly in our visuals. The following screenshot shows the `Year` column being detected as an implicit measure. Therefore, when we use it on a table visual, the table automatically calculates the `SUM` value of `Year`, which is incorrect:

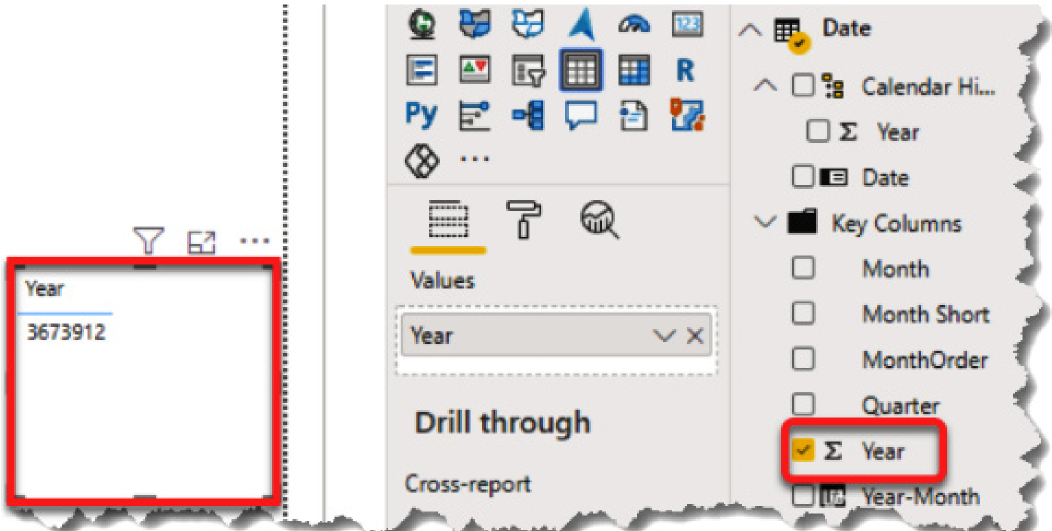


Figure 8.26 – The Year column from the Date table detected as an implicit measure

While using implicit measures is very easy, we do not recommend using this feature. There are many reasons to avoid using implicit measures, some of which are as follows:

- In many cases, the columns that are detected as implicit measures do not contain additive values; therefore, a summation of those values is incorrect, such as in the **Year** example that we looked at previously.
- Implicit measures are not reusable. They are created in the visualization layer and can only be used on the visuals.

- We do not have any control over the underlying expressions the visuals create.
- They are confusing and expensive to maintain.
- They are not compatible with some advanced data modeling techniques such as **Calculation Groups**.

There are two ways we can disable implicit measures. The first method is to set the **Summarize by** property of the columns to **Don't summarize**. We can quickly set this via the **Model** view by performing the following steps:

1. Click on a table either from the **Fields** pane or the **Model** view.
2. Press *Ctrl* + *A* on your keyboard.
3. Right-click on a table.
4. Click the **Select columns** option from the context menu.
5. Expand the **Advanced** section of the **Properties** pane.
6. Select **None** from the dropdown list for the **Summarize by** property.

The following screenshot illustrates the preceding steps:

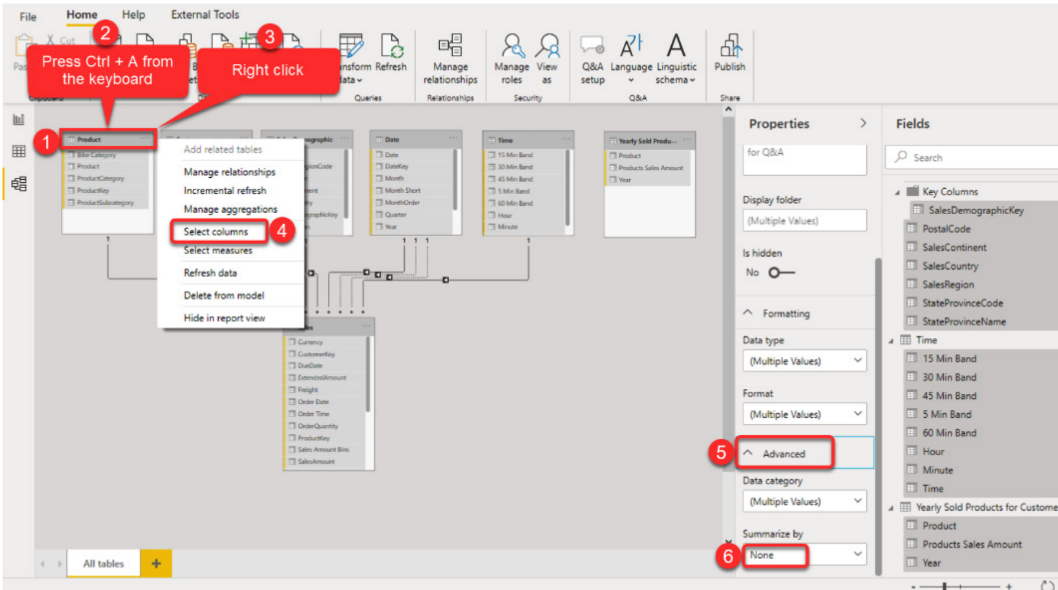


Figure 8.27 – Setting the Summarized by property to Don't summarize for all columns across the model

The second method is to disable the implicit measures using **Tabular Editor** across the entire model. This method prevents the model from detecting implicit measures; therefore, if we add more tables to our data model in the future, the model will not detect any more implicit measures. **Tabular Editor** is an **external tool** that's widely used by the community. At the time of writing this book, this option is not available directly via Power BI Desktop. The following steps explain how to disable implicit measures from the Tabular Editor:

1. In **Power BI Desktop**, click the **External Tools** tab from the ribbon.
2. If you installed the latest version of **Tabular Editor**, it will appear in **External Tool**. Click **Tabular Editor**.
3. In **Tabular Editor**, click **Model**.
4. Set the values of the **Discourage Implicit Measures** option to **True**.
5. **Save** the changes you made to the model:

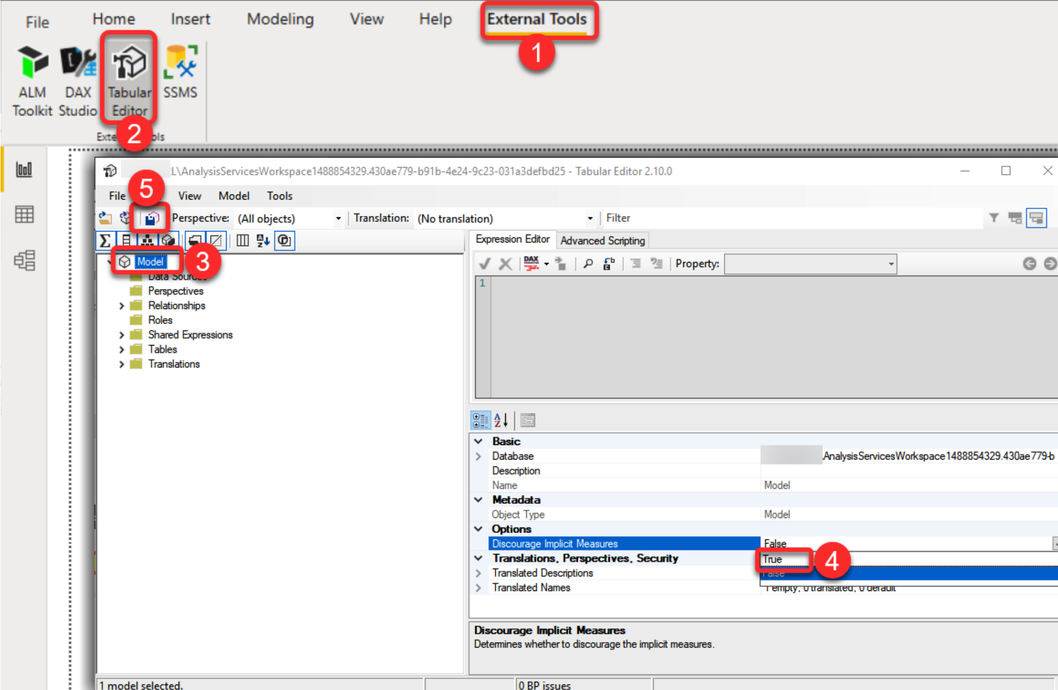


Figure 8.28 – Discouraging implicit measures for the entire model in Tabular Editor

Now that we've saved the changes back to Power BI Desktop, close **Tabular Editor**, then refresh the model in Power BI Desktop to apply the changes properly.

Explicit measures

Explicit measures are the measures we create using DAX functions. There are endless use cases for measures. We can create measures to answer easy summation, to complex time intelligence and running total questions. We can create textual measures to make visual titles more dynamic. We can also create measures that will be used in conditional formatting to make the formatting more dynamic.

Textual measures

The concept of textual measures is helpful for solving many data visualization scenarios. Here is a common scenario: the business needs to visualize the sales amount by product in a column chart. The color of the columns within the chart must change to red if the sales amount for that particular data point is below the overall average sales for products.

To solve this challenge, we need to create a textual measure such as the following:

```
Sales by Product Column Chart Colour =
     var avgSales = AVERAGEX(
                     ALL('Product'[Product])
                     , [Sales Amount]
                 )
     return
     IF([Sales Amount] < avgSales, "Red")
```

Now, we must put a column chart on a report page and follow these steps:

1. Put the `Product` column from the `Product` table on **Axis**.
2. Put the **Sales Amount** measure on **Values**.
3. Click the **Format** tab from the **Visualizations** pane.
4. **Expand** the **Data colors** dropdown.
5. Click the **fx** button.
6. Select **Field value** from the **Format by** dropdown.
7. Select the **Sales by Product Column Chart Color** measure.
8. Click **OK**.

The following figure illustrates the preceding steps:

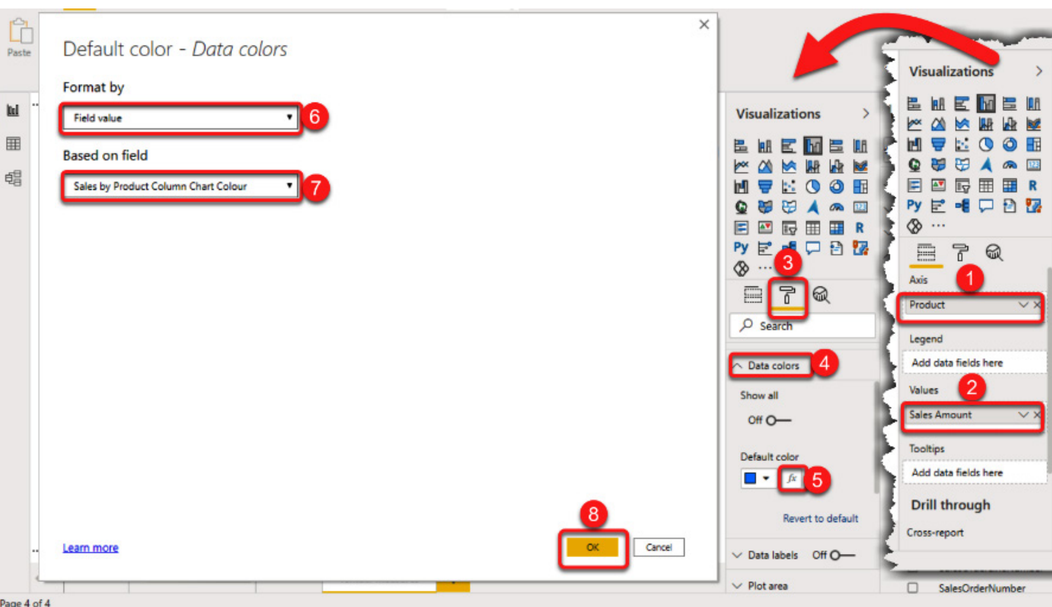


Figure 8.29 – Using a textual measure to set the colors in a column chart dynamically

The following screenshot shows the results:

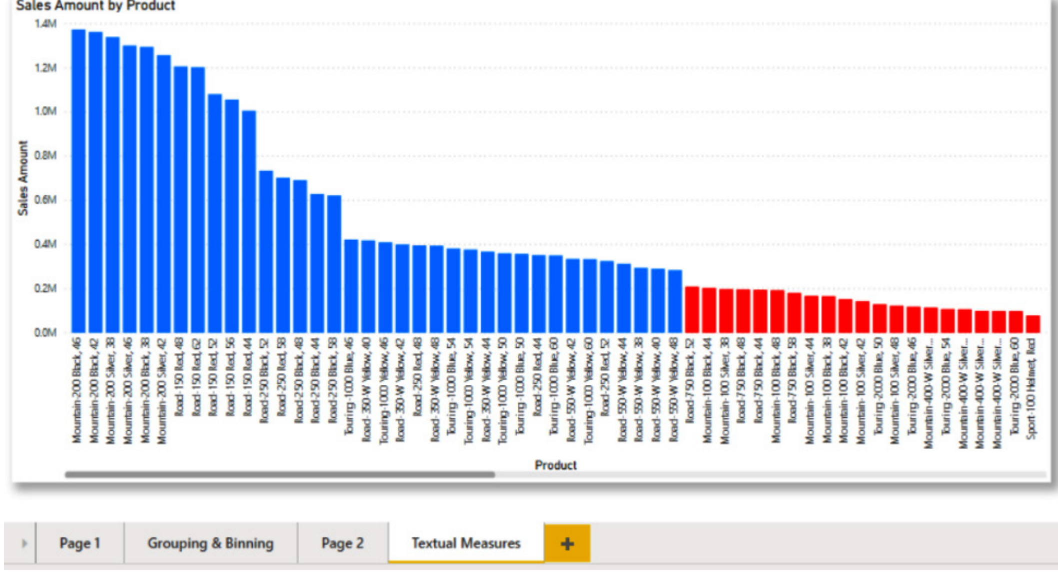


Figure 8.30 – Conditionally formatted column chart using textual measures

Using relationships

A relationship, when modeling relational data, describes the connection between two tables. For instance, there is a relationship between the `Customer` table and the `Sales` table in our example. A customer can have multiple sales transactions in the `Sales` table. To create a relationship between the `Customer` and `Sales` tables, we must link `CustomerKey` from the `Customer` table to `CustomerKey` from the `Sales` table. This linkage enables Power BI to understand that each row of data in the `Customer` table can have one or more related rows in the `Sales` table.

To create relationships between tables in Power BI Desktop, we can either use the **Model** view to drag a column from a table and drop it to the relevant column from the other table, or we click the **Manage relationships** button from the ribbon. The **Manage relationships** button appears in several places in the ribbon. The following screenshot shows the **Manage relationship** window:

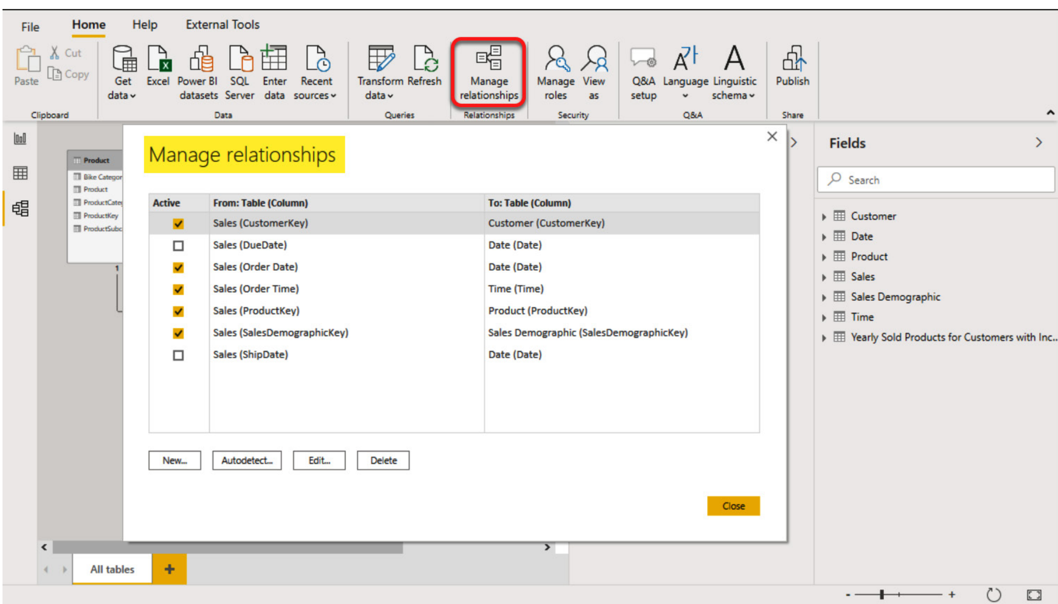


Figure 8.31 – Manage relationships window in Power BI Desktop

When we create a relationship between two tables, we can visually see the relationship in the **Model** view, so the two tables are linked by either a solid line or a dotted line. A solid line represents an **active relationship**, while a dotted line represents an **inactive relationship**. The following screenshot shows how the `Date` and `Sales` tables relate to each other:

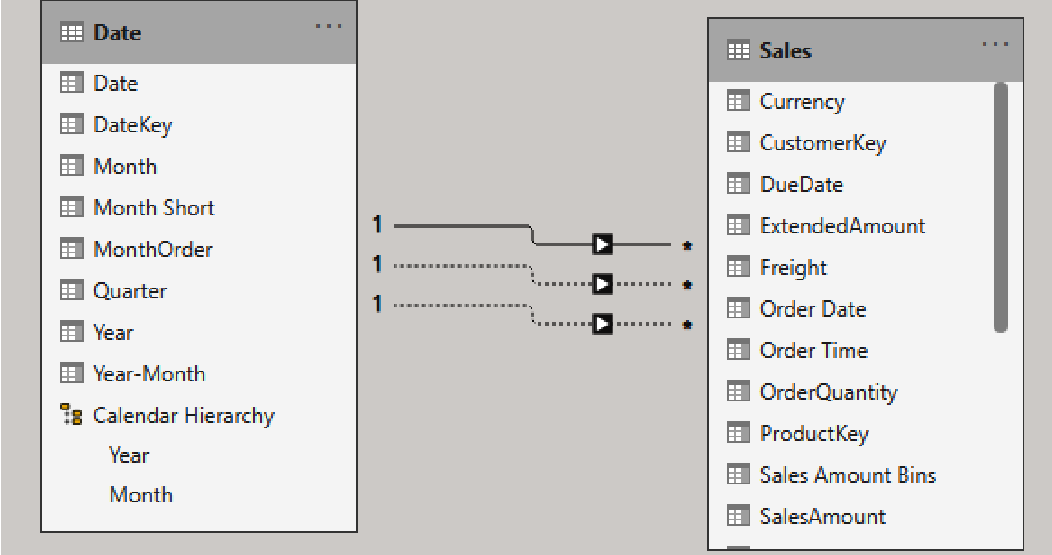


Figure 8.32 – Active and inactive relationships

Depending on the **relationship cardinality**, the relationship line starts or ends either with an asterisk (*) or a one (1). There is also an arrow on every relationship, which shows the direction of **filter propagation**.

In the next few sections, we'll discuss relationship cardinalities and filter propagation in more detail.

Primary keys/foreign keys

In relational data modeling, the tables may have a column (or a combination of columns) to guarantee the uniqueness of each row of data within that table. If each row in a table is not unique, then we have duplicate rows that we must take from them, either in the source systems or in **Power Query Editor**. The column that guarantees the uniqueness of each row within a table is the **primary key** of that table. The primary key of a table cannot contain blank values. When the primary key of a table appears in a second table, it is, by definition, called a **foreign key**, but only if the data types of both columns are the same. Power BI Desktop currently does not force the same data type requirement. Therefore, we need to be extra careful while creating relationships between two tables.

Handling composite keys

As we mentioned earlier, there might be a combination of columns that guarantee the uniqueness of each row. So, the primary key in such a table would be a composite key containing all the columns that make a row unique. In many relational database systems, including SQL Server, we can create multiple relationships between the two tables by connecting all the columns that contribute to the composite key. This is not legitimate in Power BI Desktop and all other Microsoft products, using the Tabular model engine.

To fix this issue, we need to concatenate all the columns and create a new column that guarantees each row's uniqueness. We can either use **Power Query Editor** to create this new column or use DAX by creating a calculated column, but the Power Query method is preferred. We learned how to do this in *Chapter 6*, *Star Schema Preparation in Power Query Editor*, in the *Creating dimensions* and *Creating facts* sections. To remind you what we did and how it is relevant to handling composite keys, we'll quickly repeat the process here:

1. We created the `Product` dimension while keeping the descriptive values that describe a product derived from the `Sales Base` query.
2. We removed duplicate rows.
3. We identified the key columns that make each row of the `Product` table unique; that is, `ProductCategory`, `ProductSubcategory`, and `Product`.
4. We added an `Index Column` starting from `1`. This is where we handled the composite key. If we didn't add the index column, we had to figure out how to deal with the composite key later. Just to remind you again, the composite key in this scenario is a combination of the `ProductCategory`, `ProductSubcategory`, and `Product` columns; the new `ProductKey` column is the primary key of the `Product` table.
5. When we created the fact table later, we used the `ProductCategory`, `ProductSubcategory`, and `Product` columns from the `Product` table to merge the `Product` table into the `Sales` table. Then, we expanded the `Product structured` column from the `Sales` table by importing the `ProductKey` column from the `Product` table. `ProductKey` in the `Sales` table is now the foreign key.

So, the value of taking the proper steps in the data preparation layer is, again, vital. Since we've already prepared the data to support a proper star schema in the data model, we do not need to deal with composite keys.

As we mentioned earlier, this is the Power Query method, but here is an example of how to deal with composite keys in the **Model** view. There is a good example of composite keys in the `Adventure Works DW (with Exchange Rates).pbix` sample file. This file can be accessed on GitHub via the following link:

`https://github.com/PacktPublishing/Expert-Data-Modeling-with-Power-BI/blob/master/Adventure%20Works%20DW%20(with%20Exchange%20Rates).pbix.`

Here is a scenario: the `Internet Sales` table contains the Sales Amount in different currencies. The business would like to have Internet Sales in USD.

In the sample file, there is an `Exchange Rates` table with no relationship to any other tables.

The following screenshot shows the `Internet Sales` layout of the underlying data model of the sample file:

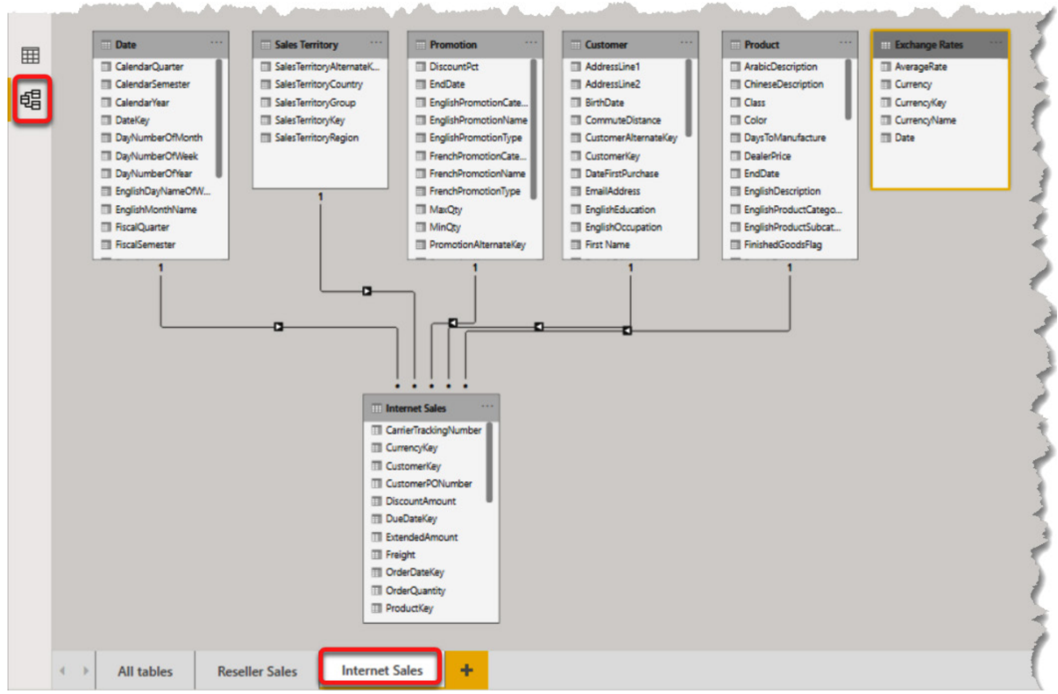


Figure 8.33 – Internet Sales layout in the Model view

We solved the same problem in *Chapter 2, Data Analysis eXpressions and Data Modeling*, in the *Relationships in virtual tables* section. Here, we want to solve the problem by creating a relationship between the `Exchange Rates` table and the `Internet Sales` table.

If we look at the data in the `Exchange Rates` table, we can see that `CurrencyKey` is not the primary key of the table as it contains lots of duplicate values. The following screenshot shows that the `Exchange Rates` table has **14,264 rows**, while the `CurrencyKey` column only has **14 distinct values**:

CurrencyKey	AverageRate	Date	Currency	CurrencyName
100	1	Wednesday, 29 December 2010	USD	US Dollar
100	1	Thursday, 30 December 2010	USD	US Dollar
100	1	Friday, 31 December 2010	USD	US Dollar
100	1	Saturday, 1 January 2011	USD	US Dollar
100	1	Sunday, 2 January 2011	USD	US Dollar
100	1	Monday, 3 January 2011	USD	US Dollar
100	1	Tuesday, 4 January 2011	USD	US Dollar
100	1	Wednesday, 5 January 2011	USD	US Dollar
100	1	Thursday, 6 January 2011	USD	US Dollar
100	1	Friday, 7 January 2011	USD	US Dollar
100	1	Saturday, 8 January 2011	USD	US Dollar
100	1	Sunday, 9 January 2011	USD	US Dollar
100	1	Monday, 10 January 2011	USD	US Dollar
100	1	Tuesday, 11 January 2011	USD	US Dollar

Figure 8.34 – The CurrencyKey column is not the primary key for the Exchange Rates table

The `Date` column is also not the primary key in the `Exchange Rates` table. But a combination of both columns gives us a higher cardinality in the data. Concatenating `CurrencyKey` and `Date` creates a primary key for the `Exchange Rates` table. So, we can use the following DAX expression to create a new calculated column called `ExchKey` in the `Exchange Rates` table:

```
ExchKey = VALUE('Exchange Rates'[CurrencyKey] &
FORMAT('Exchange Rates'[Date], "yyyymmdd"))
```

The following screenshot shows the cardinality of `ExchKey`:

Figure 8.35 – Adding a primary key to the Exchange Rates table

We also need to create a corresponding foreign key column in the `Internet Sales` table. We can use the following DAX expressions to do so:

```
ExchKey = VALUE('Internet Sales'[CurrencyKey] & 'Internet
Sales'[OrderDateKey])
```

Now, we will create a relationship between the `Exchange Rates` table and the `Internet Sales` table using the `ExchKey` column in both tables. Once this relationship has been created, we can create a new measure with a much simpler DAX expression than the one we used in *Chapter 2*, *Data Analysis eXpressions and Data Modeling*. We can use the following DAX expression to create the new measure:

```
Internet Sales in USD =
    SUMX(
        RELATEDTABLE('Exchange Rates')
        , [Internet Sales] * 'Exchange Rates'[AverageRate]
    )
```

If we use the `Internet Sales` and `Internet Sales in USD` measures side by side in a matrix visual, the results look as follows:

Product Category	Internet Sales	Internet Sales in USD
Accessories	**700,759.96**	**$640,920.111**
Bike Racks	39,360.00	$35,934.55
Bike Stands	39,591.00	$35,628.6896
Bottles and Cages	56,798.19	$52,340.3834
Cleaners	7,218.60	$6,352.5092
Fenders	46,619.58	$41,974.101
Helmets	225,335.60	$209,433.4812
Hydration Packs	40,307.67	$35,307.8438
Tires and Tubes	245,529.32	$223,948.5534
Bikes	**28,318,144.65**	**$25,107,749.7623**
Mountain Bikes	9,952,759.56	$8,954,853.1688
Road Bikes	14,520,584.04	$12,599,394.0018
Touring Bikes	3,844,801.05	$3,553,502.5943
Clothing	**339,772.61**	**$306,157.5785**
Caps	19,688.10	$18,674.1168
Gloves	35,020.70	$31,022.1741
Jerseys	172,950.68	$157,333.8088
Shorts	71,319.81	$62,692.697
Socks	5,106.32	$4,665.6082
Vests	35,687.00	$31,769.1769
Total	**29,358,677.22**	**$26,054,827.4537**

Figure 8.36 – A new version of the Internet Sales in USD measure

The preceding scenario shows how data modeling can simplify our DAX expressions. Just as a reminder, here is the same measure that we created in *Chapter 2*, *Data Analysis eXpressions and Data Modeling*, without creating the relationship between the `Exchange Rates` table and the `Internet Sales` table:

```
Internet Sales USD =
SUMX(
    NATURALINNERJOIN (
            SELECTCOLUMNS(
                'Internet Sales'
, "CurrencyKeyJoin", 'Internet Sales'[CurrencyKey] * 1
                , "DateJoin", 'Internet Sales'[OrderDate] + 0
                , "ProductKey", 'Internet Sales'[ProductKey]
                , "SalesOrderLineNumber", 'Internet
Sales'[SalesOrderLineNumber]
                , "SalesOrderNumber", 'Internet
Sales'[SalesOrderNumber]
                , "SalesAmount", 'Internet Sales'[SalesAmount]
                )
            , SELECTCOLUMNS (
                'Exchange Rates'
                , "CurrencyKeyJoin", 'Exchange
Rates'[CurrencyKey] * 1
                , "DateJoin", 'Exchange Rates'[Date] + 0
                , "AverageRate", 'Exchange Rates'[AverageRate]
            )
        )
, [AverageRate] * [SalesAmount]
)Relationship cardinalities
```

In Power BI, we can create a relationship between two tables by linking a column from the first table to a column from the second table. There are three cardinalities of relationships in relational data modeling: **one-to-one**, **one-to-many** and **many-to-many**. In this section, we'll briefly look at each from a data modeling viewpoint in Power BI.

One-to-one relationships

A one-to-one relationship is when we create a relationship between two tables using the primary keys from both tables. Every row of the first table is related to a zero or one row from the second table. For that reason, the direction of filtering in a one-to-one relationship in Power BI is always **bidirectional**. When we have a one-to-one relationship, we can potentially combine the two tables into one table, unless the business case we are working on dictates otherwise. We generally recommend avoiding one-to-one relationships when possible.

One-to-many relationships

A one-to-many relationship, which is the most common relationship cardinality, is when each row of the first table is related to many rows of the second table. Power BI Desktop uses 1 - * to indicate a one-to-many relationship.

Many-to-many relationships

A many-to-many relationship is when a row of data from the first table is related to many rows of data in the second table, and a row of data in the second table is related to many rows in the first table. While in a proper star schema, all relationships between dimensions tables and fact tables are one-to-many, the many-to-many relationship is still a legitimate cardinality relationship in Power BI. With many-to-many relationships, the necessity of having a primary table in tables participating in the relationship goes away. When we define a many-to-many relationship between two tables, the default behavio sets the filtering to bidirectional. But depending on the scenario, we can force the direction of filtering to go in a single direction. Power BI uses * - * to indicate a many-to-many relationship.

An excellent example of a many-to-many relationship can be found in our sample file (`Chapter 8, Data Modelling and Star Schema.pbix`). This shows that the business needs to create analytical reports on top of the sales for the **Customers with Yearly Income Greater Than $100,000** summary table at higher granular levels such as **Quarter** or **Year**. To allow the business to achieve this requirement, we just need to create a relationship between the `Sales for Customers with Yearly Income Greater Than $100,000` table and the `Date` table using the `Year-Month` column in both tables. We set **Cross filter direction** to **Single**, so the `Date` table filters the `Sales for Customers with Yearly Income Greater Than $100,000` table. The following screenshot shows the **Manage relationships** window upon creating the preceding relationship:

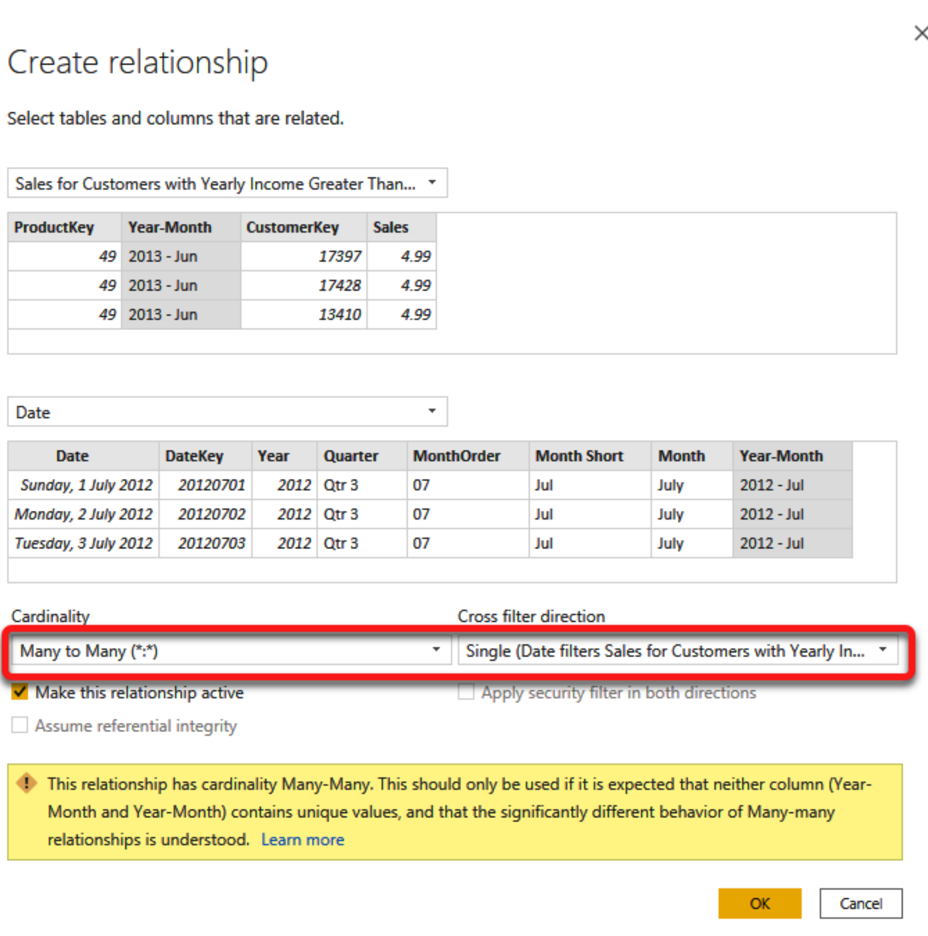


Figure 8.37 – Creating a many-to-many relationship

As the preceding screenshot shows, a warning message appears at the bottom of the **Manage relationships** window, explaining that the columns participating in a many-to-many relationship do not contain unique values. So, we must be very careful when using a many-to-many relationship unless we know what we are doing is correct. The following screenshot shows how we can visualize the data after creating the relationship:

ProductCategory	ProductSubcategory	2011	2012	2013	2014	Total
⊞ **Accessories**			**119.95**	**88,497.11**	**4,452.03**	**93,069.09**
⊟ Bikes	⊞ Mountain Bikes	168,048.61	298,209.78	1,364,324.68	6,934.97	**1,837,518.04**
	⊞ Road Bikes	837,601.50	555,045.75	682,488.09	1,120.49	**2,076,255.83**
	⊞ Touring Bikes			773,956.08	7,152.21	**781,108.29**
	Total	**1,005,650.10**	**853,255.53**	**2,820,768.85**	**15,207.67**	**4,694,882.16**
⊞ **Clothing**				**34,343.47**	**1,674.55**	**36,018.02**
Total		**1,005,650.10**	**853,375.48**	**2,943,609.43**	**21,334.25**	**4,823,969.27**

Figure 8.38 – Visualizing data from tables participating in a many-to-many relationship

It is important to note the granularity of the data. As the preceding screenshot shows, we can visualize the data at the Year and Month levels, but if we want to go one level further down, the data still represents the Year-Month level. So, it is crucial to make the visualization available to the consumers at a correct level, so that they do not get confused by seeing the data at an incorrect level of granularity.

Note

We suggest not using many-to-many cardinality in a relationship as it can dramatically elevate the level of model complexity, primarily when both tables participating in the relationship are also related to other tables. This situation can become even worse if we set the cross-filter direction of the relationship to **Both**. As a result, we may potentially need to write more complex DAX expressions, which means we end up facing poor model performance. Instead, it is better to handle a many-to-many relationship using bridge tables. More on this will be covered in *Chapter 9, Star Schema and Data Modeling Common Best Practices*.

Filter propagation behavior

Filter propagation is one of the most important concepts to understand when building a data model in Power BI Desktop. When we create a relationship between two tables, we are also filtering the data of one table by the data of another. We can see the direction of filter propagation in the **Model** view for each relationship. The following screenshot shows the relationship between the `Product` and `Sales` tables and the direction of filtering the data:

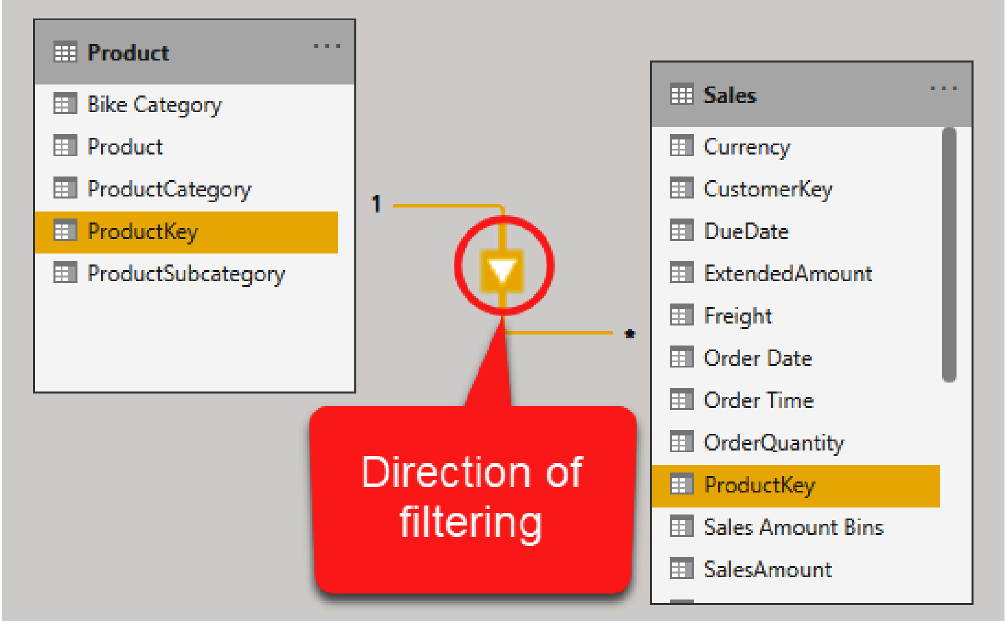


Figure 8.39 – The direction of filtering the data via a relationship

The relationship shown in the preceding screenshot shows the following:

- Each row of data in the `Product` table (the 1 side of the relationship) is related to many rows of data in the `Sales` table (the * side of the relationship).
- If we filter a row of data in the `Product` table, the filter propagates through the relationship from the `Product` table to the `Sales` table.

The following screenshot shows how the filter propagates from the `Product` table to the `Sales` table via the relationship between the two:

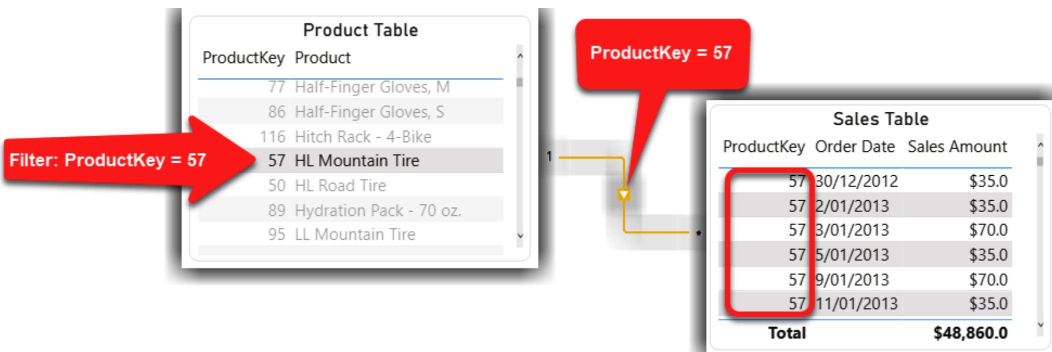


Figure 8.40 – Filter propagation via a relationship

As the preceding screenshot shows, when we click on a row of the `Product` table and select that `ProductKey` equals `57`, we are filtering the `Product` table and filtering the `Sales` table through the relationship. Therefore, the `Sales` table shows only the rows of data where `ProductKey` equals `57`. While the filter we put on `ProductKey` propagates from the `Product` table to the `Sales` table, it does not go any further than the `Sales` table. The reason for this is that the `Sales` table does not have any relationships with any other tables, with the filter direction going from the `Sales` table to the other table. The following screenshot shows how the filtering flows in our latter example:

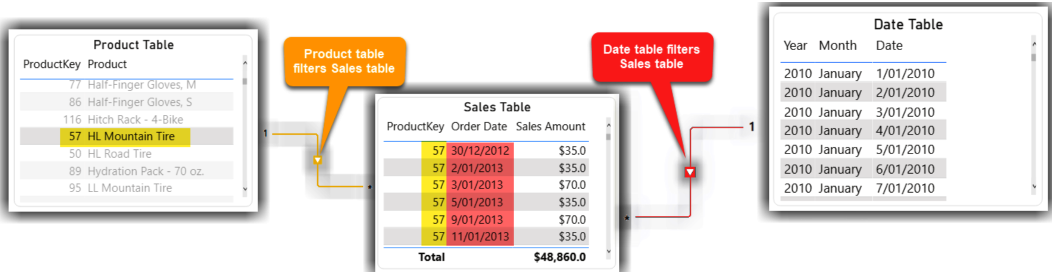


Figure 8.41 – Filter propagation

As the preceding screenshot shows, while the `Product` table and the `Date` table can filter the `Sales` table data, the filter does not flow in the opposite direction. Therefore, the `Sale` table can never filter the `Date` table with the relationships defined in the preceding model.

Bidirectional relationships

Now that we understand filter propagation, we can understand what a bidirectional relationship means and how it affects our data model. A relationship is bidirectional when we set its **Cross-filter direction** to **Both**. The ability to set the direction of filtering to both directions is a nice feature, since it can help solve some data visualization challenges. An excellent example is when we use two slicers on the report page, with one showing the `ProductCategory` column data and the other showing the **Full Name** data. The end user expects to see only relevant data in each slicer when selecting a value from the slicers. The following image shows the preceding scenario:

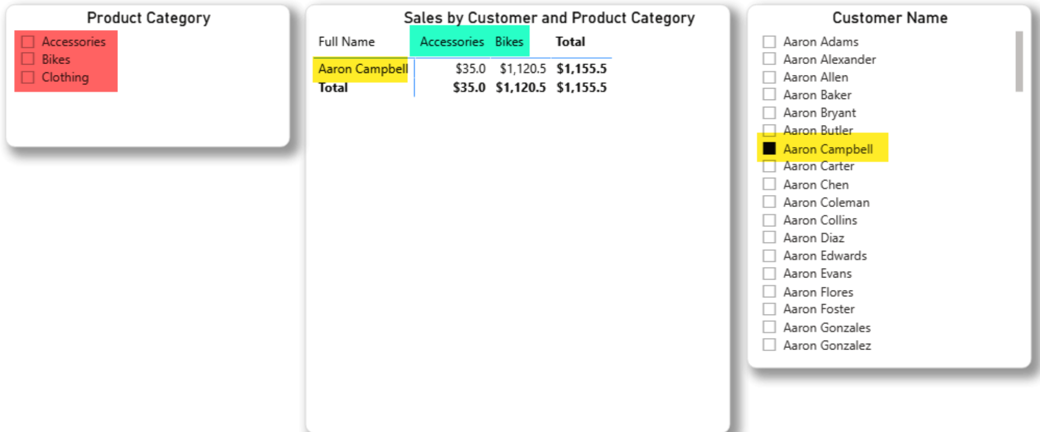


Figure 8.42 – The Customer Name slicer filters the Sales data but not the Product Category data

As the preceding image illustrates, when the user selects a value from the **Customer Name** slicer, the filter propagates from the `Customer` table to the `Sales` table via the relationship between them. Therefore, the relevant data is shown in the table visual. So, here, we can see that **Aaron Campbell** bought some accessories and bikes. Yet, the `Product Category` slicer still shows all the product categories available in the data model. The following screenshot shows how the `Product`, `Customer`, and `Sales` tables relate to each other:

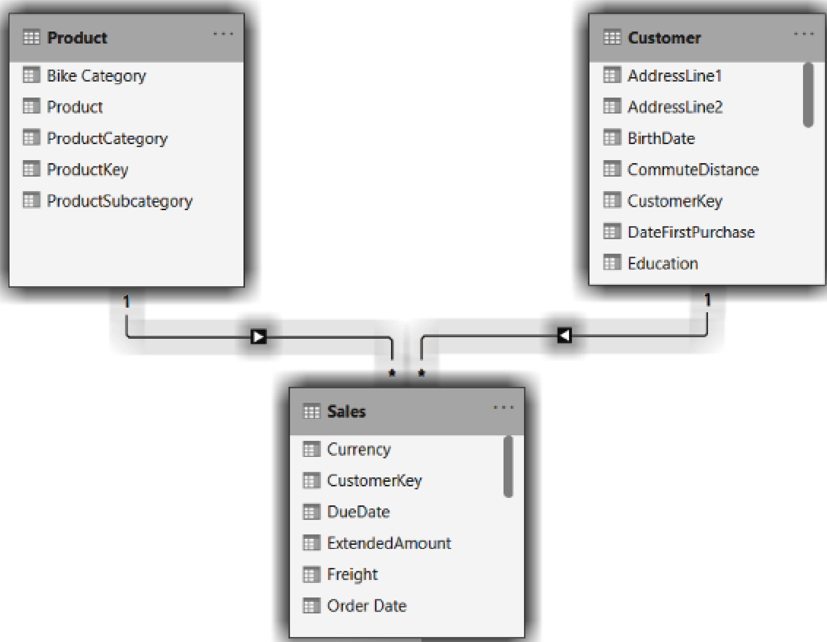


Figure 8.43 – The relationships between the Product, Customer, and Sales tables

What the end user expects is to only see `Accessories` and `Bikes` in the **Product Category slicer**. One way to solve this issue is to set the relationship between the `Sales` table and the `Product` table to bidirectional by setting the relationship's **Cross-filter direction** to **Both**.

Follow these steps to change **Cross-filter direction** to **Both** directions:

1. Switch to the **Model** view.
2. Double-click the relationship between the `Product` and `Sales` tables.
3. Set the **Cross-filter direction** to **Both**.
4. Click **OK**.

The following screenshot shows the preceding steps:

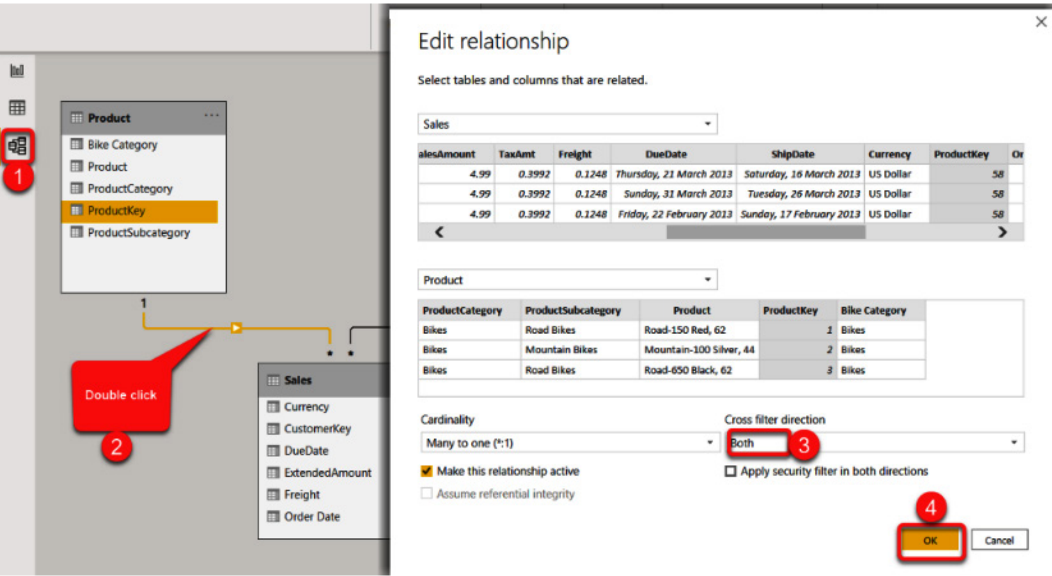


Figure 8.44 – Making a relationship bidirectional

The following screenshot shows how the relationship changed in the **Model** view:

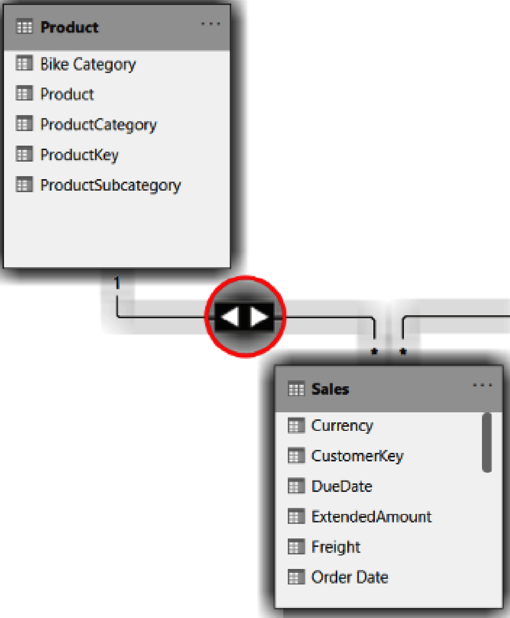


Figure 8.45 – Visual representation of a bidirectional relationship

Now, when we go back to the **Report** view, we will see that making a relationship between the `Date` and `Sales` tables resolved this issue. The following image shows the results after making the relationship bidirectional:

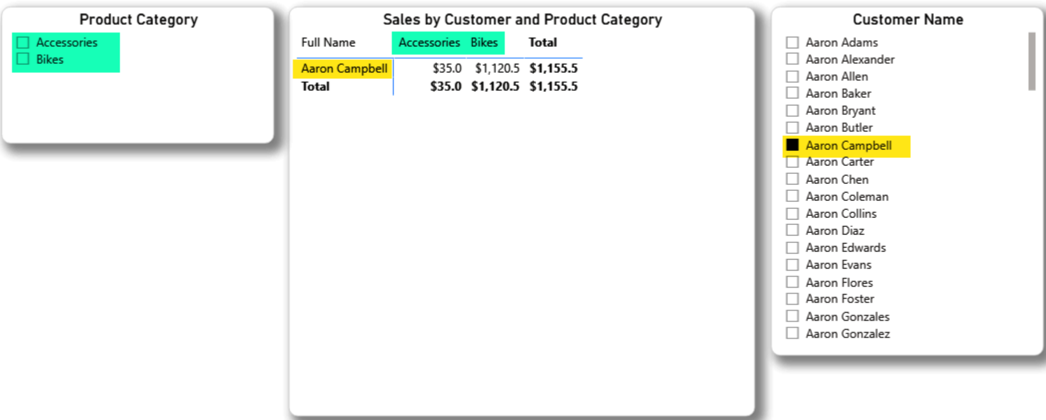


Figure 8.46 – The Customer Name slicer filters the Sales data and the Product Category data

Now, if the user selects a value from the **Product Category** slicer, the filter will propagate via the current relationship from the `Product` table to the `Sales` table. However, it will not propagate from the `Sales` table to the `Customer` table via the current relationship, since the relationship between the `Sales` table and the `Customer` table is not bidirectional. So, we should also set the latter relationship to bidirectional.

Looking at the preceding scenario raises an important point: we may end up making all the model's relationships bidirectional. Using bidirectional relationships can be a killer, especially in larger data models with more complex relationships. Bidirectional relationships have adverse effects on model performance. It also elevates the complexity level of DAX expressions dramatically. We suggest avoiding bidirectional relationships as much as possible. There are some more advanced techniques to solve similar scenarios to what we've covered in better ways without using bidirectional relationships, which we will cover in *Chapter 9*, *Star Schema and Data Modeling Common Best Practices*.

Summary

In this chapter, we learned about the data modeling components in Power BI Desktop. We learned about table and field properties; we looked at feature tables and how to make a table from our data model accessible across the organization; and we also learned how to build summary tables by creating calculated tables in DAX. We then dived deeper into one of the essential concepts in data modeling, which is relationships. We learned about different relationship cardinalities and filter propagation, and we also understood the concept of bidirectional relationships.

In the next chapter, *Star Schema and Data Modeling Common Best Practices*, we'll look at many of the concepts we learned about in this chapter in more detail.

9
Star Schema and Data Modeling Common Best Practices

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

- Dealing with many-to-many relationships
- Being cautious with bidirectional relationships
- Dealing with inactive relationships
- Using configuration tables

- Avoiding calculated columns when possible
- Organizing the model
- Reducing model size by disabling auto date/time

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

Dealing with many-to-many relationships

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

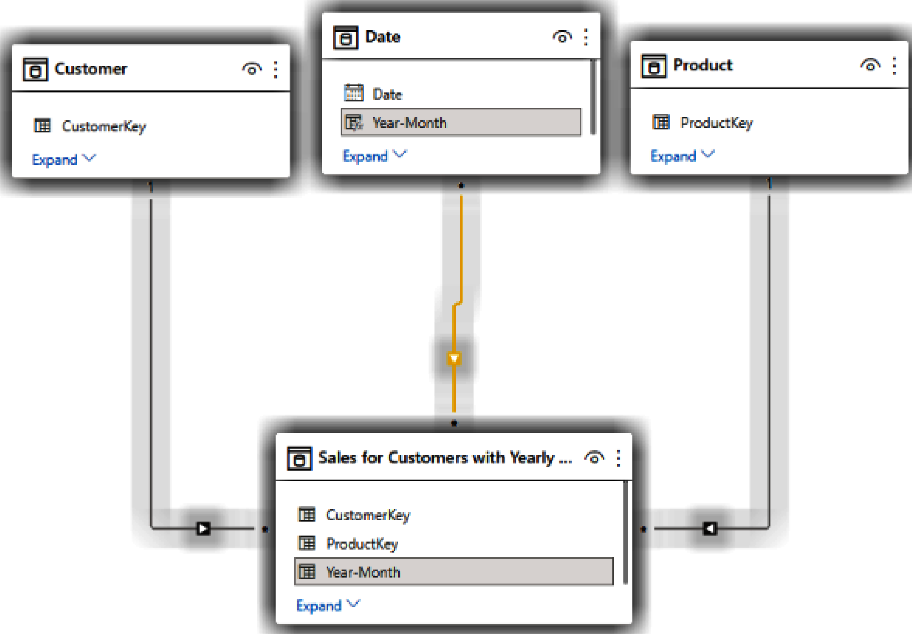


Figure 9.1 – A many-to-many cardinality relationship

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

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

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

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

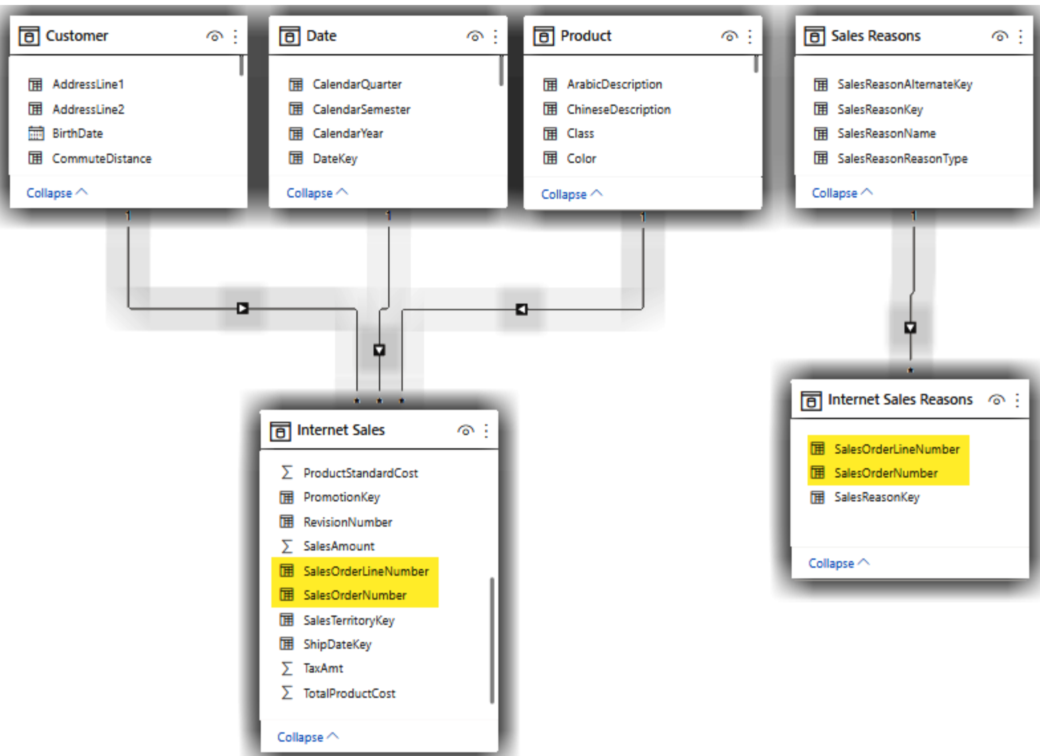


Figure 9.2 – Analyzing Internet Sales by Sales Reasons

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

Many-to-many relationships using a bridge table

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

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

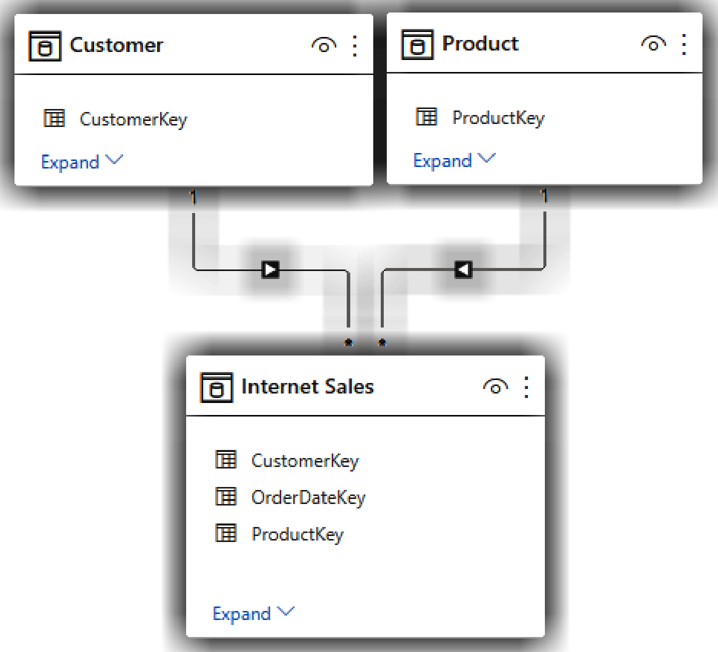


Figure 9.3 – Relationships between Customer, Product, and Internet Sales

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

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

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

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

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

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

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

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

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

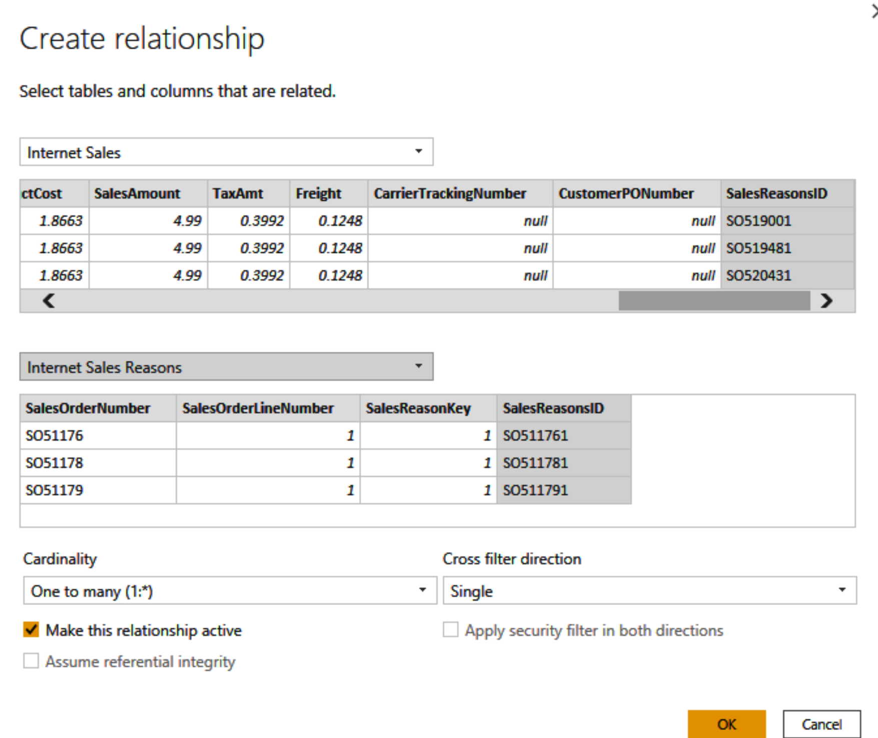


Figure 9.4 – Creating a relationship between the Internet Sales and the Internet Sales Reasons tables

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

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

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

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

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

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

The following screenshot shows the preceding steps:

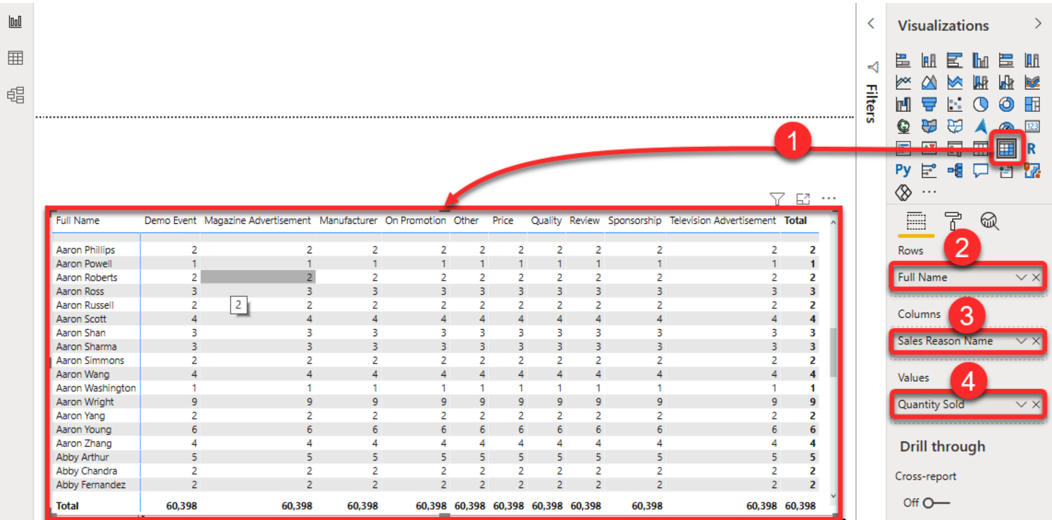


Figure 9.6 – Visualizing customers' full name, sales reason, and quantity sold in a matrix

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

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

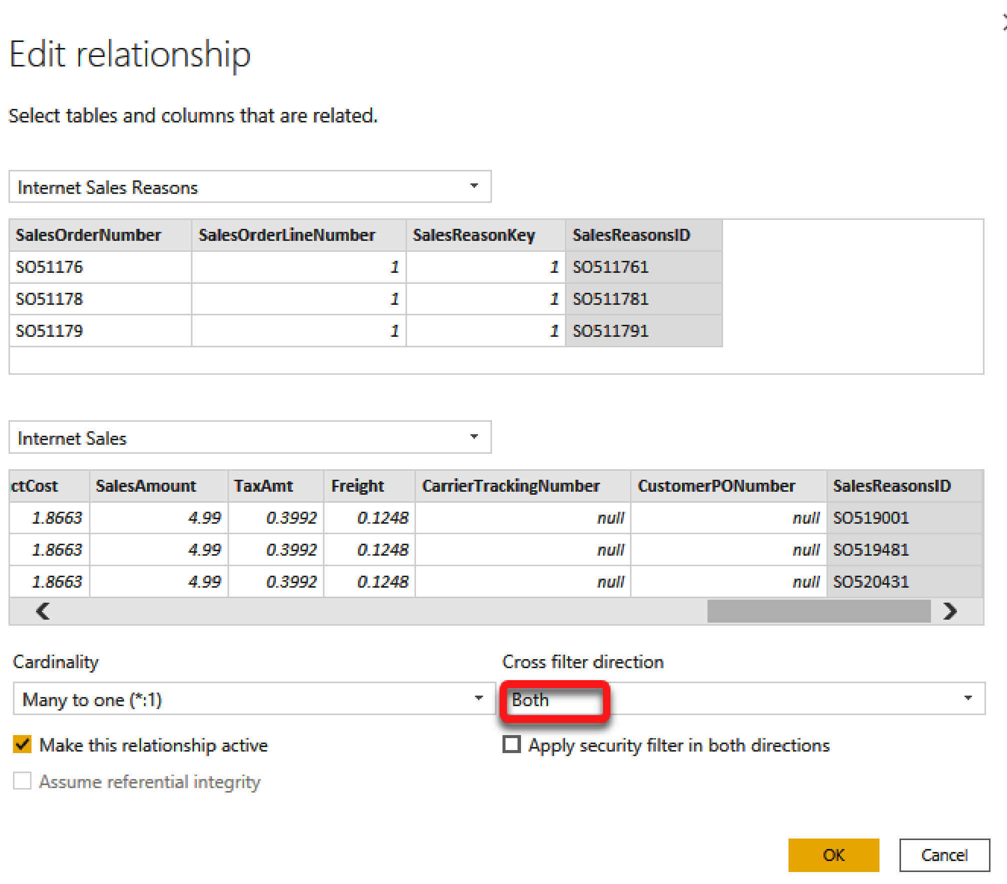


Figure 9.7 – Setting up a bidirectional relationship

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

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

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

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

1. Put another **Matrix** on the report canvas.
2. Choose the `Product Name` column from the `Product` table in the **Rows** dropdown.
3. Choose the `Sales Reason Name` column from the `Sales Reasons` table in the **Columns** dropdown.

4. Choose `Quantity Sold` measure in the **Values** dropdown.
5. Click on the **Aaron Wang** row from the previous Matrix visual to cross-filter the new Matrix.

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

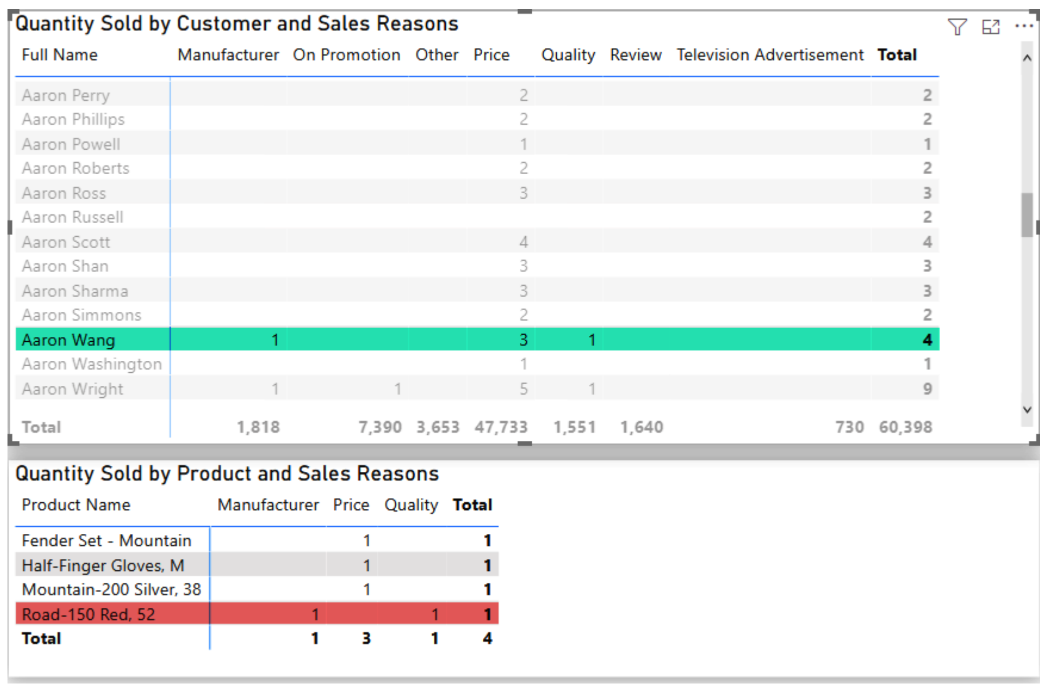


Figure 9.9 – Filtering Quantity Sold by Product and Sales Reasons with Customer

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

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

```
            , BLANK()
            )
```

Hiding the bridge table

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

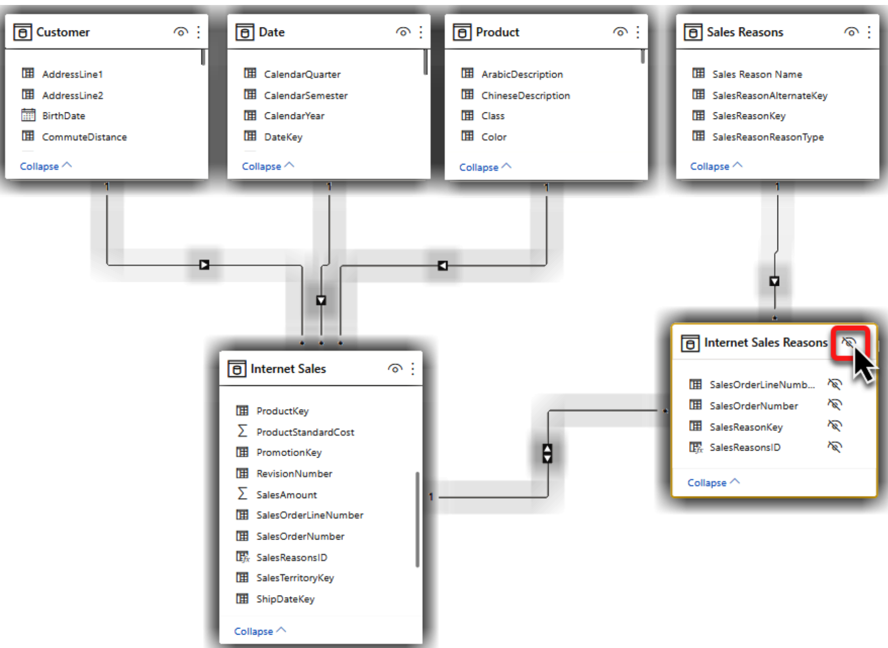


Figure 9.10 – Hiding bridge tables from the Model view

Being cautious with bidirectional relationships

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

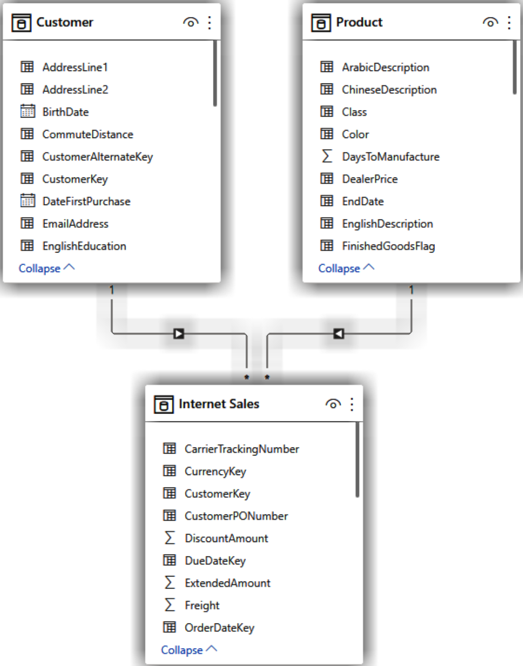


Figure 9.11 – Internet Sales data model

The following diagram shows the reporting requirements:

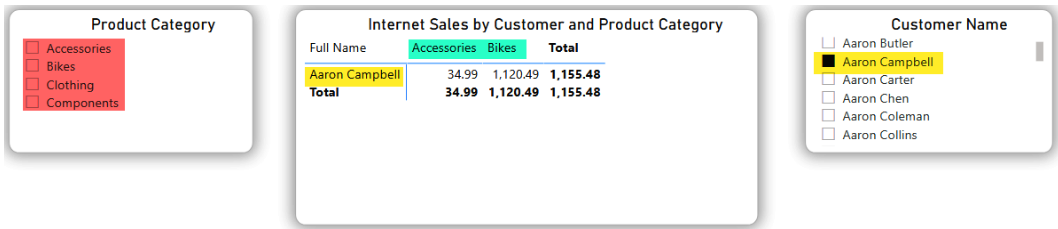


Figure 9.12 – The Customer Name slicer filters the Sales data but not the Product Category data

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

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

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

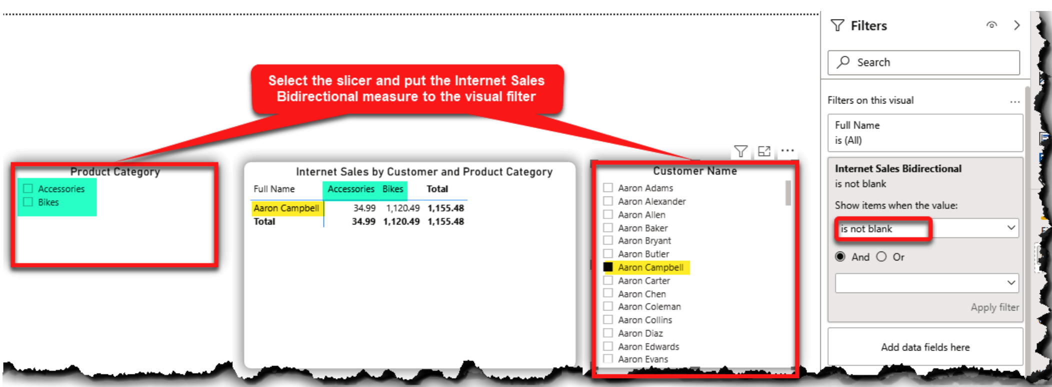


Figure 9.13 – Using a measure in the visual filter for slicers

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

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

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

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

Dealing with inactive relationships

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

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

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

Reachability via multiple filter paths

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

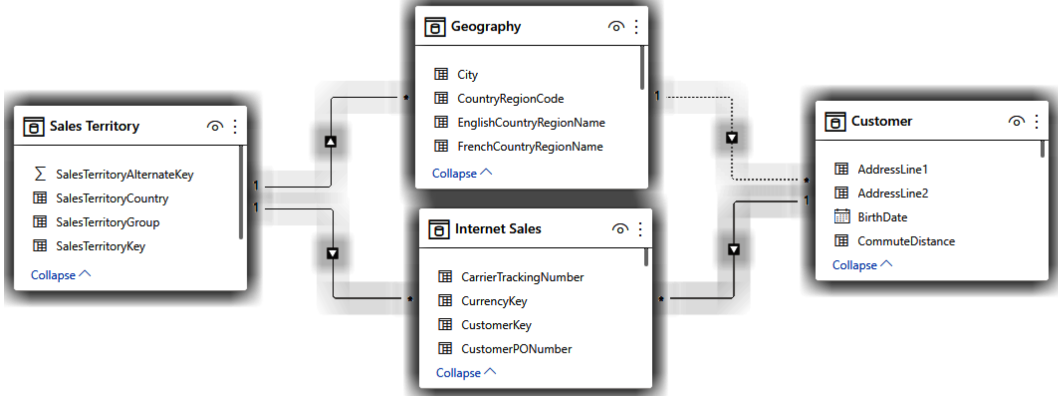


Figure 9.14 – Inactive relationship due to multiple paths detected

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

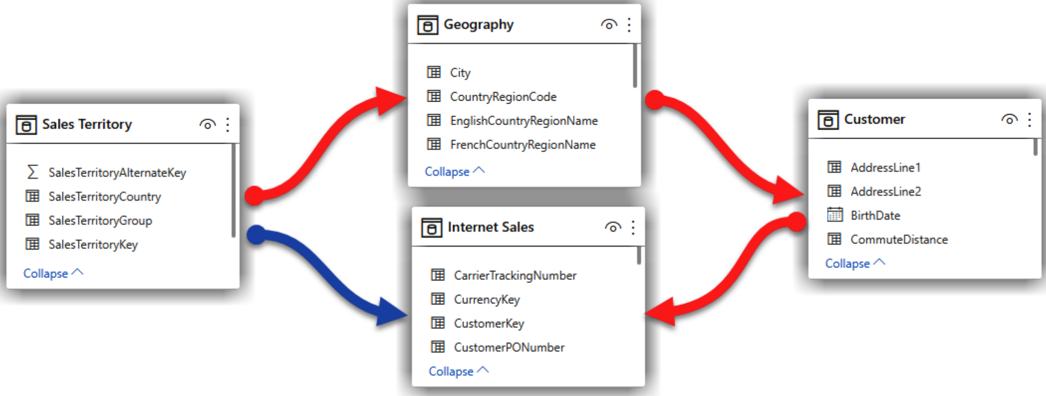


Figure 9.15 – Internet Sales reachability through multiple paths

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

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

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

Multiple direct relationships between two tables

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

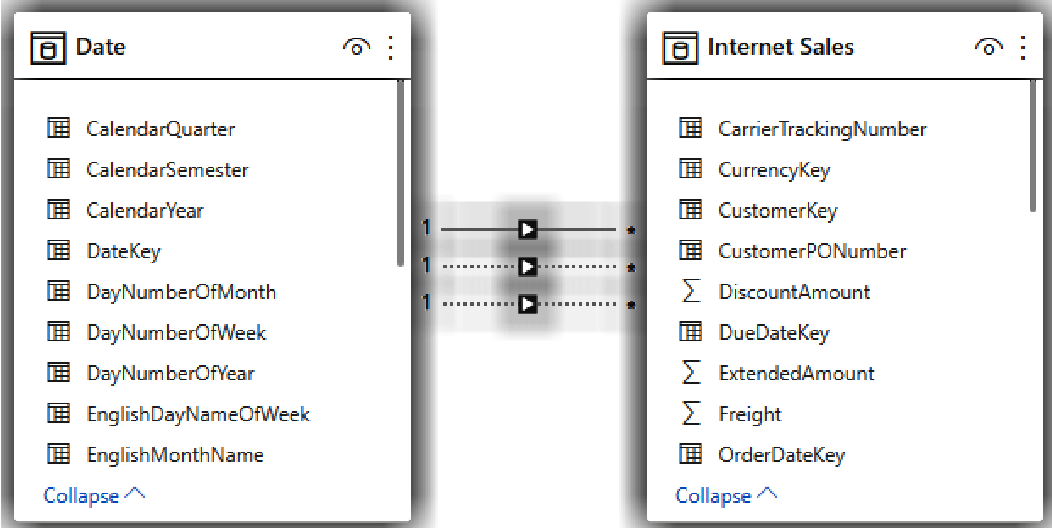
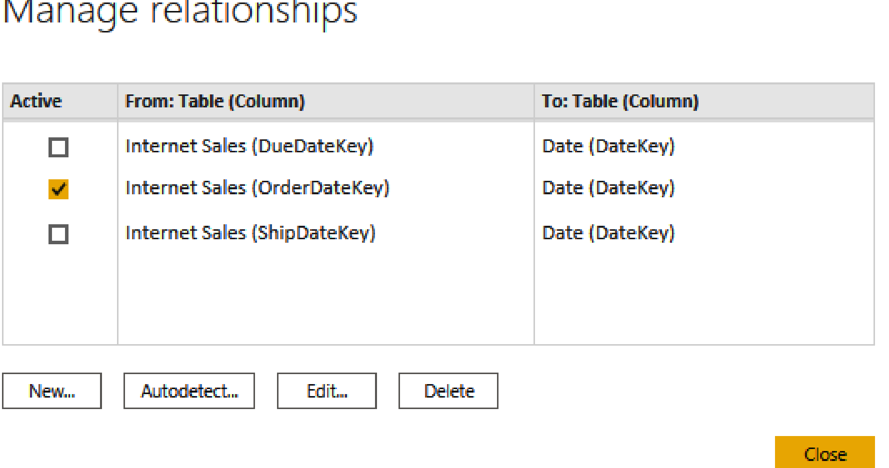


Figure 9.16 – Two tables with multiple direct relationships

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



Active	From: Table (Column)	To: Table (Column)
☐	Internet Sales (DueDateKey)	Date (DateKey)
☑	Internet Sales (OrderDateKey)	Date (DateKey)
☐	Internet Sales (ShipDateKey)	Date (DateKey)

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

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

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

```
'Date'[DateKey])
      )
```

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

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

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

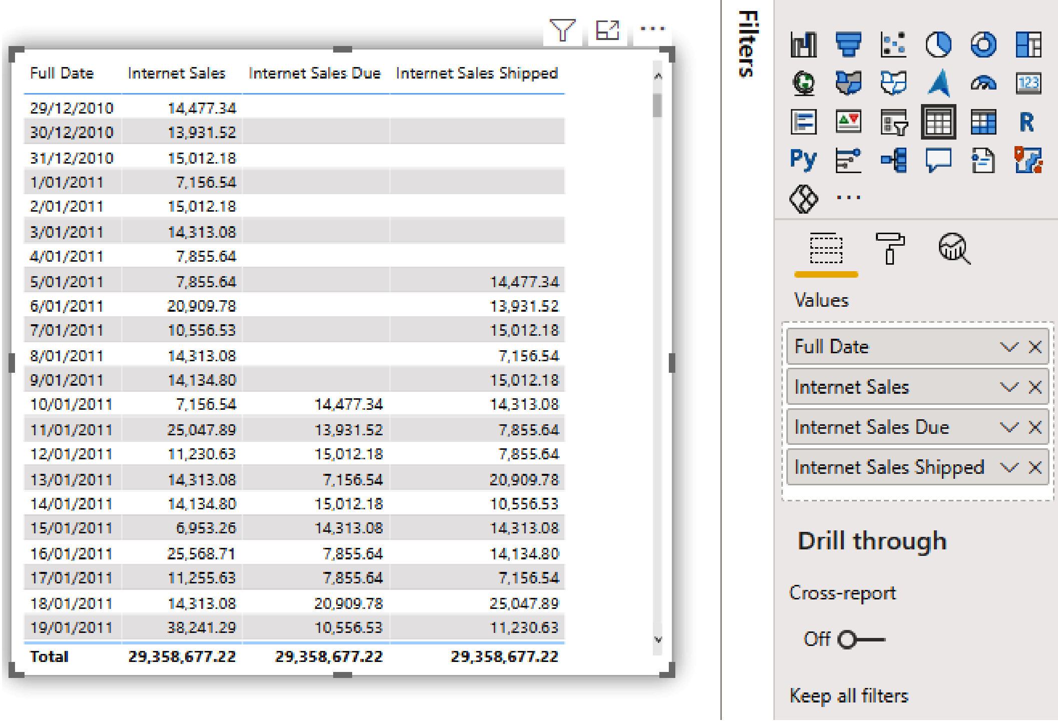


Figure 9.18 – Activating inactive relationships programmatically

Using configuration tables

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

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

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

Segmentation

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

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

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

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

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

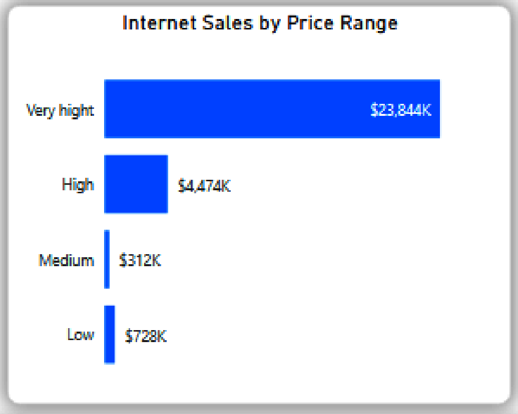
Sort	Price Range	From	To
4	Low	0	50
3	Medium	51	450
2	High	451	1500
1	Very hight	1501	15000

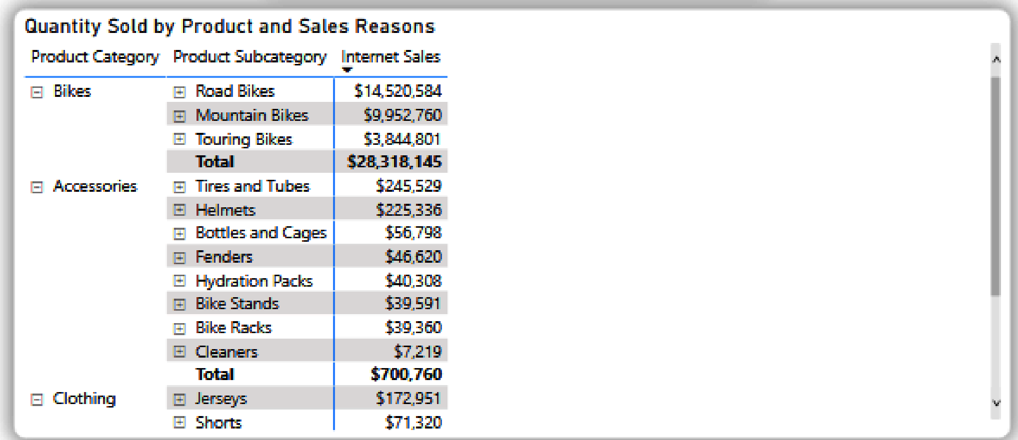
Figure 9.19 – Unit Price Ranges table

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

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

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





Quantity Sold by Product and Sales Reasons

Product Category	Product Subcategory	Internet Sales
Bikes	Road Bikes	$14,520,584
	Mountain Bikes	$9,952,760
	Touring Bikes	$3,844,801
	Total	**$28,318,145**
Accessories	Tires and Tubes	$245,529
	Helmets	$225,336
	Bottles and Cages	$56,798
	Fenders	$46,620
	Hydration Packs	$40,308
	Bike Stands	$39,591
	Bike Racks	$39,360
	Cleaners	$7,219
	Total	**$700,760**
Clothing	Jerseys	$172,951
	Shorts	$71,320

Figure 9.20 – Analyzing Internet Sales by price range

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

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

Dynamic conditional formatting with measures

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

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

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

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

Figure 9.21 – Config table defining color codes

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

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

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

Figure 9.22 – The ConfigColour table

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

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

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

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

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

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

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

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

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

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

Here is how the preceding formula works:

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

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

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

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

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

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

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

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

Figure 9.23 – Power BI default visuals supporting conditional formatting

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

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

The following screenshot shows the preceding steps:

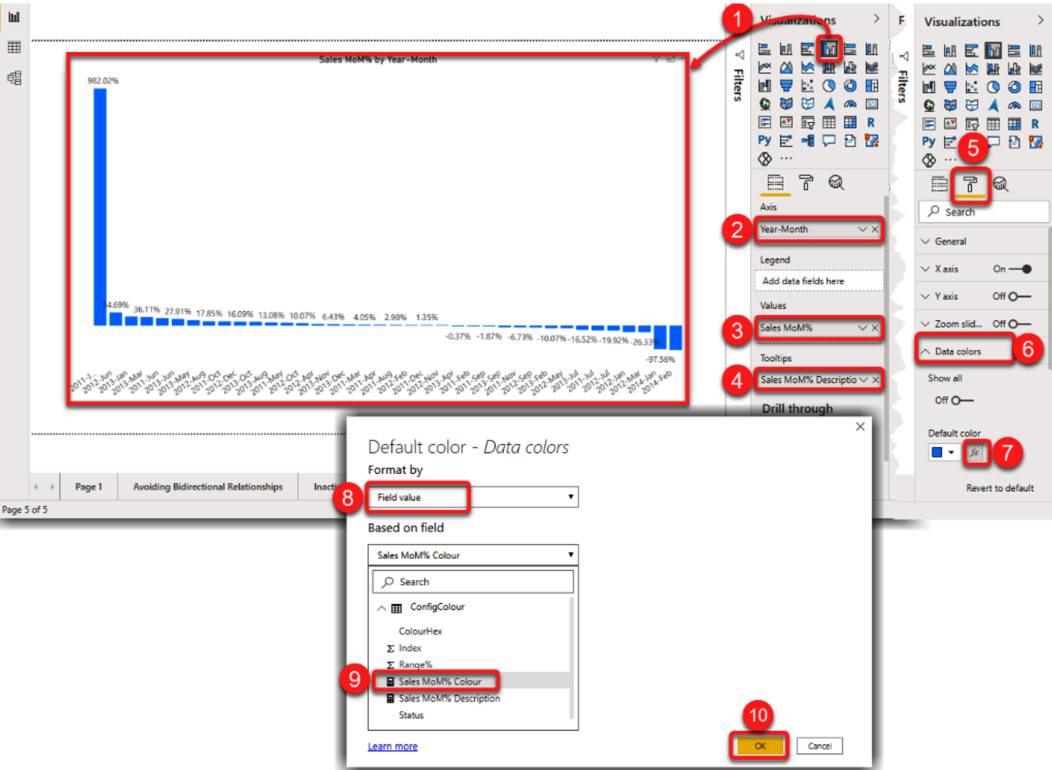


Figure 9.24 – Applying dynamic conditional formatting on a clustered column chart

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

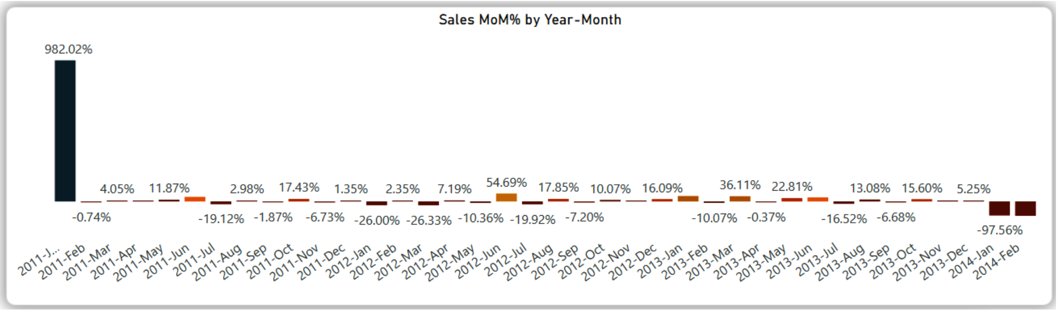
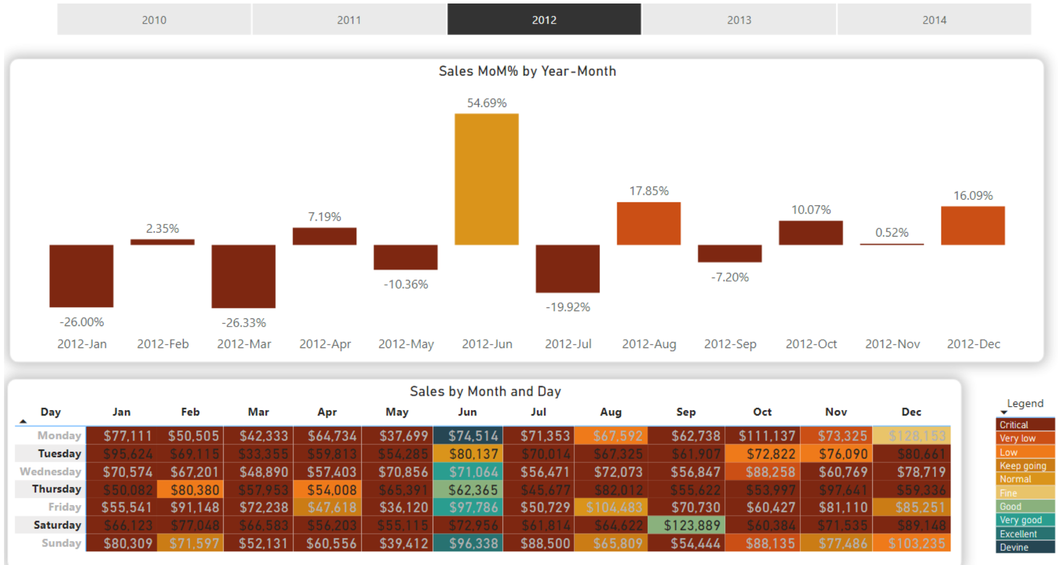


Figure 9.25 – The clustered column chart after it has been conditionally formatted

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



Day	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Monday	$77,111	$50,505	$42,333	$64,734	$37,699	$74,514	$71,353	$67,592	$62,738	$111,137	$73,325	$128,153
Tuesday	$95,624	$69,115	$33,355	$59,813	$54,285	$80,137	$70,014	$67,325	$61,907	$72,822	$76,090	$80,661
Wednesday	$70,574	$67,201	$48,890	$57,403	$70,856	$71,064	$56,471	$72,073	$56,847	$88,258	$60,769	$78,719
Thursday	$50,082	$80,380	$57,953	$54,008	$65,391	$62,365	$45,677	$82,012	$55,622	$53,997	$97,641	$59,336
Friday	$55,541	$91,148	$72,238	$47,618	$36,120	$97,786	$50,729	$104,483	$70,730	$60,427	$81,110	$85,251
Saturday	$66,123	$77,048	$66,583	$56,203	$55,115	$72,956	$61,814	$64,622	$123,889	$60,384	$71,535	$89,148
Sunday	$80,309	$71,597	$52,131	$60,556	$39,412	$96,338	$88,500	$65,809	$54,444	$88,135	$77,486	$103,235

Figure 9.26 – Color-coded Sales report

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

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

Avoiding calculated columns when possible

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

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

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

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

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

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

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

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

The following screenshot shows the preceding steps:

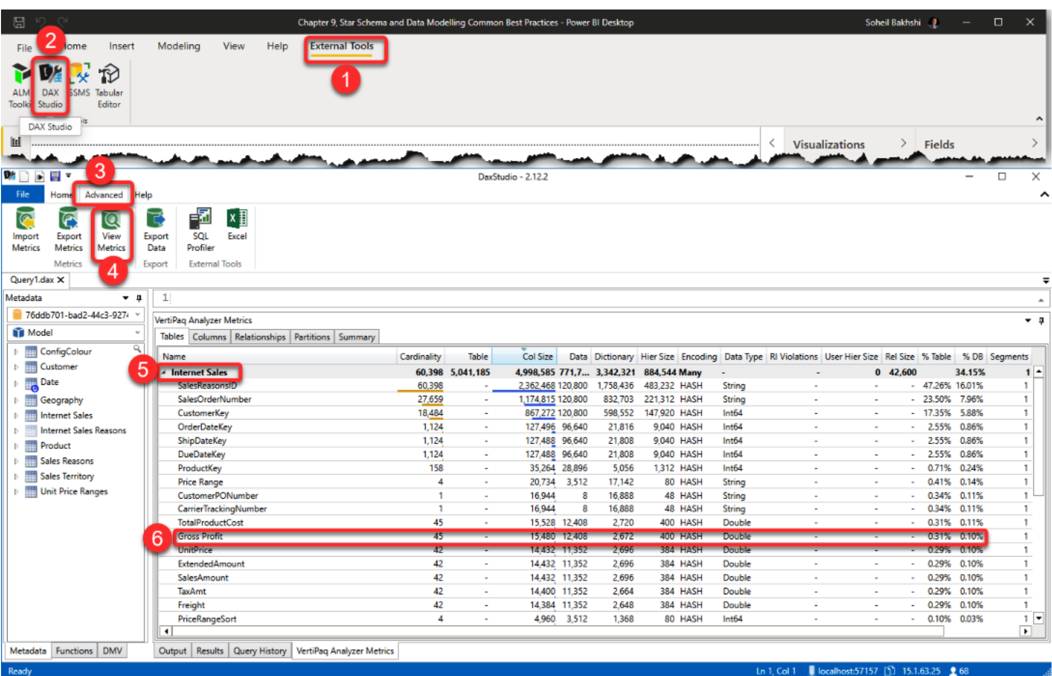


Figure 9.27 – View Metrics in DAX Studio

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

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

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

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

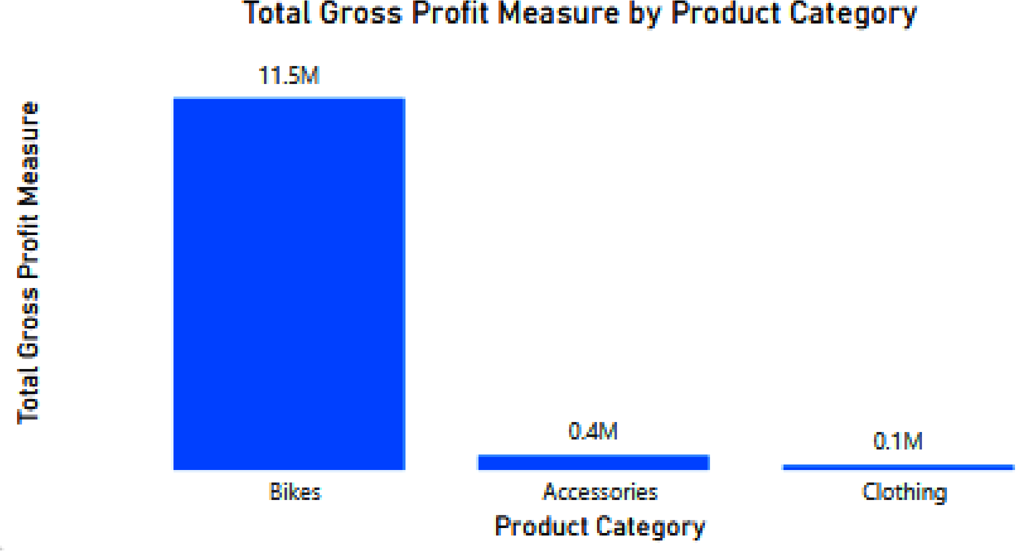


Figure 9.28 – Total Gross Profit Measure by Product Category in a clustered column chart

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

Organizing the model

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

Hiding insignificant model objects

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

Hiding unused fields and tables

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

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

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

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

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

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

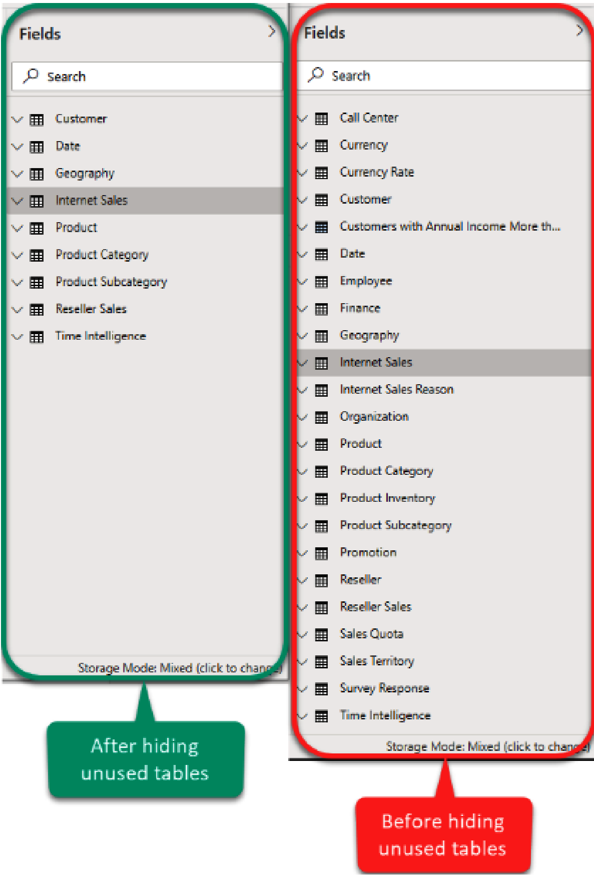


Figure 9.29 – Before and after hiding unused tables using Power BI Documenter

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

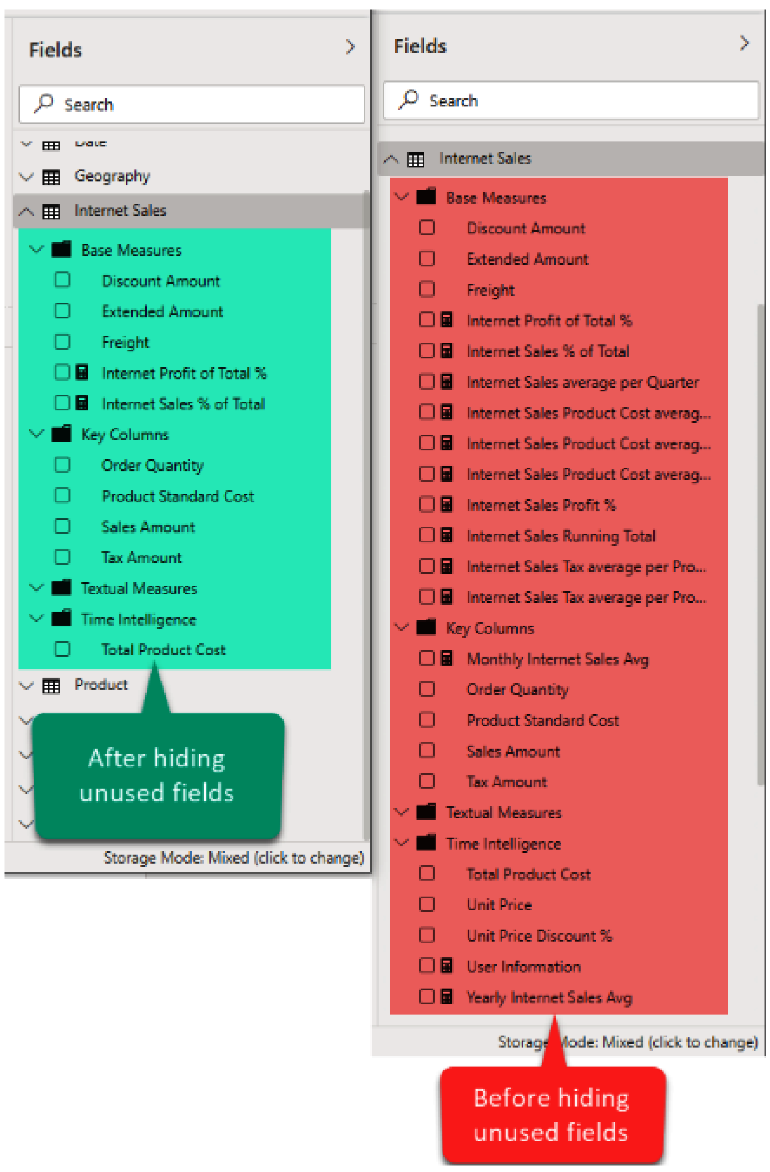


Figure 9.30 – Before and after hiding unused tables using Power BI Documenter

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

Hiding key columns

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

Hiding implicit measures

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

Hiding columns used in hierarchies when possible

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

Creating measure tables

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

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

4. Click **Load**:

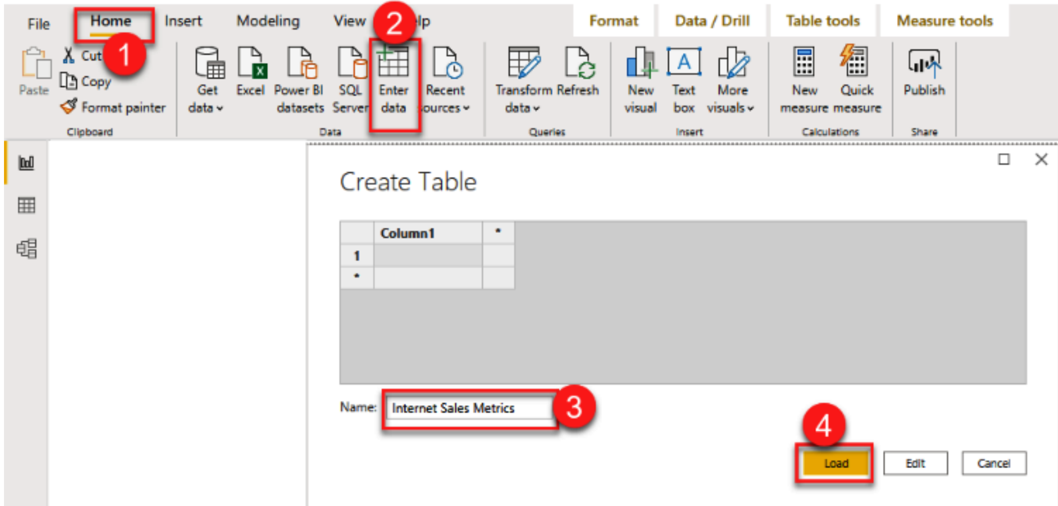


Figure 9.31 – Entering data in Power BI Desktop

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

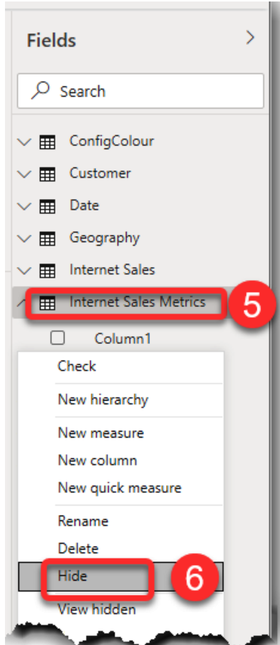


Figure 9.32 – Hiding a column

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

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

The following screenshot shows the preceding three steps:

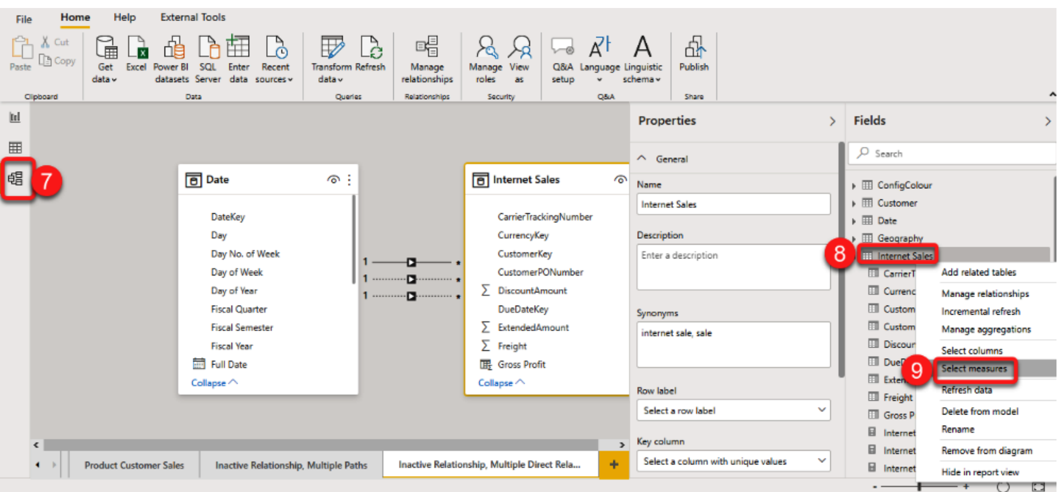


Figure 9.33 – Selecting all measures from a table

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

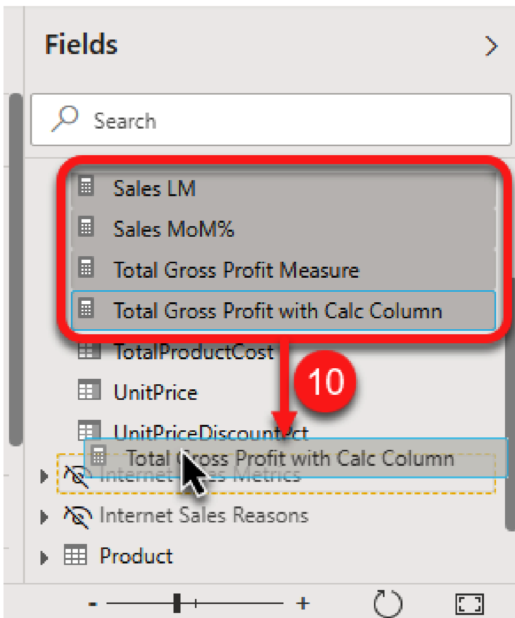


Figure 9.34 – Moving multiple measures from a table to another

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

The following screenshot shows the preceding two steps:

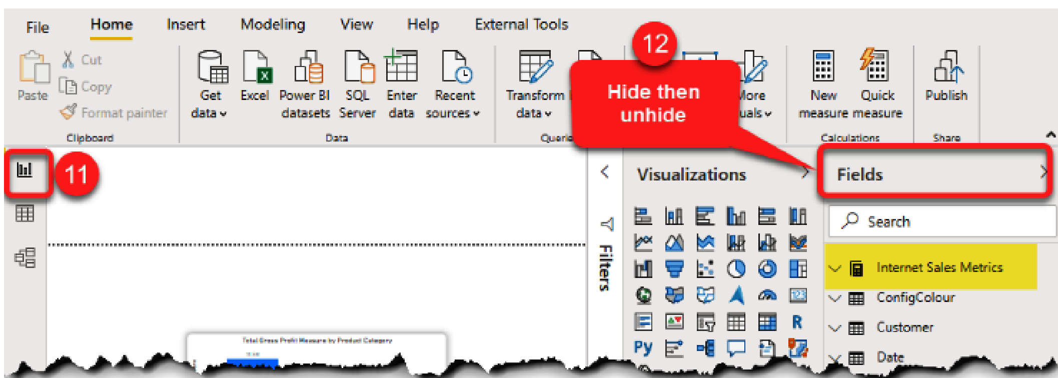


Figure 9.35 – The measure table created in Power BI Desktop

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

Considerations

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

Using folders

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

Creating a folder in multiple tables in one go

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

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

The following screenshot illustrates the preceding steps:

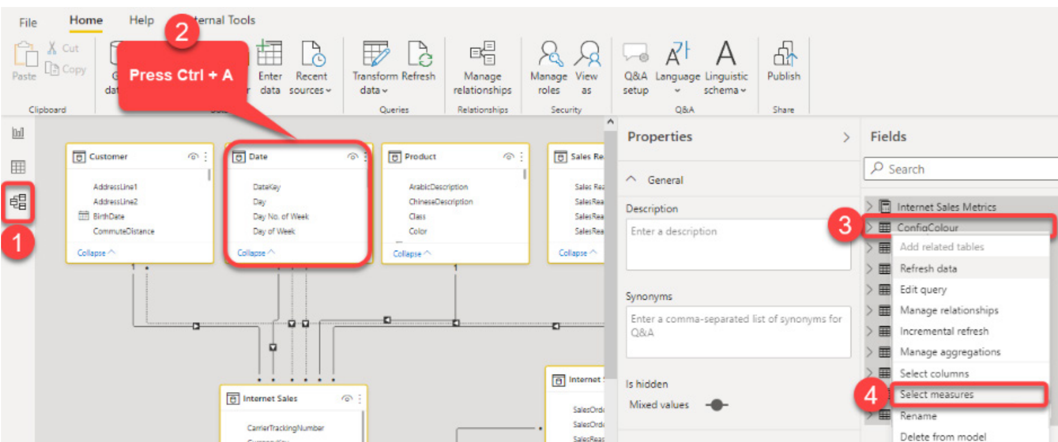


Figure 9.36 – Selecting all measures in the model

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

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

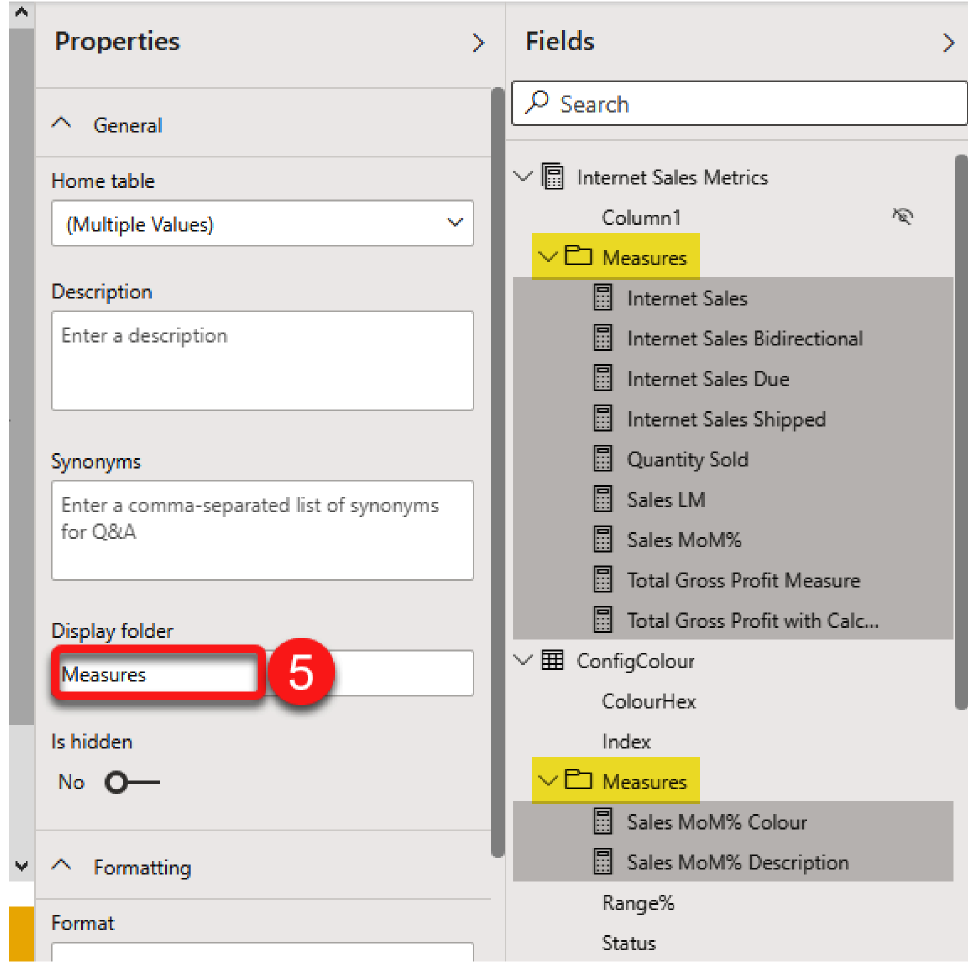


Figure 9.37 – Placing all selected measures in a folder within multiple tables

Placing a measure in multiple folders

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

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

The following screenshot shows the preceding steps:

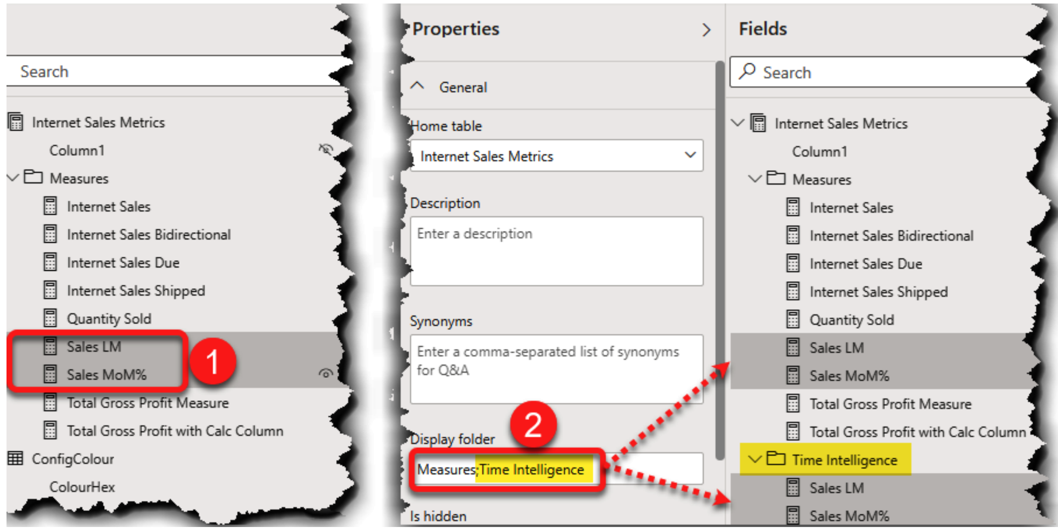


Figure 9.38 – Placing measures in multiple folders

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

```
Measures;Time Intelligence
```

Creating subfolders

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

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

The following screenshot shows the preceding steps:

Figure 9.39 – Creating subfolders

Reducing model size by disabling auto date/time

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

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

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

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

The following screenshot shows the preceding steps:

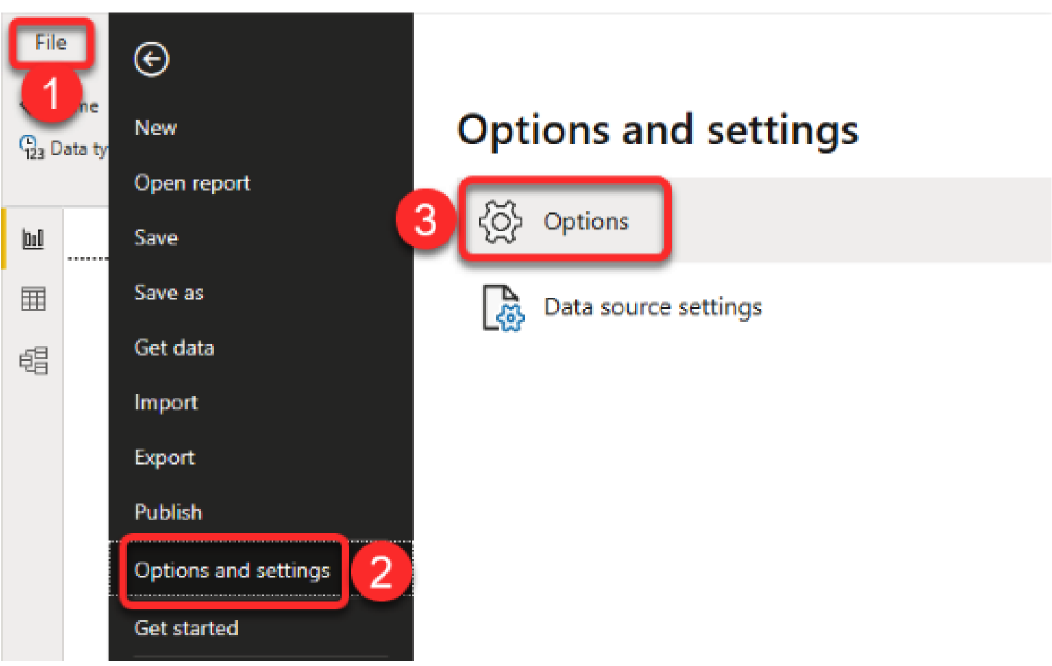


Figure 9.40 – Changing Power BI Desktop Options

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

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

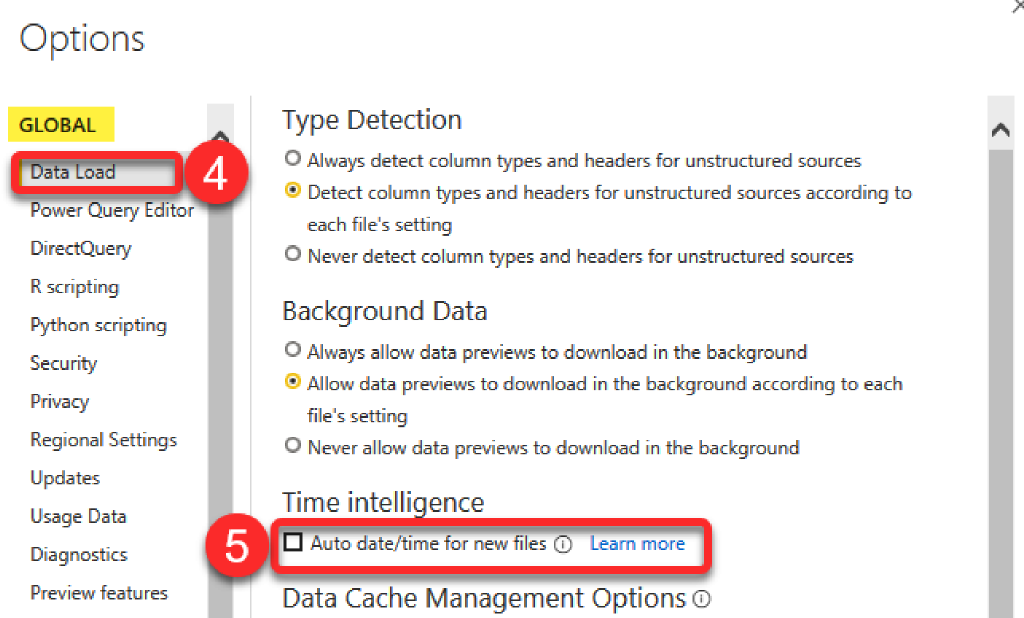


Figure 9.41 – Disabling the Auto date/time feature globally

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

8. Click **OK**:

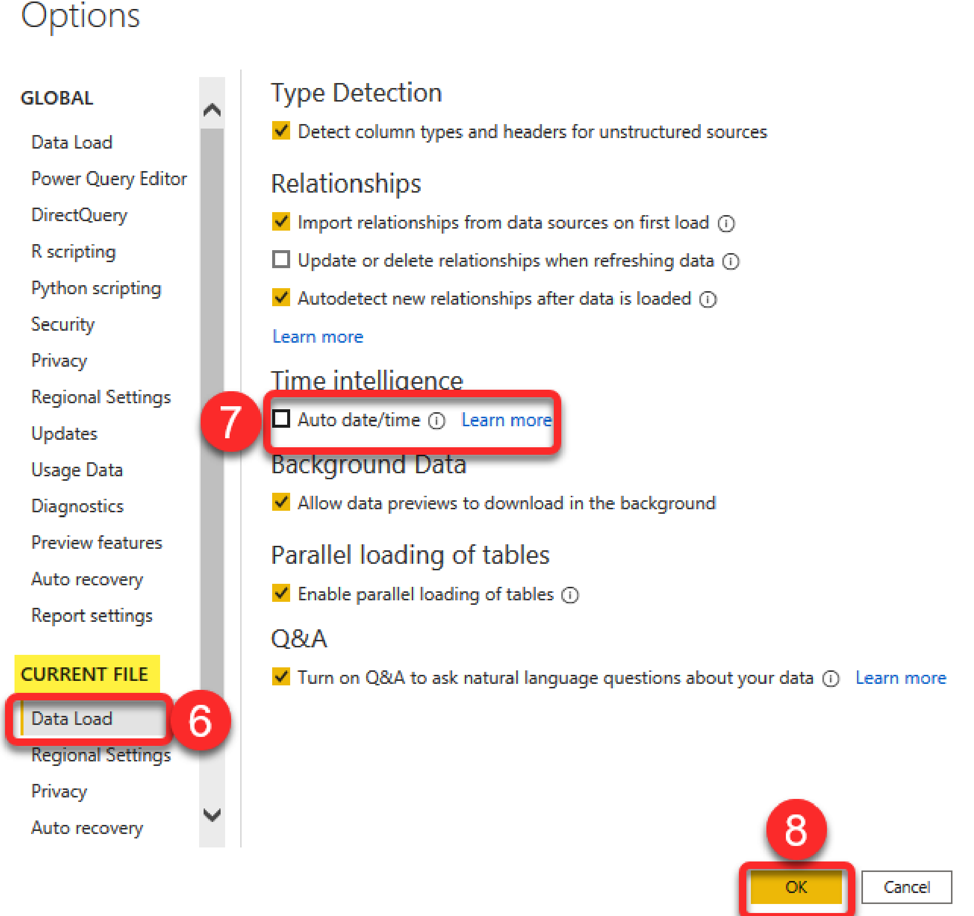


Figure 9.42 – Disabling the Auto date/time feature for the current file

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

Summary

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

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

Section 4: Advanced Data Modeling

This section focuses on advanced data modeling techniques that you may not deal with on a daily basis but are extremely important to know about. Implementing parent-child hierarchies, dealing with different calculations in a hierarchy, using aggregations, different ways to handle many-to-many relationships, along with more advanced business requirements to be implemented in the data model are all covered in different chapters of this section. You need to have a deep understanding of the star schema and DAX. Like other parts of this book, the chapters of this section are fully hands-on with real-world scenarios.

This section comprises the following chapters:

Chapter 10, Advanced Data Modeling Techniques

Chapter 11, Row-Level Security

Chapter 12, Extra Options and Features Available for Data Modeling

10 Advanced Data Modeling Techniques

In the previous chapter, we looked at some data modeling best practices, such as dealing with many-to-many relationships using bridge tables, dealing with inactive relationships, and how to programmatically enable them. We also learned how to use config tables, organize our data model, and reduce the model's size by disabling the Auto date/time feature in Power BI Desktop.

This chapter will discuss some advanced data modeling techniques that can help us deal with more complex scenarios more efficiently. Some techniques we'll discuss in this chapter only used to be available for the data models that were backed in a premium capacity. However, they are now available under the Power Pro licensing plan as well. In this chapter, we will cover the following topics:

- Using aggregations
- Incremental refresh
- Understanding parent-child hierarchies

- Implementing roleplaying dimensions
- Using calculation groups

As this chapter's name implies, we will be covering more advanced technical topics here; therefore, we will avoid explaining less advanced steps.

Using aggregations

From a data analytics viewpoint, the concept of aggregation tables has been around for a long time. The concept was widely used in SQL Server Analysis Service Multi-Dimensional. The way aggregation tables work is simple: we aggregate the data at a particular grain and make it available in the data model. Already aggregating the data that's available in the model usually translates into better performance at runtime. However, the aggregation typically happens at a different level of granularity. Therefore, we change the granularity of the base table. Now, you may be wondering, so what if the business needs to drill down to a lower granular level of data? The answer is that we need to keep the base table available in the data model. We aggregate the base table at a different granular level in a new table. Then, we implement a control mechanism to detect the level of granularity that the user is at. If the data is available at the aggregated level, the calculation happens in the aggregated table. When the user drills down to the lower grain, the control mechanism runs the calculations in the base table. This approach is an intricate design from a data modeling perspective, but it works perfectly if we get it right.

Implementing this aggregation is useful in almost all data models; it is even more valuable when one of our data sources is **big data**. For instance, we may have billions of rows hosted in **Azure Synapse Analytics**; then, it is utterly impossible to import all the data into the data model in Power BI Desktop. In those cases, using aggregation tables becomes inevitable. The good news is that there is a specific feature in Power BI Desktop that supports aggregation tables called **Manage aggregations**. When it comes to using the **Manage aggregations** feature, the data source of the base table must support **DirectQuery mode**.

> **Note**
>
> We discussed the various storage modes in *Chapter 4*, *Getting Data from Various Sources*, in the *Dataset storage modes* section.
>
> Refer to the following link to find a list of data sources that support DirectQuery: `https://docs.microsoft.com/en-us/power-bi/connect-data/power-bi-data-sources?WT.mc_id=?WT.mc_id=DP-MVP-5003466`.

The **Manage aggregations** feature takes care of the complexities of switching the calculations between the aggregated table and the base table when the user drills down to more granular details. In the next few sections, we'll discuss two different ways of implementing aggregation tables.

The first method is for implementing aggregation tables for the data sources that do not support **DirectQuery mode**, such as Excel files. Therefore, we need to implement the aggregation table and take care of the control mechanism to switch between the aggregated table and the base table when the user drills down to a higher grain of data.

The second method uses the **Manage aggregations** feature when the connection to the data source supports **DirectQuery mode**.

Implementing aggregations for non-DirectQuery data sources

In many real-world scenarios, our data sources may not support **DirectQuery mode**. Therefore, we cannot use the **Manage aggregations** feature in Power BI Desktop. However, we can manually implement aggregations by going through the following process:

1. Summarizing the table at a different grain.
2. Creating relationships between the new summary table and the dimensions at the summary grain.
3. Creating the desired measures (note that the new measures work at the summary grain).
4. Creating another set of new measures that control the level of grain selected by the user.
5. Hiding the summary table to make it transparent from the users.

In this section, we will use the `Chapter 10, Aggregations on Non-DirectQurey Data Sources.pbix` sample file, which is sourcing data from the `AdventureWorksDW2017.xlsx` Excel file.

Implementing aggregation at the Date level

In our sample file, we want to create an aggregated table on top of the `Internet Sales` table. The `Internet Sales` table currently contains `60,398` rows. We need to summarize the `Internet Sales` table at the `Date` grain only. We'll name the new summary table `Internet Sales Aggregated`.

> **Note**
>
> We discussed **Summarization** in *Chapter 5*, *Common Data Preparation Steps*, in the *Group by* section. Therefore, we will not explain how to summarize the `Internet Sales` table here.

Summarizing the Internet Sales table

To summarize the `Internet Sales` table at the `Date` granularity level, we must use the `Group by` functionality in Power Query Editor. We need to aggregate the following columns while using `SUM` as the aggregation operation:

- `Order Quantity`
- `Sales`
- `Tax`
- `Freight Costs`

These need to be grouped by the following key columns:

- `OrderDateKey`
- `DueDateKey`
- `ShipDateKey`

The following screenshot shows the **Group By** step within **Power Query Editor**:

Group By

Specify the columns to group by and one or more outputs.

Basic Advanced

OrderDateKey

DueDateKey

ShipDateKey

Add grouping

New column name	Operation	Column
Order Quantity	Sum	OrderQuantity
Sales	Sum	SalesAmount
Tax	Sum	TaxAmt
Freight Costs	Sum	Freight

Add aggregation

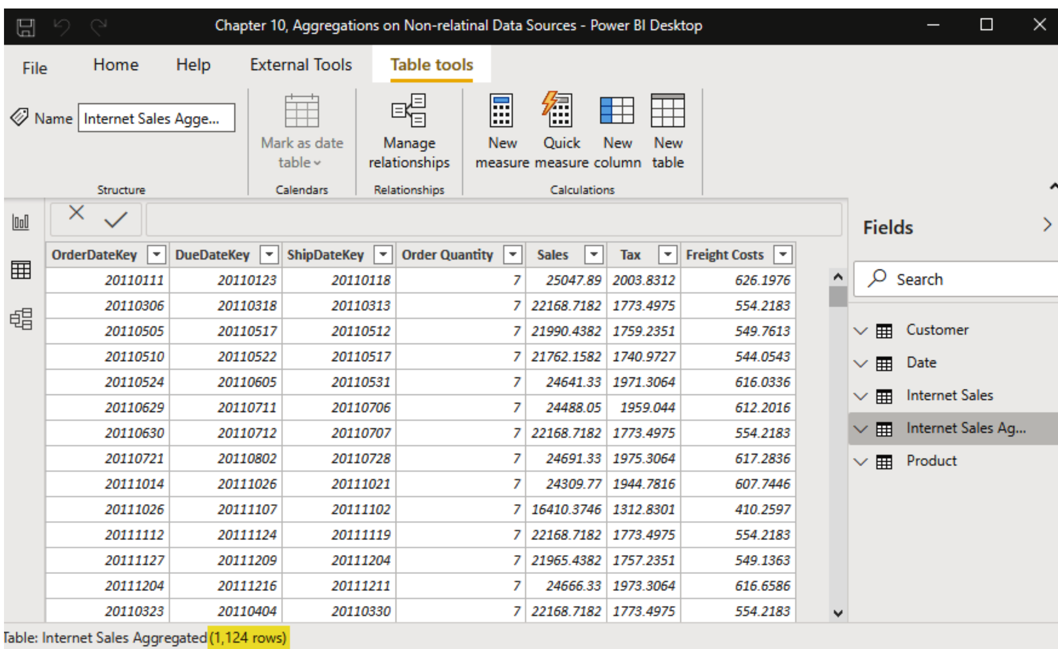
OK Cancel

Figure 10.1 – Summarizing Internet Sales in Power Query Editor with the Group By functionality

Now that we've summarized the `Internet Sales` table at the `Date` level (the new `Internet Sales Aggregated` table), we can close and apply the changes within Power Query Editor to load the data into the data model.

Creating relationships

At this point, we've loaded the data into the data model. The `Internet Sales Aggregated` table only contains `1,124` rows, which is significantly fewer than the `Internet Sales` table (the base table). The following screenshot shows the `Internet Sales Aggregated` table after being loaded into the data model:



OrderDateKey	DueDateKey	ShipDateKey	Order Quantity	Sales	Tax	Freight Costs
20110111	20110123	20110118	7	25047.89	2003.8312	626.1976
20110306	20110318	20110313	7	22168.7182	1773.4975	554.2183
20110505	20110517	20110512	7	21990.4382	1759.2351	549.7613
20110510	20110522	20110517	7	21762.1582	1740.9727	544.0543
20110524	20110605	20110531	7	24641.33	1971.3064	616.0336
20110629	20110711	20110706	7	24488.05	1959.044	612.2016
20110630	20110712	20110707	7	22168.7182	1773.4975	554.2183
20110721	20110802	20110728	7	24691.33	1975.3064	617.2836
20111014	20111026	20111021	7	24309.77	1944.7816	607.7446
20111026	20111107	20111102	7	16410.3746	1312.8301	410.2597
20111112	20111124	20111119	7	22168.7182	1773.4975	554.2183
20111127	20111209	20111204	7	21965.4382	1757.2351	549.1363
20111204	20111216	20111211	7	24666.33	1973.3064	616.6586
20110323	20110404	20110330	7	22168.7182	1773.4975	554.2183

Figure 10.2 – The Internet Sales Aggregated table

The next step is to create relationships between the new table and the `Date` table. The following screenshot shows the relationships that have been created:

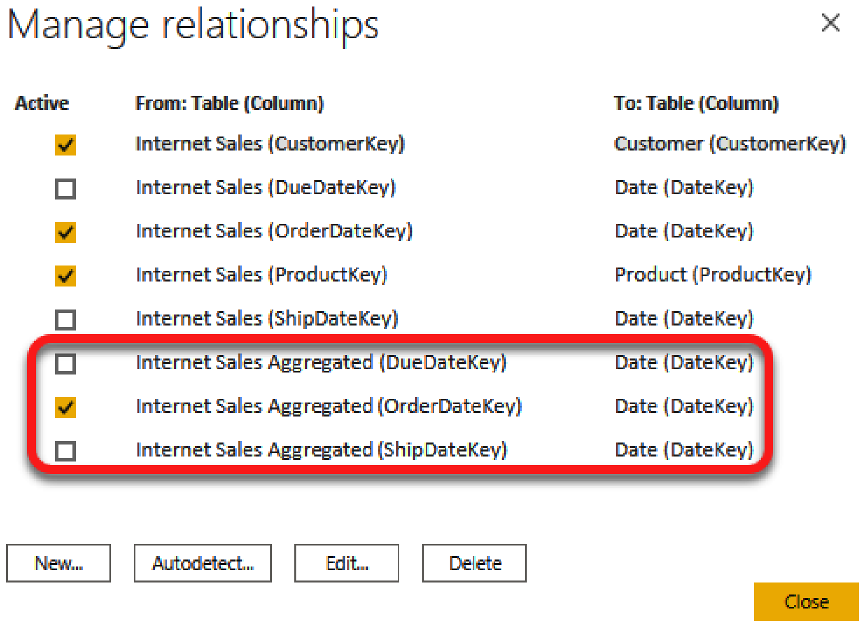


Figure 10.3 – Relationships between the Internet Sales Aggregated table and the Date table

The next step is to create a new measure to calculate the `Sales` amount.

Creating new measures in the summary table

Now, it is time to create a new measure within the `Internet Sales Aggregated` table. To keep this scenario simple, we'll create one new measure that calculates the summation of the `Sales` column from the `Internet Sales Aggregated` table. The DAX expression for this is as follows:

```
Sales Agg = SUM('Internet Sales Aggregated'[Sales])
```

The `Sales Agg` measure calculates the sales amount. Remember, the `Internet Sales Aggregated` table is derived from the `Internet Sales` table. Therefore, the `Sales Agg` measure uses the same calculation as the `Internet Sales` measure from the `Internet Sales` table. Here is the DAX expression of the `Internet Sales` measure:

```
Internet Sales = SUM('Internet Sales'[SalesAmount])
```

The main differences between the `Sales Agg` measure and the `Internet Sales` measure are as follows:

- `Sales Agg` is calculating the sales amount over a small table (`Internet Sales Aggregated`), while `Internet Sales` is calculating the same thing but over a much larger table (`Internet Sales`), so the `Sales Agg` calculation could be faster than `Internet Sales` when many concurrent users are using the report.

The users can filter the `Internet Sales` values by `Customer`, `Product`, and `Date`, but they can only filter the `Sales Agg` values by `Date`.

Now that we've created the `Sales Agg` measure, we are ready for the next step in our design: creating a new control measure to detect the level of grain selected by the user.

Creating control measures in the base table

In this section, we will create a new control measure to detect the grain that's been selected by the user. When a user is in `Date` grain only, the control measure redirects to the `Sales Agg` measure (from the summary table). However, if the user selects a column from either the `Product` or `Customer` tables, then the control measure redirects to the `Internet Sales` measure (from the base table). We create the control measure in the base table, which in our example is the `Internet Sales` table. We will be creating the control measure in the base table as we will be hiding the summary table in the next step. We have a couple of DAX options when it comes to identifying the grain that's been selected by the user. We can use either the `IF` or `SWITCH` function to check certain conditions, along with one or a combination of the following DAX functions:

- `ISFILTERED`
- `ISINSCOPE`
- `HASONEFILTER`
- `HASONEVALUE`

The following DAX expression detects if the user has selected a column from either the `Product` table or the `Customer` table and redirects the calculation to the corresponding measure:

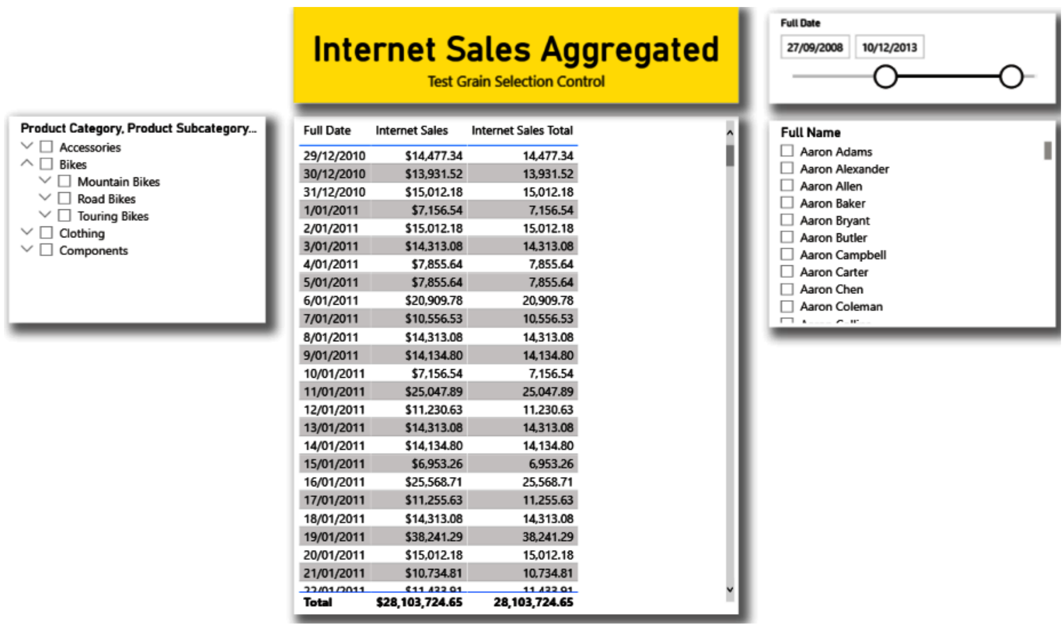
```
Internet Sales Total =
    IF(
        OR(ISFILTERED('Product'), ISFILTERED('Customer'))
        , [Internet Sales]
        , [Sales Agg]
        )
```

Hiding the summary table

The last piece of the puzzle is to hide the `Internet Sales Aggregated` table and the `Internet Sales` measure within the `Internet Sales` table to avoid confusion for other content creators. The following screenshot shows the model after hiding the `Internet Sales Aggregated` table and the `Internet Sales` measure from the `Internet Sales` table:

Figure 10.4 – The data model after hiding the Internet Sales Aggregated table and the Internet Sales measure

Now that we have everything sorted, it is time to test the solution in the data visualization layer. We need to make sure that the `Internet Sales` measure and the `Internet Sales Total` measure always result in the same values. We also need to ensure that when the user selects any columns from either the `Product` table or the `Customer` table, the `Internet Sales Total` measure uses the base table (the `Internet Sales` table). In our example, we created a report page that shows the `Internet Sales` and `Internet Sales Total` measures side by side. The following screenshot shows the report page:



Full Date	Internet Sales	Internet Sales Total
29/12/2010	$14,477.34	14,477.34
30/12/2010	$13,931.52	13,931.52
31/12/2010	$15,012.18	15,012.18
1/01/2011	$7,156.54	7,156.54
2/01/2011	$15,012.18	15,012.18
3/01/2011	$14,313.08	14,313.08
4/01/2011	$7,855.64	7,855.64
5/01/2011	$7,855.64	7,855.64
6/01/2011	$20,909.78	20,909.78
7/01/2011	$10,556.53	10,556.53
8/01/2011	$14,313.08	14,313.08
9/01/2011	$14,134.80	14,134.80
10/01/2011	$7,156.54	7,156.54
11/01/2011	$25,047.89	25,047.89
12/01/2011	$11,230.63	11,230.63
13/01/2011	$14,313.08	14,313.08
14/01/2011	$14,134.80	14,134.80
15/01/2011	$6,953.26	6,953.26
16/01/2011	$25,568.71	25,568.71
17/01/2011	$11,255.63	11,255.63
18/01/2011	$14,313.08	14,313.08
19/01/2011	$38,241.29	38,241.29
20/01/2011	$15,012.18	15,012.18
21/01/2011	$10,734.81	10,734.81
Total	$28,103,724.65	28,103,724.65

Figure 10.5 – Testing the aggregation in data visualization

As the preceding screenshot shows, there is a card visual at the top of the page. The card visual shows what table is used in the calculation within the table visual (in the middle of the report page). We used the following measure in the card visual:

```
Test Grain Selection Control =
        IF(
            OR(ISFILTERED('Product'), ISFILTERED('Customer'))
            , "Internet Sales"
            , "Internet Sales Aggregated"
            )
```

The preceding expression checks whether either the `Product` table or the `Customer` table have been filtered. If the result is `TRUE`, then the `Internet Sales` table is being used; otherwise, the `Internet Sales Aggregated` table is being sued.

The following screenshot shows the report page when the user selects a `Product Category` from the slicer on the left-hand side of the report:

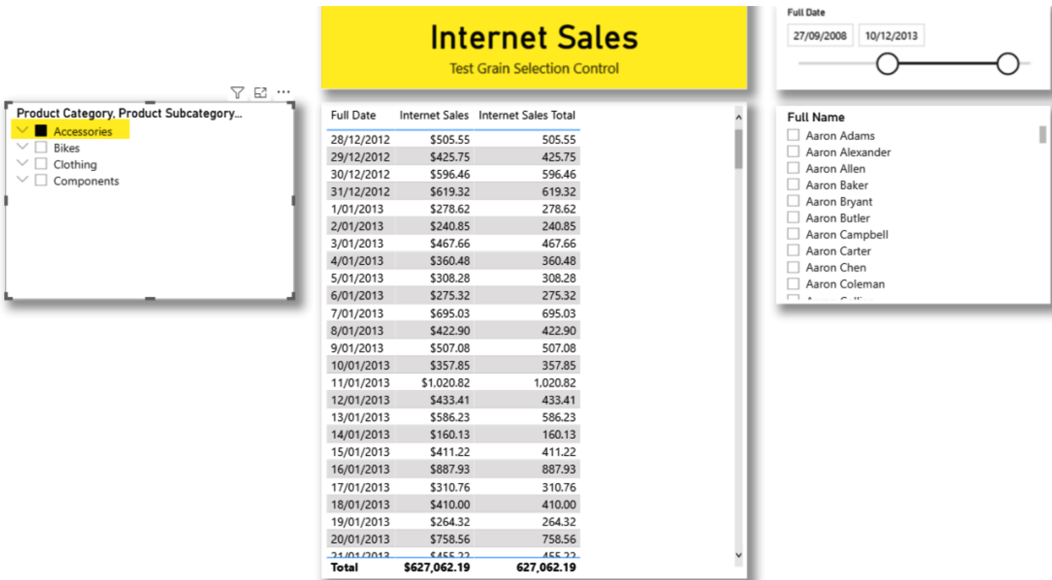


Figure 10.6 – The test visuals after selecting a Product Category from the slicer

As the preceding screenshot shows, the aggregation is working correctly.

So far, we've implemented a simple use case for implementing aggregations in Power BI Desktop. But there are many other use cases for aggregations. In the next section, we'll go one step further to elevate this scenario's complexity.

Implementing aggregation at the Year and Month level

This time, we would like to create another summary table that summarizes the `Internet Sales` table at the `Year` and `Month` granular levels and only based on the `OrderDateKey` column. We also want to include row counts of `Internet Sales` as an aggregated column. The steps for implementing this scenario are the same as in the previous scenario. The only difference is in the data preparation layer in terms of summarizing the `Internet Sales` data and changing the granularity of the data from `Date` to `Year` and `Month`. Let's have a closer look at the data preparation layer.
To summarize the `Internet Sales` data at the `Year` and `Month` level, we need to aggregate the following columns (with the mentioned aggregation operation):

- `SUM` of `Order Quantity`
- `SUM` of `Sales`
- `SUM` of `Tax`
- `SUM` of `Freight Costs`
- `Count` of `Internet Sales`

`Grouped By` the following key column:

- `OrderDateKey`

The critical point to notice is that we need to modify `OrderDateKey` so that it's at the `Year` and `Month` level. However, `OrderDateKey` is a `number` value, not a `date` value. The other point is that we need to create a relationship between the new summary table and the `Date` table using `OrderDateKey` from the summary table and `DateKey` from the `Date` table. Therefore, `OrderDateKey` must remain a number to match the values of the `DateKey` column from the `Date` table. To overcome this challenge, we only need to use the following math:

```
New OrderDateKey = (CONVERT(Integer, ([OrderDateKey]/100)) *
100) + 1
```

The preceding math converts `20130209` into `20130201`. At this point, you may be thinking that `New OrderDateKey` is still at the day level. That is correct. But we are changing all the dates to the first day of the month, which means that when we aggregate the values by `New OrderDateKey`, we are indeed aggregating at the `Year` and `Month` level. So, we only need to replace the `OrderDateKey` values using the `New OrderDateKey` math.

Once we've removed all the unnecessary columns within **Power Query Editor**, we have to add a step to replace the values of `OrderDateKey` with the following expression:

```
Table.ReplaceValue(#"Removed Other Columns",each
[OrderDateKey], each (Int64.From([OrderDateKey]/100) * 100) +
1,Replacer.ReplaceValue,{"OrderDateKey"})
```

The following screenshot shows these Power Query steps:

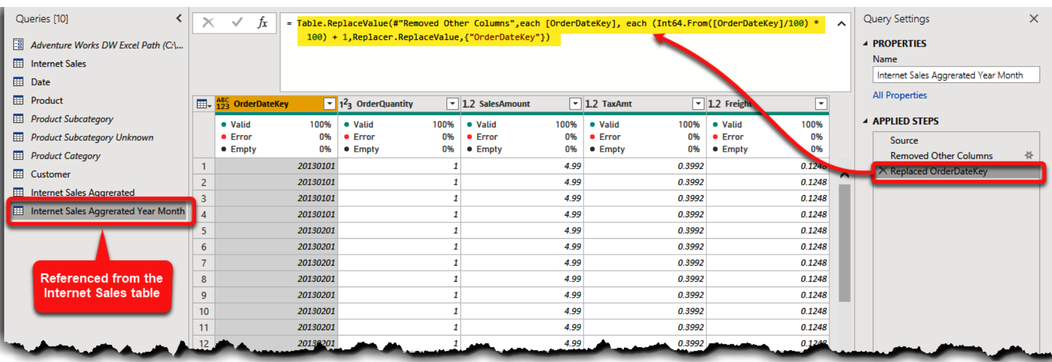


Figure 10.7 – Replacing the OrderDateKey values

The next step is to summarize the results. Remember, we also need to add `Count of Internet Sales` as an aggregation. The following screenshot shows the `Group By` step resulting in our summary table:

Group By

Specify the columns to group by and one or more outputs.

Basic   Advanced

OrderDateKey

Add grouping

New column name	Operation	Column
Order Quantity	Sum	OrderQuantity
Sales	Sum	SalesAmount
Tax	Sum	TaxAmt
Freight Costs	Sum	Freight
Internet Sales Count	Count Rows	

Add aggregation

OK   Cancel

Figure 10.8 – Summarizing the Internet Sales table at the Year and Month level

We can now apply the changes within the **Power Query Editor** window, which adds `Internet Sales Aggregated Year Month` as a new table. Moving forward with our implementation, we have to create a relationship between the `Internet Sales Aggregated Year Month` table and the `Date` table. The following screenshot shows the relationship once it's been created:

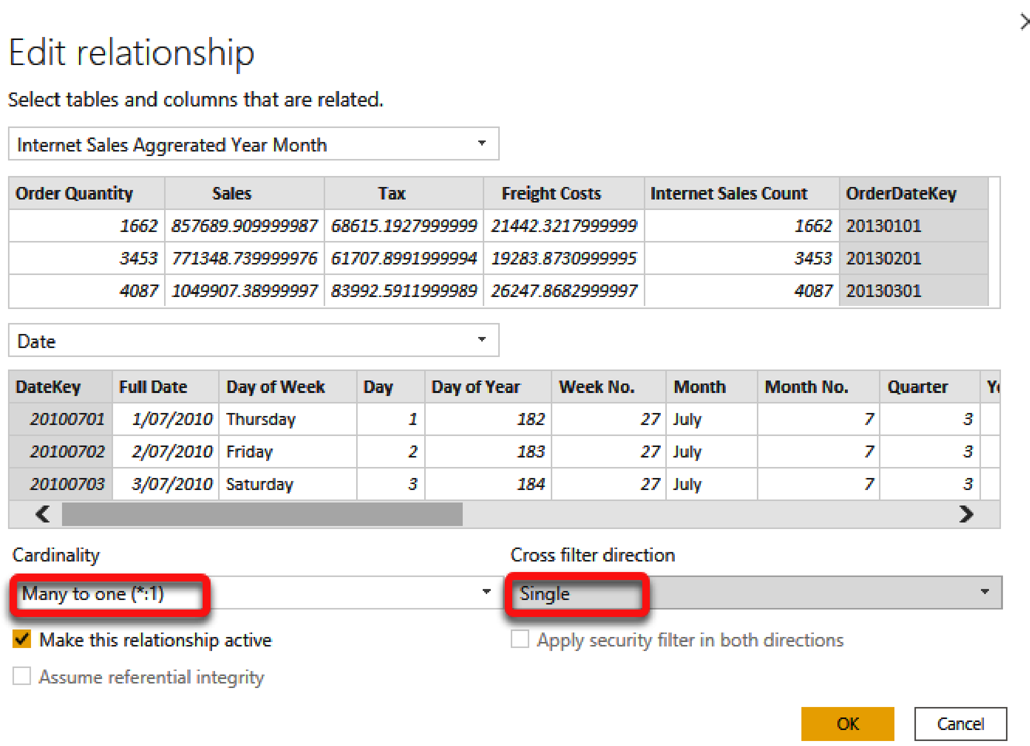


Figure 10.9 – Creating a relationship between Internet Sales Aggregated Year Month and Date

> **Note**
>
> We set the relationship's **Cardinality** to **Many to one** and **Cross-filter direction** to **Single**.

When we create the preceding relationship, Power BI automatically detects the relationship as a one-to-one relationship. Conceptually, this is correct. Each row of the `Internet Sales Aggregated Year Month` table is occurring on the first day of each month related to only one row in the `Date` table. However, we need to change the relationship's cardinality from one to one to many to one. This avoids filter propagation flowing from the `Internet Sales Aggregated Year Month` table to the `Date` table.

The next step is to create a new measure using the following DAX expression in the `Internet Sales Aggregated Year Month` table:

```
Sales Year Month = SUM('Internet Sales Aggregated Year
Month'[Sales])
```

Now, we need to create a new control measure to detect the level of details the user selects, not only at the `Product` and `Customer` levels but also at the `Date` level in the `Internet Sales` table. The following DAX expression caters for this:

```
Internet Sales Agg =
    IF(
        ISFILTERED('Product') || ISFILTERED('Customer') ||
ISFILTERED('Date'[Full Date])
        , [Internet Sales]
        , [Sales Year Month]
    )
```

Let's create a test measure that evaluates the `IF()` function in the preceding expression. The following DAX expression evaluates if either the `Product` and `Customer` tables or the `Full Date` column from the `Date` table are filtered, which results in `"Internet Sales"` (a text value); otherwise, the output is `"Internet Sales Aggregated"` (a text value). With the following test measure, we can get a better understanding of how the `Internet Sales Agg` measure works:

```
Internet Sales Agg Test =
    IF(
        ISFILTERED('Product') || ISFILTERED('Customer') ||
ISFILTERED('Date'[Full Date])
        , "Internet Sales"
        , "Internet Sales Aggregated"
    )
```

The last bit is to test the aggregation on a report page. The following screenshot shows the report page we created to test the aggregation results before the user selects any filters from the `Product` or `Customer` tables:

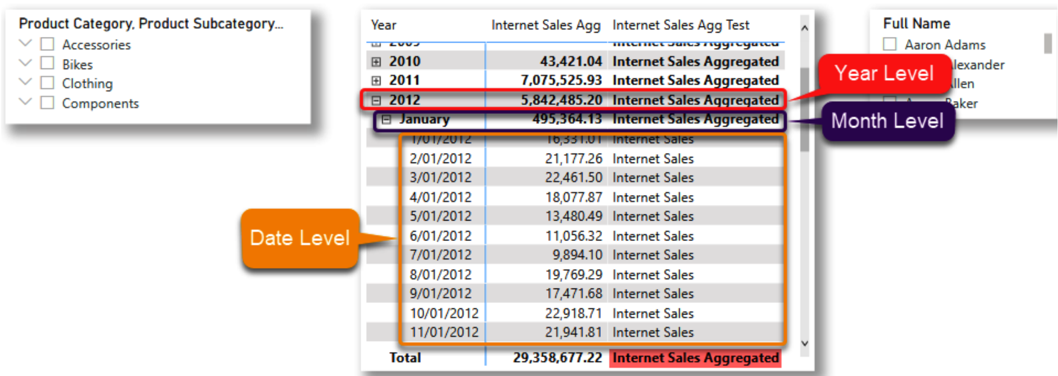


Figure 10.10 – Testing the aggregation before filtering by the Product or Customer tables

Note the value of `Total` for the `Internet Sales Agg Test` measure, which shows `Internet Sales Aggregated`. As the preceding screenshot shows, when we do not filter by `Product` or `Customer`, the `Internet Sales Agg` measure calculates the results for the `Year` and `Month` levels. But at the `Date` level, the `Internet Sales` measure calculates the values.

We can use this method as a performance optimization step, which mainly shows its values when many concurrent users use the same report. However, we have to be mindful of the following side effects:

- The summary tables will increase memory and storage consumption.
- The summary tables will also increase the data refresh time.
- The development time also increases as we need to create the summary table, create the relationships, and create the measures.

Now that we've implemented aggregations on top of non-DirectQuery data sources, it is time to look at the **Manage aggregations** feature.

Using the Manage Aggregations feature

Managing aggregations is the most significant scalability feature and is also one of the most powerful data modeling features available within Power BI Desktop, since it unlocks the usage of Big Data in Power BI. As we all know, Power BI compresses and caches data into memory when the query connection mode is **Import mode**. Imagine a scenario where we have a reasonably large data source that contains as billions of rows of data stored in an Azure Synapse server. Our job is to create an enterprise-grade analytical solution in Power BI. We also know that Power BI has a storage limit, especially when we are on either the Free or Pro licensing tier. Even if we own a Premium capacity, we are still limited to the amount of dedicated memory available in our Premium capacity. Of course, with Power BI Premium Gen 2, this memory limitation would be less of a concern. But if we are not careful about our data modeling, then our data processing and memory consumption will quickly become a bottleneck issue. This is where using the **Manage Aggregations** feature becomes vital to our data model. As we explained in the previous sections, the **Manage aggregations** feature is only available for the base tables that source from a data source supporting **DirectQuery mode**.

> **Note**
>
> In some Power BI resources, the base table is also referred to as the detail table. However, they are the same thing – the table that we are using to create a summary table.
>
> The summary table is also referred to as the aggregation table, aggregated table, or agg table.

The supporting **DirectQuery mode** is a vital point when it comes to data modeling. As we discussed previously in *Chapter 4*, *Getting Data From Various Sources*, in the *Connection modes* section, when a query is in **Import mode**, we cannot switch it to either **DirectQuery mode** or **Dual mode**. Therefore, we have to think about the necessity of managing aggregations at design time and before we start implementing the data model; otherwise, we will end up creating the data model from scratch, which is far from ideal. With that in mind, here is the process of managing aggregations in Power BI Desktop:

1. We connect to the data source.
2. We load all the necessary tables in **DirectQuery mode**:

- The data source may include the aggregation table already.
- If the summary table is not already in the data source, we can use either Power or DAX to create an aggregation table.

3. We create relationships between tables.
4. We change the storage mode of the summary table to **Import mode**. Again, we are mindful that this is an irreversible action. We must also change the storage mode of all the dimensions with an active relationship with the summary table to **Dual mode**.
5. We configure **Manage aggregation**.
6. We test the aggregation to make sure our queries hit the aggregation.

In this section, we will use AdventureWorksDW2019 (SQL Server database).

> **Note**
> You can download the SQL Server backup file from Microsoft's website: `https://docs.microsoft.com/en-us/sql/samples/adventureworks-install-configure?view=sql-server-ver15&tabs=ssms&WT.mc_id=?WT.mc_id=DP-MVP-5003466`.

Let's implement the aggregations on top of `FactInternetSales` using the following columns and aggregations:

- `FactInternetSales` (`Count Rows`)
- `SalesAmount` (`Sum`)
- `OrderQuantity` (`Sum`)
- `TaxAmt` (`Sum`)
- `Freight` (`Sum`)
- `ProductKey` (`Group by`)
- `Year` (`Group by`)
- `Month` (`Group by`)

To create the aggregation table, we must use the following T-SQL scripts to create a database view:

```
CREATE VIEW vw_Sales_Agg AS
SELECT dp.ProductKey
       , dd.CalendarYear
       , dd.EnglishMonthName
       , COUNT(1) Sales_Count
```

```
         , SUM(fis.SalesAmount)    AS Sales_Sum
         , SUM(fis.OrderQuantity) AS OrderQty_Sum
         , SUM(fis.TaxAmt)         AS Tax_Sum
         , SUM(fis.Freight)        AS Freight_Sum
FROM     FactInternetSales fis
         LEFT JOIN DimDate dd
                ON dd.DateKey = fis.OrderDateKey
         LEFT JOIN DimProduct dp
                ON fis.ProductKey = dp.ProductKey
GROUP BY dp.ProductKey
         , dd.CalendarYear
         , dd.EnglishMonthName
```

We can then use the preceding database view in Power BI Desktop.

Managing aggregations in Power BI Desktop for data sources that support DirectQuery and big data

Now that we have the aggregation table handy, it is time to use the **Manage aggregations** feature. The following steps show how to do so:

1. Use the SQL Server connection to connect to your instance of SQL Server:

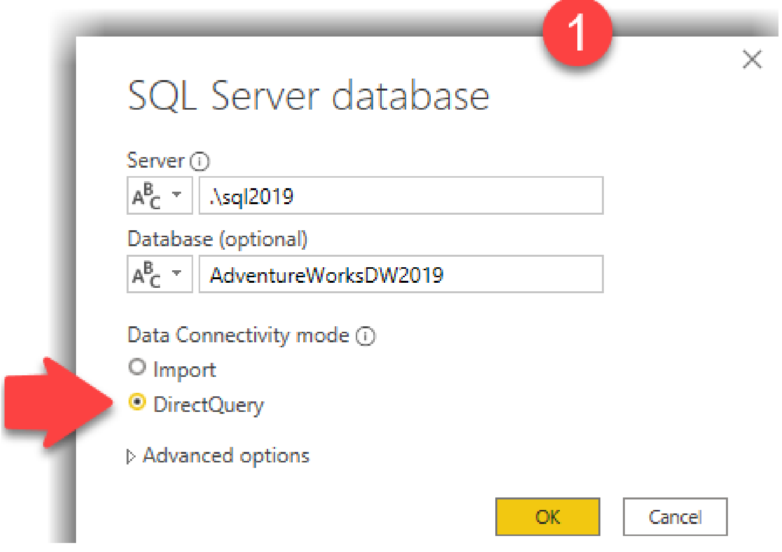


Figure 10.11 – Connecting to a SQL Server database in DirectQuery mode

2. Load the following tables in **DirectQuery mode**:

- `DimDate`
- `DimCustomer`
- `DimProduct`
- `FactInternetSales`
- `vw_Sales_Agg`:

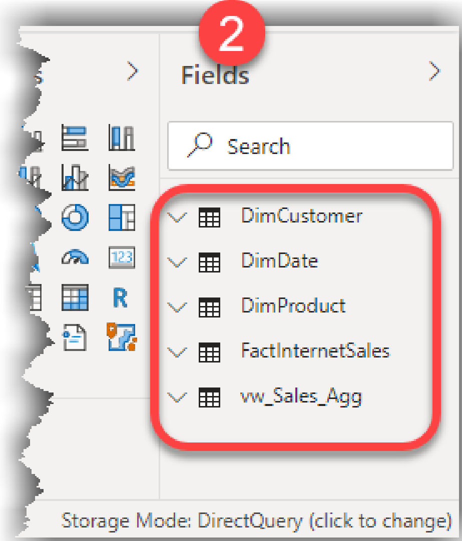


Figure 10.12 – Tables loaded in DirectQuery mode

> **Note**
> We could create the aggregation table in Power Query. However, for this sample, we use the `vw_Sales_Agg` database view.

3. Rename `vw_Sales_Agg` to `Sales_Agg`:

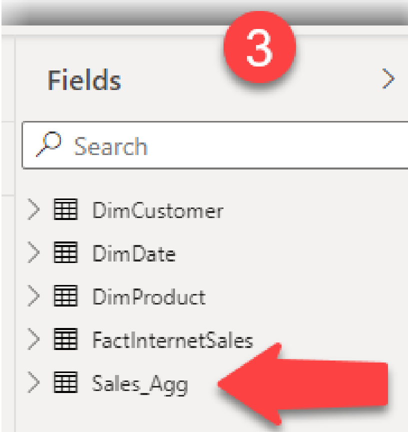


Figure 10.13 – Renaming vw_Sales_Agg to Sales_Agg

4. Create relationships between the tables:

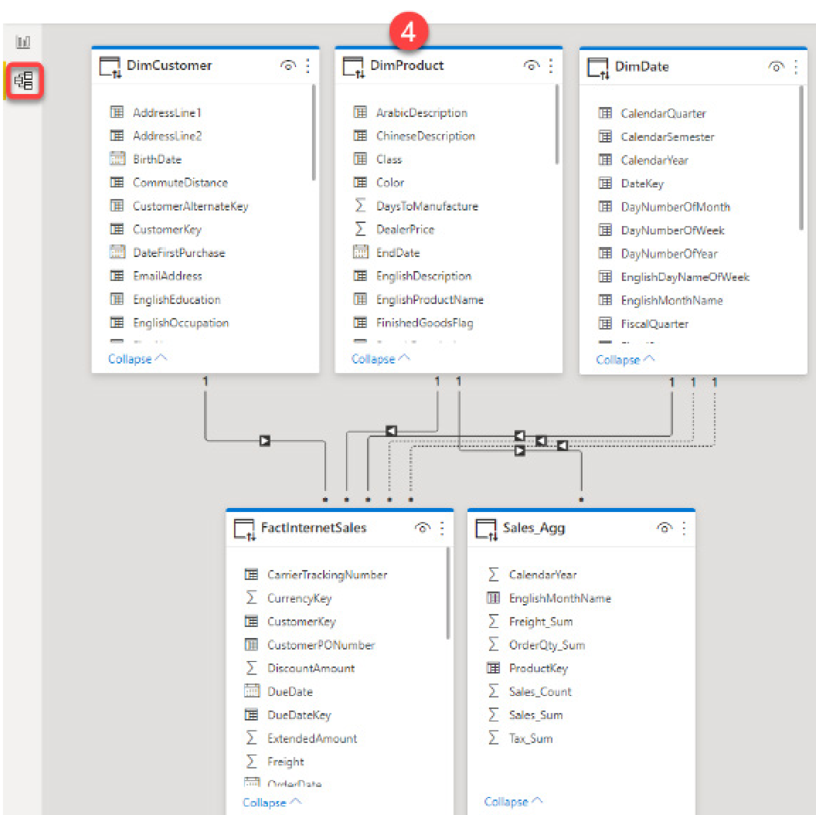


Figure 10.14 – Creating relationships in the Model view

> **Note**
>
> We need to create a relationship between the aggregation table and all the necessary dimensions to resolve the queries in the aggregation table. In our example, we have to create the relationship between the `Sales_Agg` and `DimProduct` tables via the `ProductKey` column.

5. Change the storage mode of the aggregation table and all its related dimensions. Follow these rules:

- Change the storage mode of the `Sales_Agg` table to **Import mode**. You will get a warning message, stating that the storage mode of the tables related to `Sales_Agg` must switch to **Dual mode**. We can tick the **Set the affected tables to dual** option here. Otherwise, we can set the storage mode of the related tables later.
- Change the storage mode of `DimProduct` to **Dual mode**:

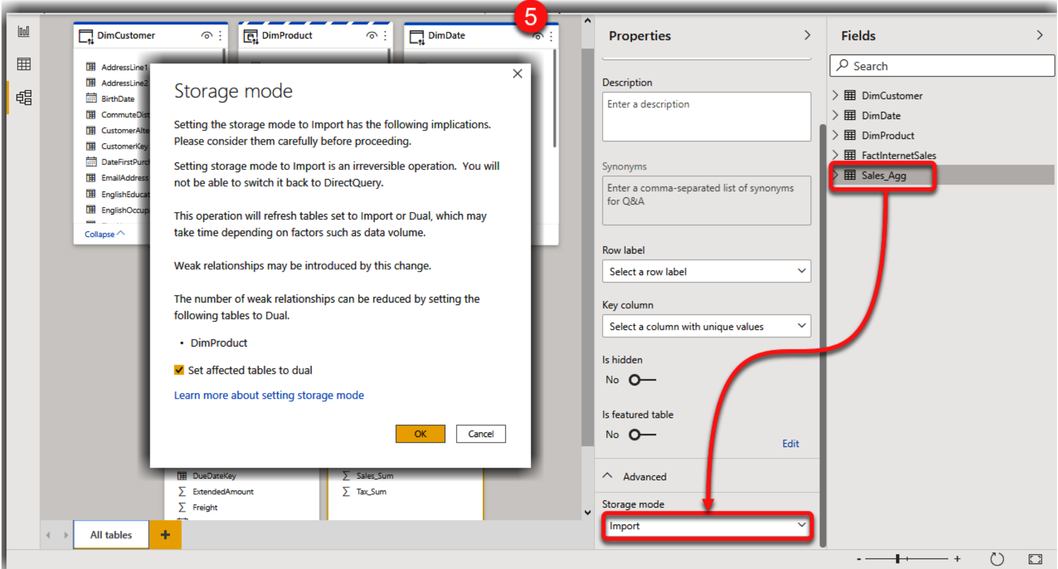


Figure 10.15 – Changing the storage mode of the aggregation table to Import and the related tables to Dual

We have changed the aggregation table's storage mode to **Import mode** because we want the queries that are hitting the aggregation table to run quickly. This is only possible when we import the data and make it available to the xVelocity engine so that it can resolve the queries internally. If we put a measure from `FactInternetSale` (the detail table) and `EnglishProductName` from the `DimProduct` table, then the `FactInternetSale` table is in **Import mode**, while the `DimProduct` table is in **DirectQuery mode**. The engine has to resolve a part of the query internally when it hits the aggregation table. The other part, which is the **DirectQuery**, must translate into the **Native Query** of the source system. For that reason, we must change the storage mode of the `DimProduct` table to **Dual mode**. Let's continue with the aggregation's configuration:

6. Right-click the `Sales_Agg` table and click **Manage aggregations**:

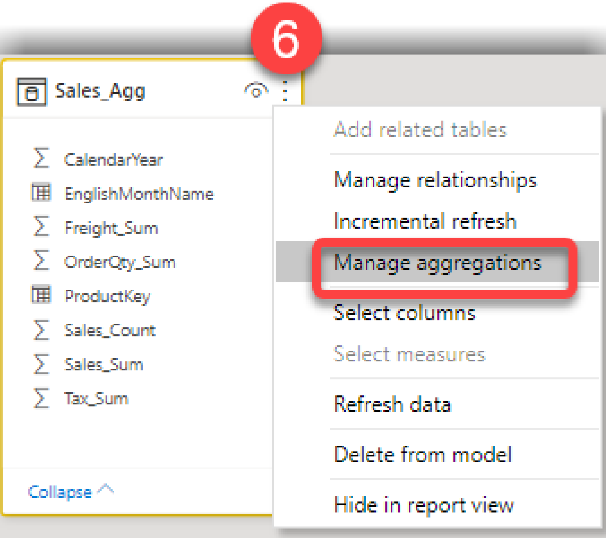


Figure 10.16 – Manage aggregations on the Sales_Agg table

7. In the **Manage aggregations** window, do the following:

- Step **a** – Make sure that the `Sales_Agg` table is selected as **Aggregation table**.
- Step **b** – Leave **Precedence** set to `0`.
- Step **c** – Set **Summarization** for the `Sales_Agg` columns. We deliberately named the columns with their summarization prefix, which will come in handy here.

- Step **d** – Click **Apply all**:

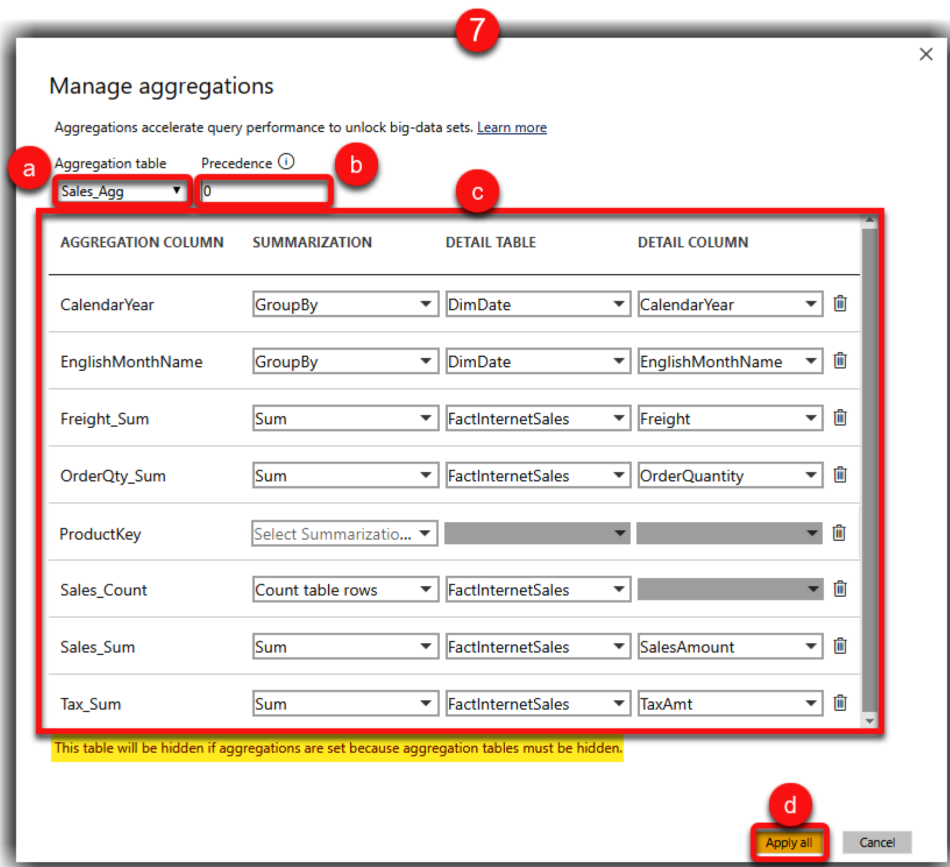


Figure 10.17 – Manage aggregations

> **Notes**
>
> We do not need to set **Summarization** of `ProductKey` to `Group by`. There is already a relationship between the `DimProduct` and `Sales_Agg` tables.
>
> After applying the aggregation, the `Sales_Agg` table is hidden.

8. Now, we must create the following measure in the detail table (`FactInternetSales`):

```
Total Internet Sales = SUM(FactInternetSales[SalesAmount])
```

So far, we've configured the aggregation on top of `FactInternetSales`. We also created a new measure. We need to test the aggregation and make sure that the aggregation table gets hit when the queries are at the `Product`, `Year`, or `Month` levels.

Testing the aggregation

At this stage, we have to test the aggregation and make sure it is getting hit. We can use **SQL Server Profiler** connected to our Power BI Desktop model to monitor the model. **SQL Server Profiler** is a part of the **SQL Server Management Studio** (**SSMS**) installation package.

> **Read More about SQL Server Profiler**
>
> `https://docs.microsoft.com/en-us/sql/tools/sql-server-profiler/sql-server-profiler?view=sql-server-ver15&WT.mc_id=5003466`.

We have the following two options in the way we use **SQL Server Profiler**:

- We can register **SQL Server Profiler** as an external tool. Here, we can open **SQL Server Profiler** directly from Power BI Desktop from the **External Tools** tab. With this method, **SQL Server Profiler** automatically connects to our Power BI data model via the Power BI Diagnostic Port.

> **Read More About How to Register SQL Server Profiler in Power BI:**
>
> `https://www.biinsight.com/quick-tips-registering-sql-server-profiler-as-an-external-tool/`.

- We can open **SQL Server Profiler** and manually connect to our Power BI data model through the Power BI Desktop Diagnostic Port. To do so, we must find the Diagnostic Port number first, then connect to the data model using the port number.

> **Learn More About the Different Ways to Find the Power BI Desktop Diagnostic Port:**
>
> `https://www.biinsight.com/four-different-ways-to-find-your-power-bi-desktop-local-port-number/`.

We use the first option as it is simpler than the second option. We have to trace the following events within the SQL Server Profiler:

If we're using SSMS v17.9 (or later), then we must select the following events:

- Query Processing\`Aggregate Table Rewrite Query`
- Query Processing\`Direct Query Begin`
- Query Processing\`Vertipaq SE Query Begin`

For the older versions of SSMS, we must select the following events:

- Queries Events\`Query Begin`
- Query Processing\`DirectQuery Begin`
- Query Processing\`Vertipaq SE Query Begin`

In this section, we'll discuss the first option, as follows:

1. Click **SQL Server Profiler** from the **External Tools** tab of the ribbon to open the Profiler, which automatically connects to the data model via **Diagnostic Port**:

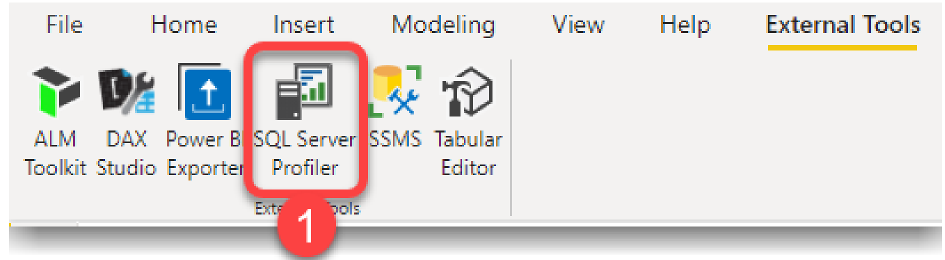


Figure 10.18 – Running SQL Server Profiler from External Tools

2. From the **Trace Properties** window in **SQL Server Profiler**, click the **Events Selection** tab.
3. Select the events we mentioned earlier under the **Query Processing** section.
4. Click **Run**, as shown in the following screenshot:

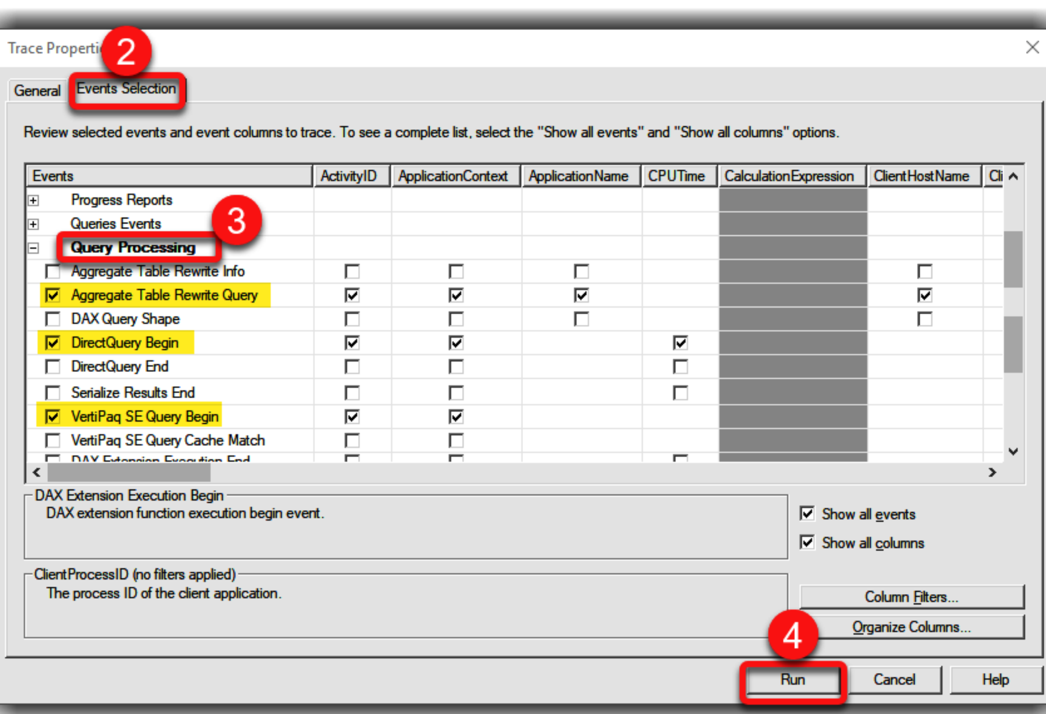


Figure 10.19 – Selecting events from Trace Properties in SQL Server Profiler

5. Go back to Power BI Desktop, put a Matrix visual on the report page, and put the `Total Internet Sales` measure into **Values**:

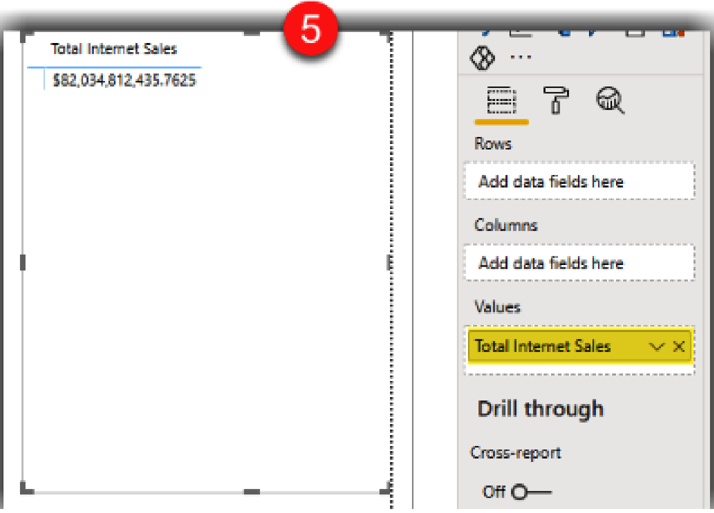


Figure 10.20 – Adding Total Internet Sales to a Matrix visual

6. Go back to **SQL Server Profiler** and click the `Aggregate Table Rewrite Query` event.
7. Check the output. If the aggregation table got hit, we will see `"matchingResult": "matchFound"` in the output results. Another indication that the query hit the aggregation table is that the `DirectQuery Begin` event does not show up:

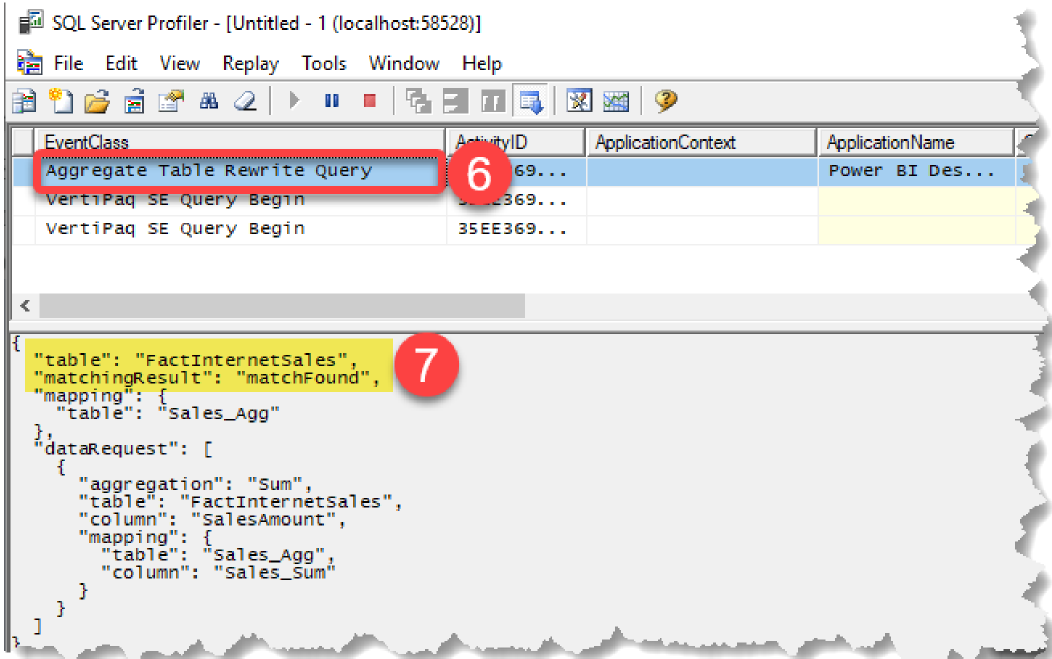


Figure 10.21 – Using SQL Server Profiler to check if the aggregation table gets hit

8. Go back to Power BI Desktop and add `EnglishProductName` from `DimProduct` to the Rows of the Matrix visual. Suppose you go back to **SQL Server Profiler** and click the `Aggregate Table Rewrite Query` event. In that case, you will see that the aggregation table got hit again.

The following image shows both the **Power BI Desktop** and **SQL Server Profiler** windows side by side:

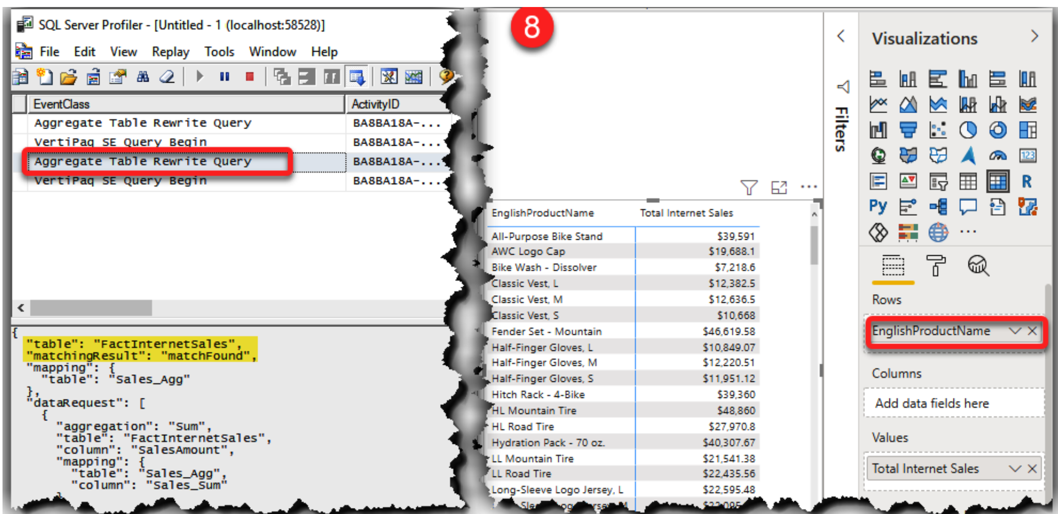


Figure 10.22 –The Sales_Agg table gets hit again after adding the EnglishProductName column from DimProduct

9. Add the `CalendarYear` and `EnglishMonthName` columns from the `DimDate` table to the Columns of the Matrix visual and check the Profiler to ensure it gets hit. Note that in the Matrix visual, we must drill down from `CalendarYear` to the `EnglishMonthName` level to get the month level:

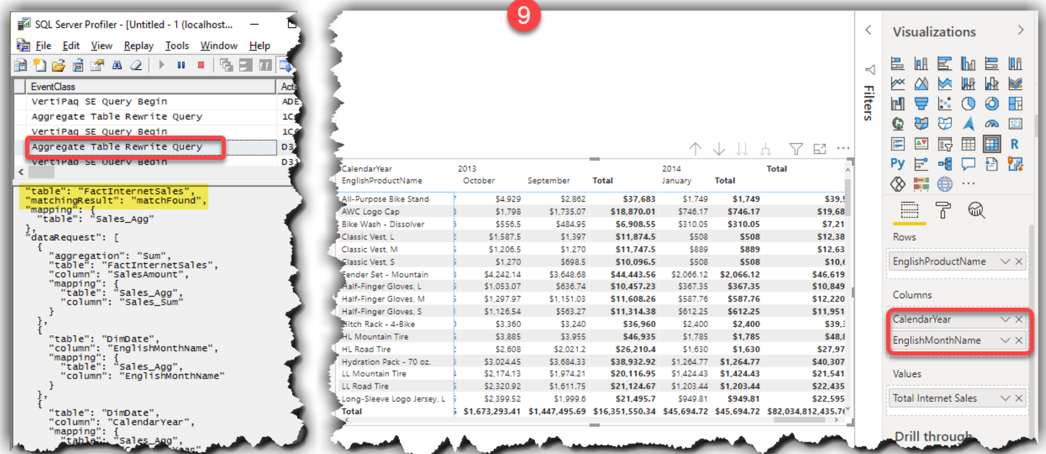


Figure 10.23 – The Sales_Agg table gets hit after adding the CalendarYear and EnglishMonthName columns

Now, let's take this one step further and make a simple change to our model and see if it affects the aggregation hit.

10. In the Matrix visual, drill up to the `CalendarYear` level. Then, in `DimDate`, set `Sort By` for the `EnglishMonthName` column to `MonthNumberOfYear`. From the Matrix, drill down to the `EnglishMonthName` level. Now, check the Profiler to see if `Sales_Agg` gets hit again. As the following image shows, this time, the result is `"matchingResult": "attemptFailed"`, which means that `Sales_Agg` did not get hit:

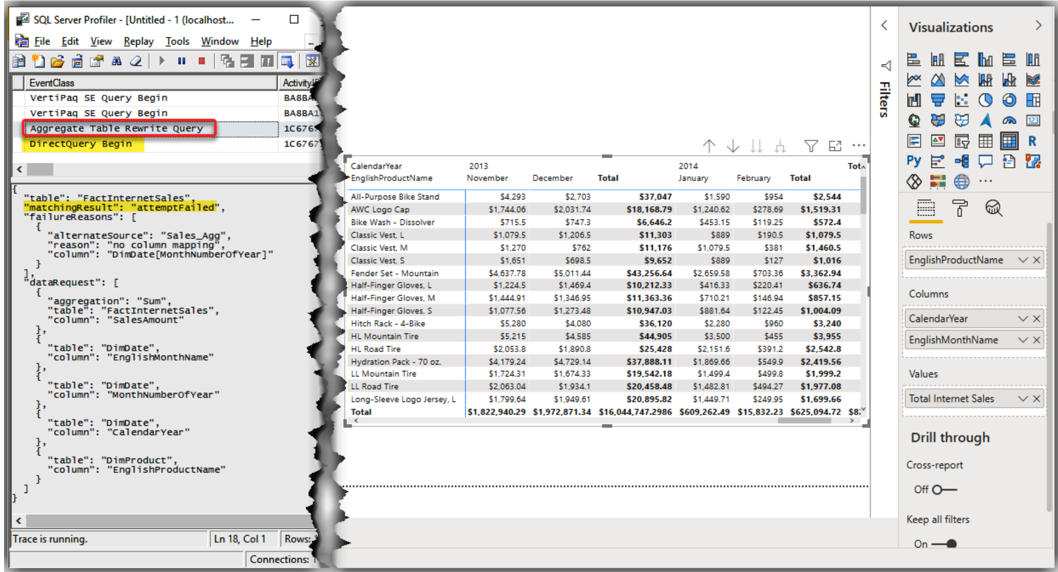


Figure 10.24 – The Sales_Agg table did not hit after sorting EnglishMonthName by MonthNumberOfYear

Another indication that `Sales_Agg` did not get hit is the appearance of the DirectQuery Begin event.

To fix this issue, we must go through the following steps:

1. Add the `MonthNumberOfYear` column to `Sales_Agg` in the SQL view.
2. Refresh the `Sales_Agg` table from the **Model** view.
3. Configure manage aggregation to `Group by` the `MonthNumberOfYear` column.

We leave this part to you.

> **Important Notes About Aggregations**
>
> The detail table (base table) must be in DirectQuery mode.
>
> **Precedence** in the **Manage aggregations** window prioritizes the aggregation hits when we have more than one aggregation table. The bigger the Precedence number, the higher the priority. For example, if we have another aggregation table that we would like to get hit before `Sales_Agg`, we must set a larger Precedence number for that aggregation table. The xVelocity engine tries to solve the underlying aggregation queries based on their Precedence value.
>
> The data types of the aggregation columns from the aggregation table and the corresponding columns from the detail table must be the same; otherwise, we cannot complete the **Manage aggregations**.
>
> We cannot chain aggregations for three or more tables. For example, we cannot create aggregations when `Table X` is the aggregation table for `Table Y`, and when `Table Y` is the aggregation table for `Table Z`.
>
> While configuring the **Manage aggregations** window, we should set `Summarization` for each aggregation column. Use the Count table rows for the aggregations showing the count rows of the detail table.
>
> Be mindful that after applying the aggregations, the aggregated table gets hidden by default.
>
> Always create measures in the detail table, not in the aggregation table. The aggregation table is meant to be hidden from the end user.
>
> We do not need to `Group By` the aggregation columns from a table with an active relationship with the aggregation table.
>
> The aggregation for inactive relationships is not supported, even if we use the `USERELATIONSHIP()` function to activate the relationship.

Incremental refresh

Incremental refresh was available only in Premium capacities. However, from February 2020 onward, it is also supported in the Professional licensing tier. Incremental refresh refers to incremental data loading, which has been around for a long time. Let's see what incremental refresh (or incremental data loading) is.

From a technical standpoint in data movement, there are always two options when we transfer data from location `A` to location `B`:

- **Truncation and load**: We transfer the data as a whole from location `A` to location `B`. If location `B` contains some data already, we truncate location `B` entirely and reload the entire data from location `A` to `B`.
- **Incremental load**: We transfer the data as a whole from location `A` to location `B` just once for the first time. The next time, we only load the data changes from `A` to `B`. In this approach, we never truncate `B`. Instead, we only transfer the data that exists in `A` but not in `B`.

When we refresh the data in Power BI, if we have not configured an incremental refresh, we use the first approach, which is truncation and load. Needless to say that in Power BI, any of the preceding data refresh methods only apply to tables with Import or Dual storage modes.

Once we've successfully configured the incremental refresh policies in Power BI, we always have two ranges of data: the *historical range* and the *incremental range*. The *historical range* includes all the data that has been processed in the past, while the *incremental range* is the current range of data to process. Incremental refresh in Power BI always looks for data changes in the *incremental range*, not the *historical range*. Therefore, changes in historical data will not be noticed.

> **Note**
>
> Incremental refresh detects updated data as deleting the old data and inserting new data.

Configuring incremental refresh is beneficial for large tables with millions of rows.

The following are some benefits of incremental refresh in Power BI:

- The data will refresh much faster because we only transfer the changes, not the entire data.
- The data refresh process is less resource-intensive than refreshing the entire data all the time.
- The data refresh process is less expensive and more maintainable than the non-incremental refreshes over large tables.

Incremental refresh is inevitable when we're dealing with massive datasets with billions of rows that do not fit into our data model in Power BI Desktop. Remember, Power BI is an in-memory data processing platform; therefore, it is improbable that our local machine can handle importing billions of rows into memory.

Now that we understand what incremental refresh is, let's see how it works in Power BI.

Configuring incremental refresh in Power BI Desktop

We always configure incremental refresh in Power BI Desktop. Once we've published the model to Power BI Service, the first data refresh takes longer. We transfer all the data from the data source(s) to Power BI Service for the first time. All future data refreshes will be incremental. The way incremental refresh works in Power BI Desktop is simple. First, we require to define two `date` or `date/time` parameters in Power Query Editor, `RangeStart` and `RangeEnd`, which are reserved for defining incremental refresh policies.

> **Note**
>
> Power Query is case sensitive, so the names of the parameters must be `RangeStart` and `RangeEnd`.

The next step is to filter the data in large tables by filtering a `date` or `date/time` column using the `RangeStart` and `RangeEnd` parameters, when the value of the `date/time` column is between `RangeStart` and `RangeEnd`.

> **Note**
>
> The `date` or `date/time` values must have an equal to (=) sign either on `RangeStart` or `RangeEnd`, not both.

The filter on the `date` or `date/time` column will be used to partition the data into ranges. With the preceding conditions in mind, let's implement incremental refresh. In this section, we will be using the `Chapter 10, Incremental Refresh.pbix` sample file, which sources the data from a restored backup of the `AdventureWorksDW2019` SQL Server database.

> **Note**
>
> You can download the SQL Server backup file from here: `https://docs.microsoft.com/en-us/sql/samples/adventureworks-install-configure?view=sql-server-ver15&tabs=ssms&WT.mc_id=?WT.mc_id=DP-MVP-5003466`.

Here is our scenario for an incremental refresh: we would like to have a refresh policy that stores 10 years of data plus the data up to the current date, and then incrementally refreshes 1 months' worth of data. Follow these steps to implement the preceding scenario:

1. In **Power Query Editor**, get data from the `FactInternetSales` table from `AdventureWorksDW2019` from SQL Server and rename it `Internet Sales`:

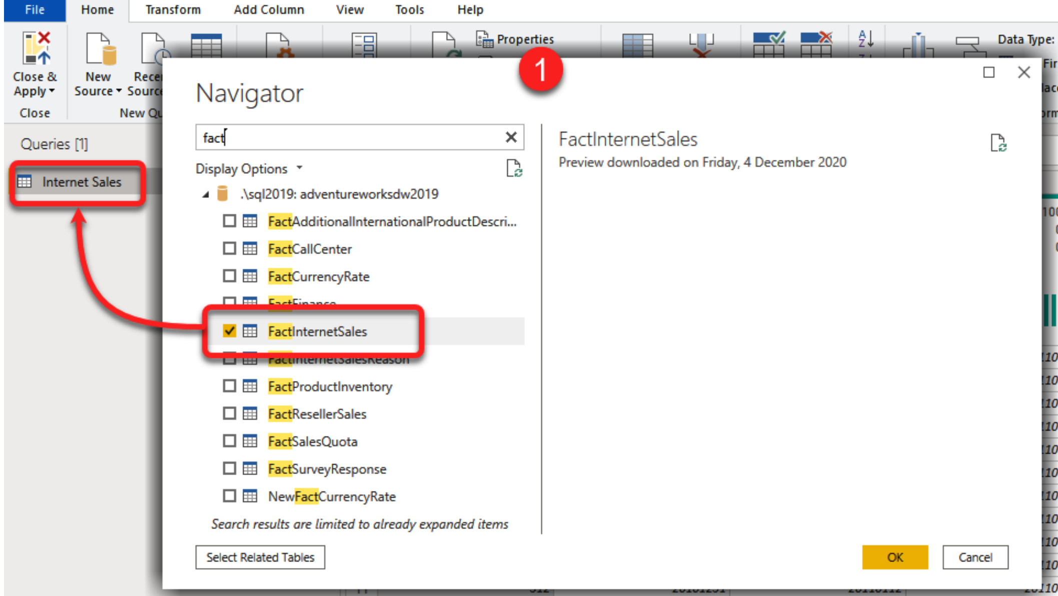


Figure 10.25 – Getting data from the source

2. Define the `RangeStart` and `RangeEnd` parameters with the `Date/Time` type. As we mentioned earlier, `RangeStart` and `RangeEnd` are reserved for configuring incremental refresh. So, the parameter names must match the preceding names. Set `Current Value` of the parameters as follows:

- `Current Value` of `RangeStart`: `1/12/2010 12:00:00 AM`

- `Current Value` of `RangeEnd`: `31/12/2010 12:00:00 AM`:

> **Note**
>
> Set a `Current Value` for the parameters that work for your scenario. Just keep in mind that these values are only useful at development time as the `Internet Sales` table will only include the values between `Current Value` of `RangeStart` and `RangeEnd` after defining the filter in the next steps. In our example, `Current Value` of `RangeStart` is the first day of the month for our table's first transaction. The first transaction in `Internet Sales` for the `OrderDate` column is on `29/12/2010`. Therefore, `Current Value` of the `RangeStart` parameter is `1/12/2010`. `Current Value` of `RangeEnd` in our example is `31/12/2010`.

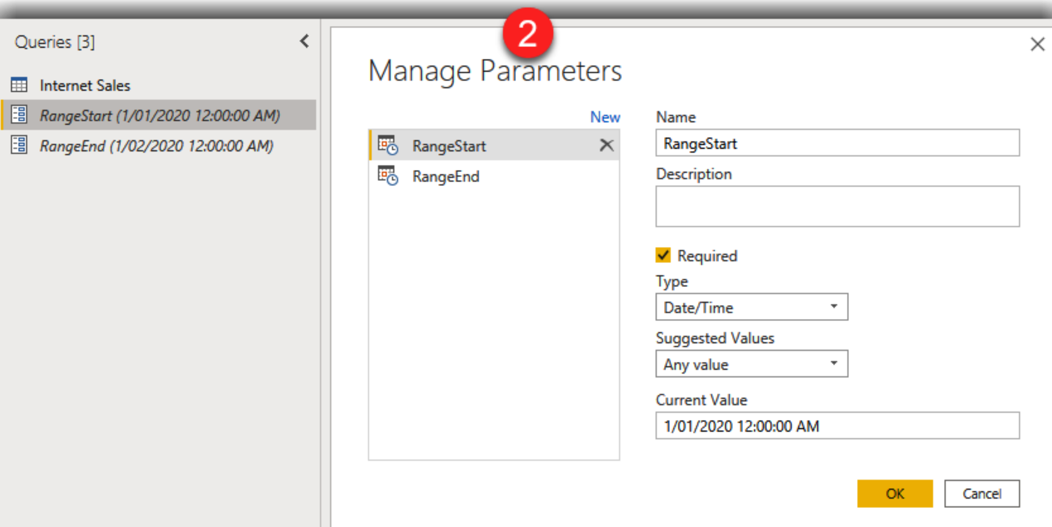


Figure 10.26 – Defining the RangeStart and RangeEnd parameters

3. Filter the `OrderDate` column, as shown in the following screenshot:

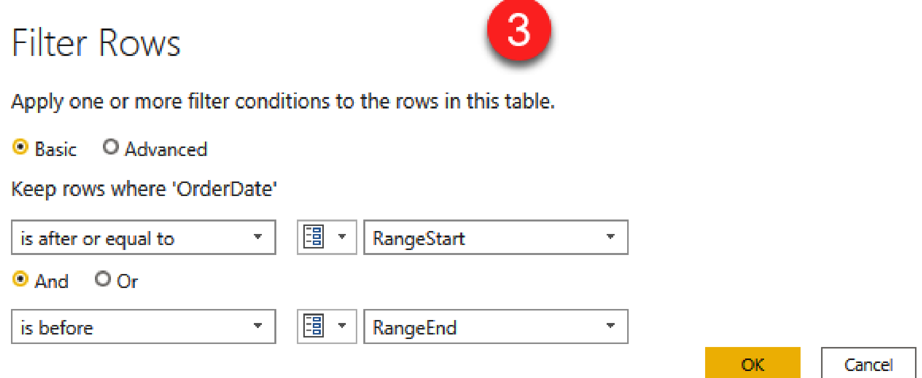


Figure 27 – Filtering the OrderDate column by the RangeStart and RangeEnd parameters

4. Click the **Close & Apply** button to import the data into the data model:

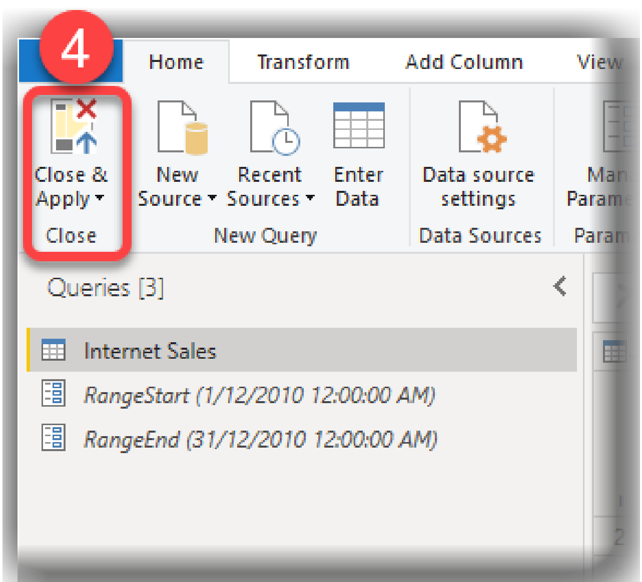


Figure 10.28 – Appling changes and loading data into the data model

5. Right-click the `Internet Sales` table and click **Incremental refresh**. The **Incremental refresh** option is available in the context menu of the **Report** view, **Data** view, or **Model** view:

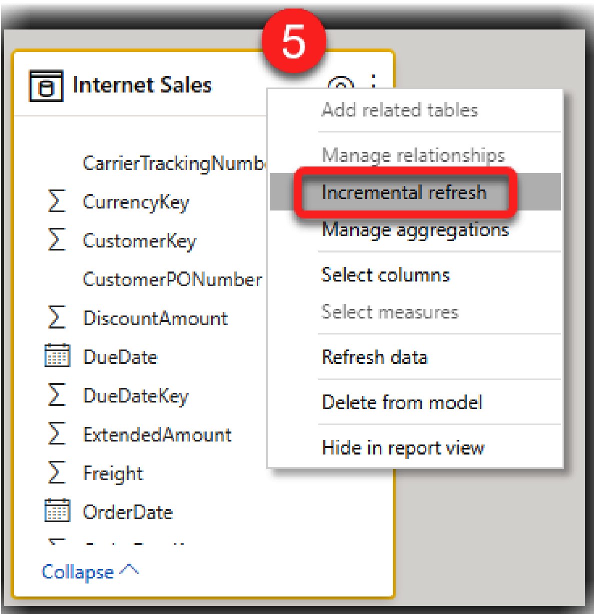


Figure 10.29 – Selecting Incremental refresh from the context menu

6. In the **Incremental refresh** window, do the following:

   a. Toggle on **Incremental refresh**

   b. Set the **Store rows where column "OrderDate" is in the last:** setting to `10 Years`

   c. Set the **Refresh rows where column "OrderDate" is in the last:** setting to `1 Month`

   d. Leave the **Detect data changes** and **Only refresh complete month** options unticked

e. Click **Apply all**:

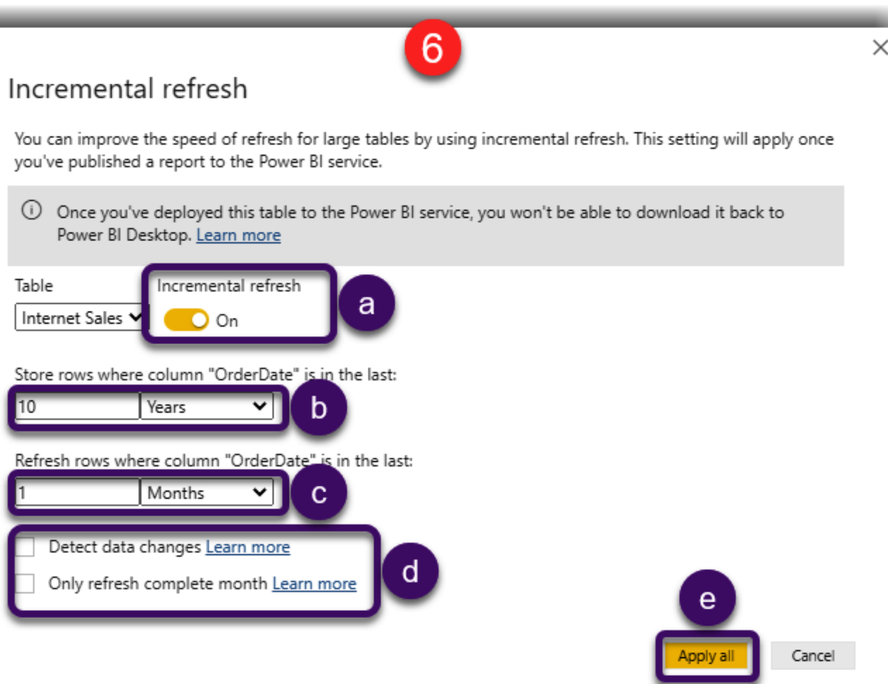


Figure 10.30 – Configuring the Incremental refresh window

Note

In many data integration and data warehousing processes, some auditing columns are added to the tables that collect some useful metadata, such as `Last Modified At`, `Last Modified By`, `Activity`, `Is Processed`, and so on. If you have a `Date/Time` column indicating the data changes (such as `Last Modified At`), the `Detect data changes` option would be helpful. We do not have any auditing columns in our data source; therefore, we will leave it unticked.

The **Only refresh complete month** option depends on the period we selected in the **Refresh rows where column "OrderDate" is in the last:** setting (item **C** in the preceding screenshot). With this option, we can force the incremental refresh to happen only for the entire period. In our scenario, this option is not useful; hence, we have left it unticked.

7. Click the **Publish** button to publish the data model to Power BI Service. Here, select the desired workspace and click **Select**, as shown in the following screenshot:

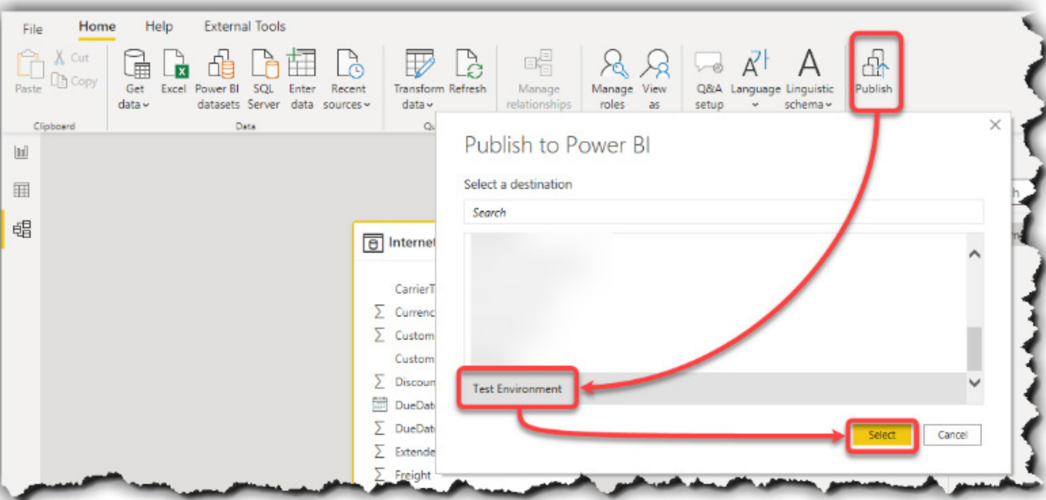


Figure 10.31 – Publishing the report to Power BI Service

So far, we've configured the incremental refresh and published the data model to Power BI Service. At this point, a Power BI administrator should take over this process to complete the **Schedule refresh**, which they do by setting up an **On-premises Data Gateway**, passing sufficient credentials, and more. These settings are outside the scope of this book, so we leave them to you. So, let's assume the Power BI administrators have completed the settings in Power BI Service. You may now be wondering how we can test that the incremental refresh is working. In the next section, we'll explain how to test the incremental refresh.

Testing the incremental refresh

At the time of writing this book, we must have either a Premium or an Embedded capacity to be able to connect the desired workspace in Power BI Service. You must use some applications such as **SQL Server Management Studio** (**SSMS**) or **DAX Studio** to see the partitions we created for incremental data refresh.

> **Note**
>
> If you do not have a Premium or Embedded capacity, you can test them for free for a trial period. You can learn more about how to do that here: `https://www.biinsight.com/what-does-xmla-endpoints-mean-for-power-bi-and-how-to-test-it-for-free/`.

If you already have either a Premium or an Embedded capacity, then perform the following steps to connect to a Premium Workspace:

1. In Power BI Service, navigate to the desired Workspace, backed by a Premium capacity, that contains a dataset with incremental refresh.
2. Click **Settings**.
3. Click the **Premium** tab.
4. Copy the **Workspace Connection** URL:

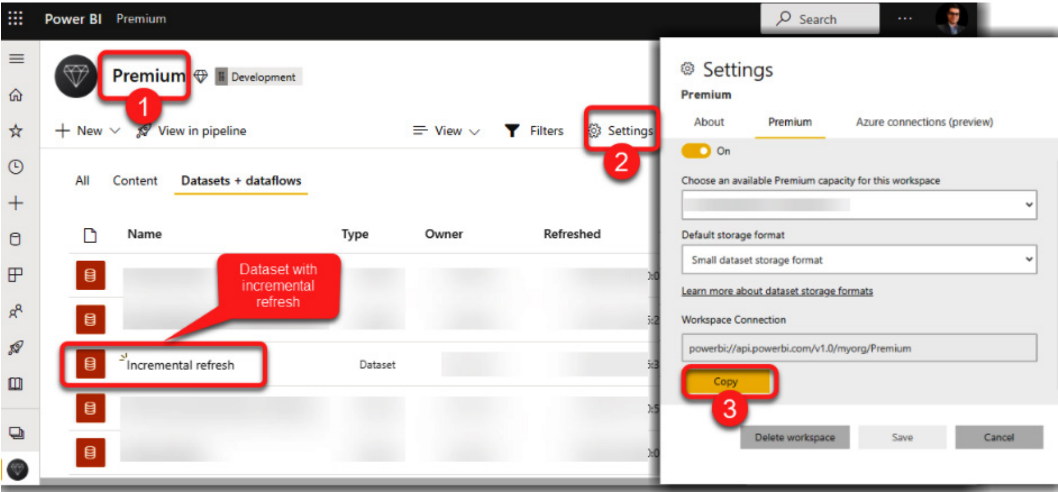


Figure 10.32 – Getting Workspace Connection from Power BI Service

5. Open **SSMS** and in the **Connect to Server** window, do the following:

   a. Select `Analysis Services` as **Server type**.

   b. Paste the **Workspace Connection** link into the **Server name** box.

   c. Select `Azure Active Directory – Universal with MFA` (if you have MFA enabled in your tenant) for **Authentication**.

   d. Type in your **User name**.

e. Click **Connect**:

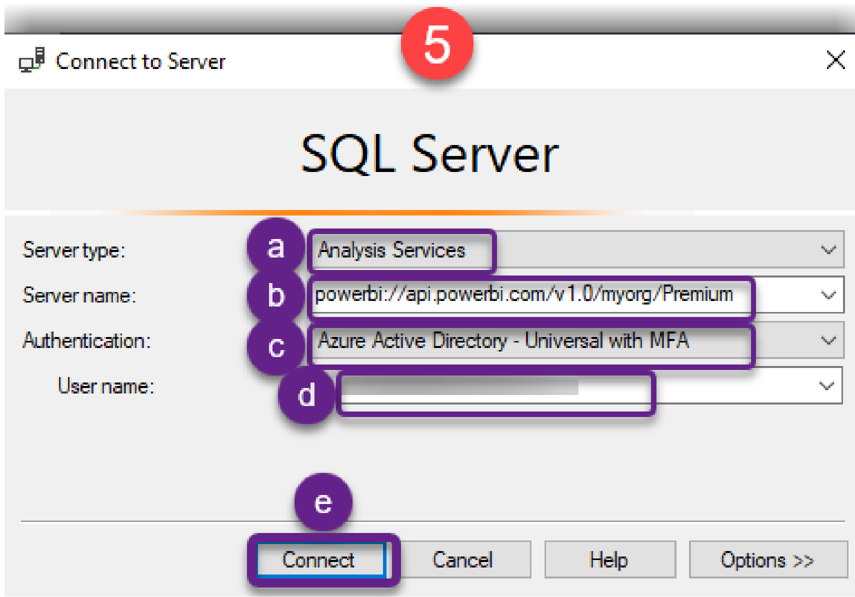


Figure 10.33 – Connecting to a Power BI Premium Workspace from SSMS

6. We can now see all the datasets contained in the Workspace under **Databases** within the **Object Explorer** pane. Expand **Databases**, expand the dataset, and then expand **Tables**.
7. Right-click the `Internet Sales` table.
8. Click **Partitions…**.

9. Now, you can view all the partitions that were created by the incremental refresh process, as shown in the following screenshot:

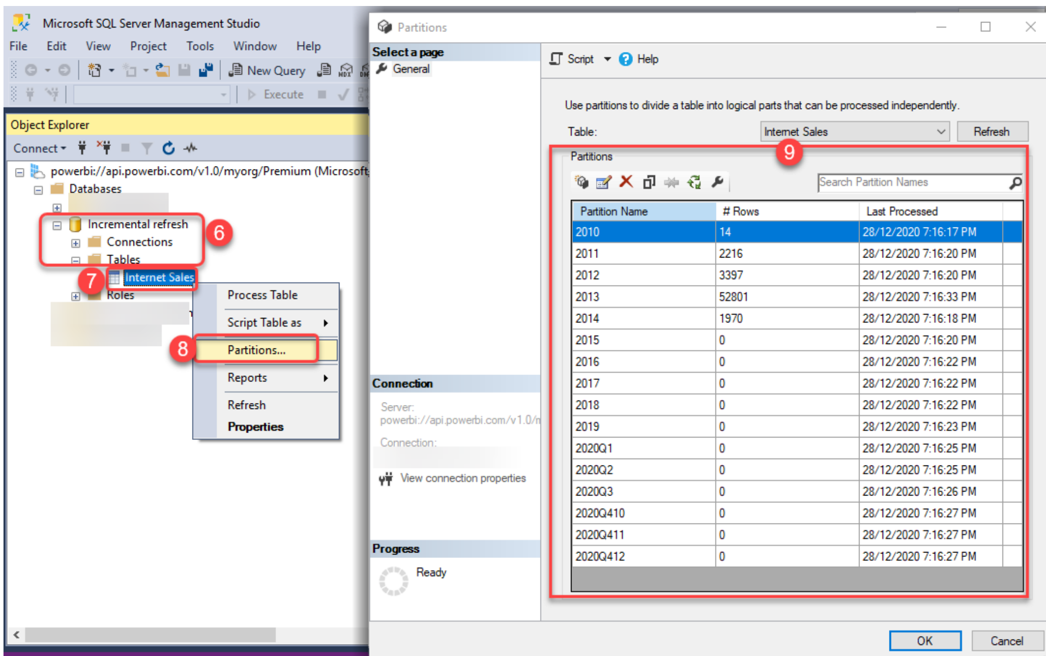


Figure 10.34 – Navigating to a table's partitions from SSMS

As you can see, you can see or modify the created partitions directly from SSMS and save the changes back to Power BI Service. This is made possible by the XMLA endpoint read/write capability, which is available in the Power BI Premium and Embedded capacities.

You can read more about *Dataset connectivity with the XMLA endpoint* here: `https://docs.microsoft.com/en-us/power-bi/admin/service-premium-connect-tools?WT.mc_id=?WT.mc_id=DP-MVP-5003466`.

Understanding Parent-Child hierarchies

The concept of a Parent-Child hierarchy is commonly used in relational data modeling. We have a Parent-Child hierarchy when the values of two columns in a table represent hierarchical levels in the data. Parents have children; their children have children too, which creates a hierarchical graph. Let's continue with an example to understand Parent-Child hierarchies and implement them in relational data modeling. Then, we'll look at the Parent-Child design in Power BI. The following diagram shows a typical Parent-Child graph. Each node of the graph contains an `ID` and the person's `Name`:

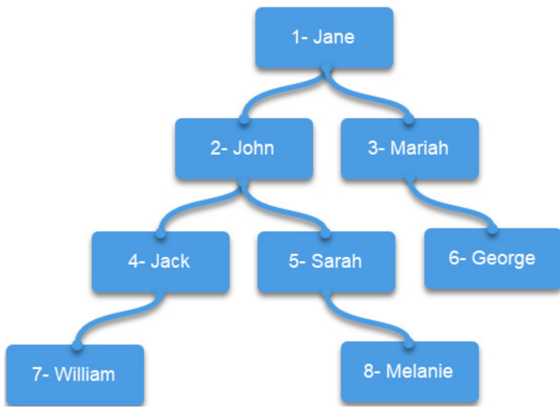


Figure 10.35 – A Parent-Child graph

We can represent the preceding graph in a data table, as shown in the following screenshot:

Parent Child

ID	Name	ParentID
1	Jane	*NULL*
2	John	1
3	Mariah	1
4	Jack	2
5	Sarah	2
6	George	3
7	William	4
8	Melanie	5

Figure 10.36 – Parent-Child graph representation in a data table

We can quickly discover that there is a one-to-many relationship between the **ID** and **ParentID** columns. In generic relational data modeling, we usually create a relationship between the **ID** and **ParentID** columns. The `Parent-Child` table would be a self-referencing table, as the following screenshot:

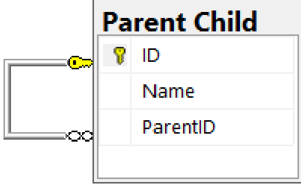


Figure 10.37 – The Parent-Child table is a self-referencing table

So far, we know how to represent a Parent-Child graph in relational data modeling. We also know that data modeling in Power BI is relational. But unfortunately, Power BI does not support self-referencing tables, in order to avoid ambiguity in the data model. However, the good news is that there is a set of DAX functions specifically designed to implement Parent-Child hierarchies in Power BI. Before jumping to their implementation, let's take a moment to understand the implementation process:

- First, we must identify the depth of the hierarchy. The following diagram shows that our example has four levels, so the depth is 4. The nodes at level 4 are leaves of the hierarchy, so William and Melanie are leaves:

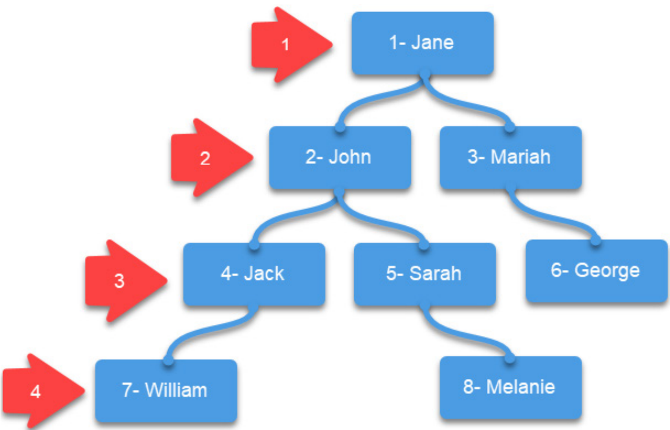


Figure 10.38 – Hierarchy depth

- Then, we must create calculated columns for each level of the hierarchy.
- Finally, we must create a hierarchy using the calculated levels.

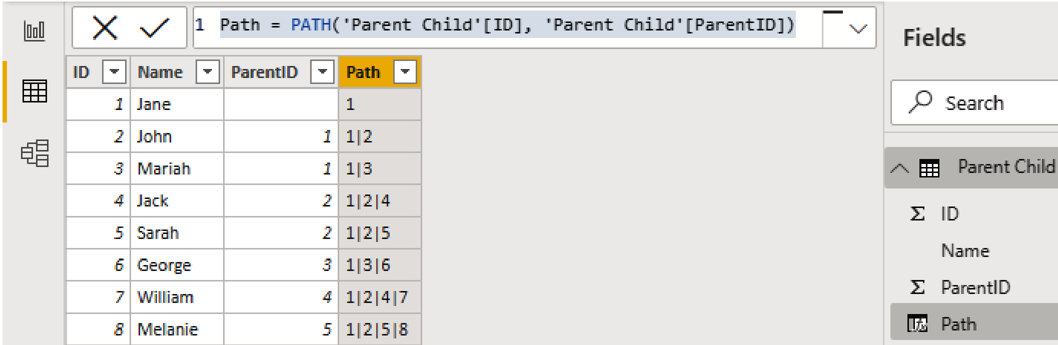
As you can see, the process is quite simple. Now, let's implement a Parent-Child hierarchy.

Identifying the depth of the hierarchy

To identify the depth of a Parent-Child hierarchy, we can use the `PATH(ID, ParentID)` function. The `PATH()` function returns a pipe (|) delimited string starting from the parent and ending with the current child. We need to create a new calculated column with the following DAX expression:

```
Path = PATH('Parent Child'[ID], 'Parent Child'[ParentID])
```

The output of the preceding expression provides the path of our hierarchy, as shown in the following screenshot:



ID	Name	ParentID	Path
1	Jane		1
2	John	1	1|2
3	Mariah	1	1|3
4	Jack	2	1|2|4
5	Sarah	2	1|2|5
6	George	3	1|3|6
7	William	4	1|2|4|7
8	Melanie	5	1|2|5|8

Figure 10.39 – Creating a new calculated column containing the hierarchy path

Considerations in Using the PATH() Function

`ID` and `ParentID` must have the same data type of either `integer` or `text`.

The values of the `ParentID` column must exist in the `ID` column. The `PATH()` function cannot find a parent if the child level does not exist.

If `ParentID` is null, then that node is the root of the graph.

Each `ID` (child level) can have one and only one `ParentID`; otherwise, the `PATH()` function will throw an error.

If `ID` is `BLANK()`, then `PATH()` returns `BLANK()`.

Now that we have the values of the `Path` column, we can quickly identify the depth of the hierarchy using the `PATHLENGTH(PathColumn)` function. We need to create another calculated function using the following DAX expression:

```
Path Length = PATHLENGTH('Parent Child'[Path])
```

The following screenshot shows the output of running the preceding expression:

```
1 Path Length = PATHLENGTH('Parent Child'[Path])
```

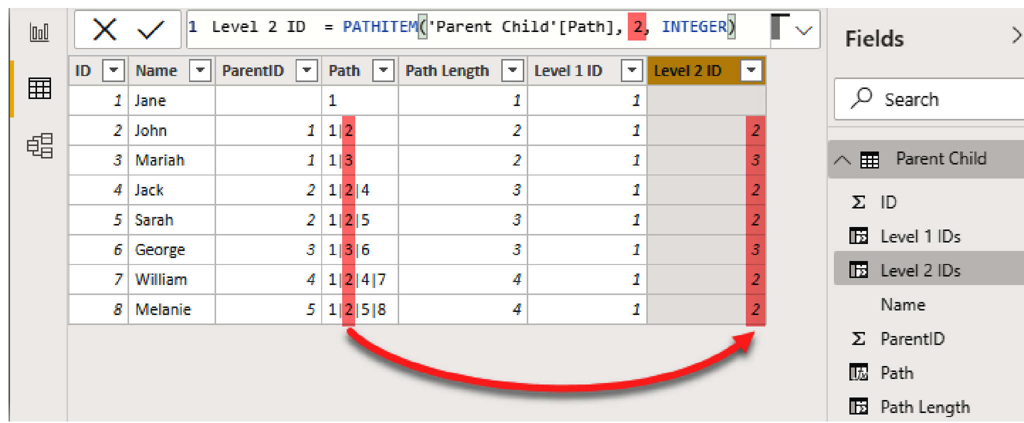
ID	Name	ParentID	Path	Path Length
1	Jane		1	1
2	John	1	1|2	2
3	Mariah	1	1|3	2
4	Jack	2	1|2|4	3
5	Sarah	2	1|2|5	3
6	George	3	1|3|6	3
7	William	4	1|2|4|7	4
8	Melanie	5	1|2|5|8	4

Figure 10.40 – Calculating Path Length

Now that we've calculated `Path Length`, we know that the `depth` value of the hierarchy is `4`. Therefore, we need to create four new calculated columns – one column for each hierarchy level.

Creating hierarchy levels

So far, we've identified the depth of the hierarchy. To implement a Parent-Child hierarchy, in our example, we know that the hierarchy's depth is 4, so we need to create four calculated columns. We can identify these hierarchy levels using the `PATHITEM(Path, Path Length, Datatype)` function, which returns the item for the specified `Path Length` within the `Path` column. In other words, the `PATHITEM()` function returns the values of each specified hierarchy level. `Datatype` is an optional operand that defines the data type of the output results, which is either `INTEGER` or `TEXT`. The default `Datatype` is `TEXT`. Let's take a moment to understand how the `PATHITEM()` function works. For instance, we can read the `PATHITEM('Parent Child'[Path], 2, INTEGER)` expression as "Return the integer item of `Path` when the hierarchy level is `2`." The following screenshot shows the results of running the preceding expression:



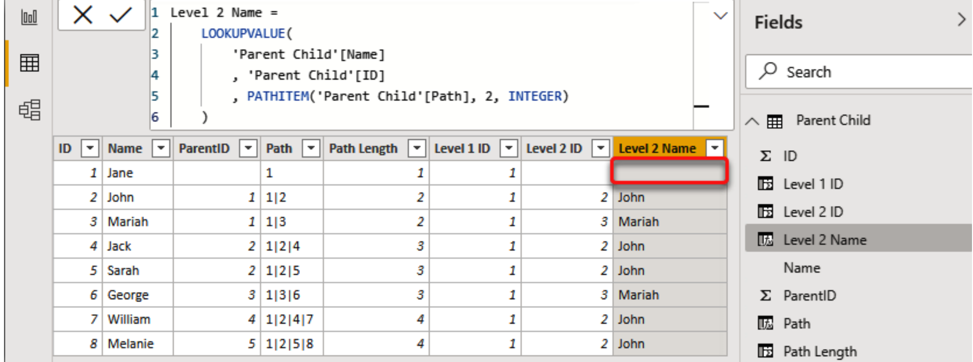
ID	Name	ParentID	Path	Path Length	Level 1 ID	Level 2 ID
1	Jane		1	1	1	
2	John	1	1|2	2	1	2
3	Mariah	1	1|3	2	1	3
4	Jack	2	1|2|4	3	1	2
5	Sarah	2	1|2|5	3	1	2
6	George	3	1|3|6	3	1	3
7	William	4	1|2|4|7	4	1	2
8	Melanie	5	1|2|5|8	4	1	2

Figure 10.41 – The results of running the PATHITEM('Parent Child'[Path], 2, INTEGER) expression

As you can see, `PATHITEM()` returns the IDs of the specified level. But `ID` is not what we are after. We need to return the corresponding value of the `Name` column. To get the corresponding `Name`, we can use the `LOOKUPVALUE(Returning Column, Lookup Column, Lookup Value)` function. For instance, the following expression returns the values from the `Name` column that correspond to the values from the ID column where the hierarchy level is `2`:

```
Level 2 Name =
    LOOKUPVALUE(
        'Parent Child'[Name]
        , 'Parent Child'[ID]
        , PATHITEM('Parent Child'[Path], 2, INTEGER)
    )
```

Running the preceding expression results in the following output:



ID	Name	ParentID	Path	Path Length	Level 1 ID	Level 2 ID	Level 2 Name
1	Jane		1	1	1		
2	John	1	1|2	2	1	2	John
3	Mariah	1	1|3	2	1	3	Mariah
4	Jack	2	1|2|4	3	1	2	John
5	Sarah	2	1|2|5	3	1	2	John
6	George	3	1|3|6	3	1	3	Mariah
7	William	4	1|2|4|7	4	1	2	John
8	Melanie	5	1|2|5|8	4	1	2	John

Figure 10.42 – The results of running the Level 2 Name expression

As you can see, the expression returns nothing (it's blank) for the first value. When we have blank values in a hierarchy, we call it a ragged hierarchy. Ragged hierarchies can cause confusion in the visualization layer as we get `BLANK()` for every hierarchy level. One way to manage ragged hierarchies is to add a filter to avoid `BLANK()` values. The other way is to manage ragged hierarchies in our DAX expressions. We only need to check the value of the `Path Length` column. If it is bigger than or equal to the specified hierarchy level we are looking at, then we return the corresponding value of the `Name` column; otherwise, we return the value of the previous level's `Name`. Obviously, for the first level of the hierarchy, we do not have null values. Hence, we do not need to check `Path Length`. So, we must create four new calculated columns using the following expressions.

The following expression returns the values of the `Name` column for level `1` of the hierarchy:

```
Level 1 Name =
    LOOKUPVALUE(
        'Parent Child'[Name]
        , 'Parent Child'[ID]
        , PATHITEM('Parent Child'[Path], 1, INTEGER)
        )
```

The following expression returns the values of the `Name` column for level `2` of the hierarchy:

```
Level 2 Name =
    IF(
        'Parent Child'[Path Length] >=2
        , LOOKUPVALUE(
            'Parent Child'[Name]
            , 'Parent Child'[ID]
            , PATHITEM('Parent Child'[Path], 2, INTEGER)
        ) //End LOOKUPVALUE
        , 'Parent Child'[Level 1 Name]
    ) // End IF
```

The following expression returns the values of the `Name` column for level `3` of the hierarchy:

```
Level 3 Name =
    IF(
        'Parent Child'[Path Length] >=3
        , LOOKUPVALUE(
            'Parent Child'[Name]
            , 'Parent Child'[ID]
            , PATHITEM('Parent Child'[Path], 3, INTEGER)
        ) //End LOOKUPVALUE
        , 'Parent Child'[Level 2 Name]
    ) // End IF
```

The following expression returns the values of the `Name` column for level `4` of the hierarchy:

```
Level 4 Name =
    IF(
        'Parent Child'[Path Length] >=4
        , LOOKUPVALUE(
            'Parent Child'[Name]
            , 'Parent Child'[ID]
            , PATHITEM('Parent Child'[Path], 4, INTEGER)
        ) //End LOOKUPVALUE
        , 'Parent Child'[Level 3 Name]
    ) // End IF
```

The following screenshot shows the results of running the preceding expressions. Note that we removed the `Level 1 ID` and `Level 2 ID` columns as they were redundant:

```
1  Level 4 Name =
2      IF(
3          'Parent Child'[Path Length] >=4
4          , LOOKUPVALUE(
5              'Parent Child'[Name]
6              , 'Parent Child'[ID]
7              , PATHITEM('Parent Child'[Path], 4, INTEGER)
8          ) //End LOOKUPVALUE
9          , 'Parent Child'[Level 3 Name]
10     ) // End IF
```

ID	Name	ParentID	Path	Path Length	Level 1 Name	Level 2 Name	Level 3 Name	Level 4 Name
1	Jane		1	1	Jane	Jane	Jane	Jane
2	John	1	1|2	2	Jane	John	John	John
3	Mariah	1	1|3	2	Jane	Mariah	Mariah	Mariah
4	Jack	2	1|2|4	3	Jane	John	Jack	Jack
5	Sarah	2	1|2|5	3	Jane	John	Sarah	Sarah
6	George	3	1|3|6	3	Jane	Mariah	George	George
7	William	4	1|2|4|7	4	Jane	John	Jack	William
8	Melanie	5	1|2|5|8	4	Jane	John	Sarah	Melanie

Fields
Search
Parent Child
ID
Level 1 Name
Level 2 Name
Level 3 Name
Level 4 Name
Name
ParentID
Path
Path Length

Figure 10.43 – Four new calculated columns created returning the hierarchy levels

At this point, we have all the hierarchy levels. Now, we can create a hierarchy using these four levels. The following screenshot shows the new hierarchy and a visual representation of the hierarchy in a **Slicer**:

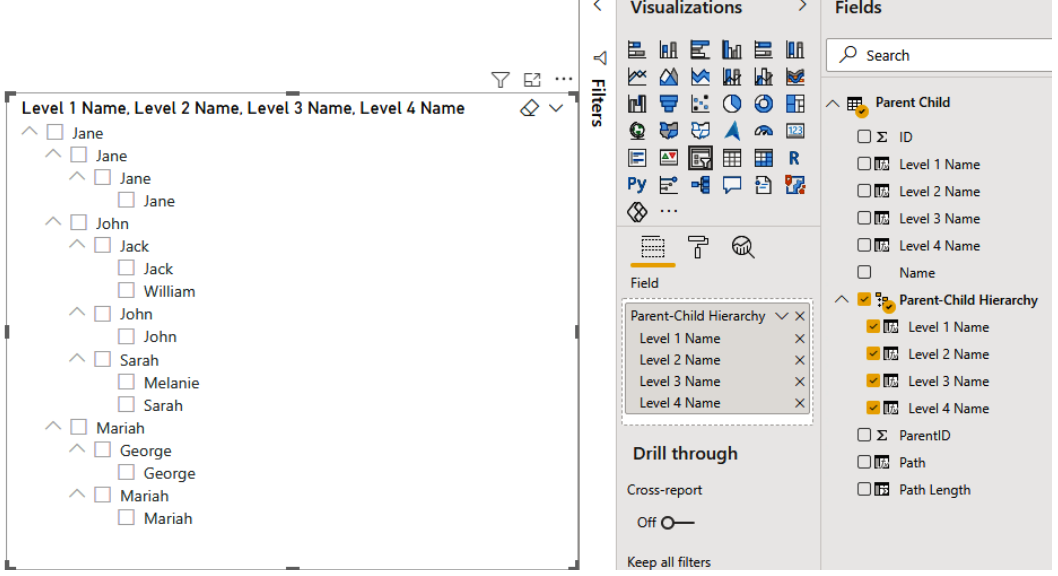


Figure 10.44 – Parent-Child hierarchy

In this section, we discussed how to implement Parent-Child hierarchies. The example we used in this section was a straightforward one, but the principles remain the same. You can use the techniques we discussed in this section to overcome real-world scenarios such as Employee hierarchies, Organizational Charts, and so on. In the next section, we'll discuss roleplaying dimensions and learn how to implement them in Power BI.

Implementing roleplaying dimensions

The roleplaying dimension is one of the most common scenarios we face in data modeling. The term was inherited from multidimensional modeling within the SQL Server Analysis Services Multidimensional. Before jumping to the implementation part, let's take a moment and understand what the roleplaying dimension is. When we create multiple relationships between a fact table and a dimension for logically distinctive roles, we use the concept of a roleplaying dimension. The most popular roleplaying dimensions are the `Date` and `Time` dimensions. For instance, we may have multiple dates in a fact table such as `Order Date`, `Due Date`, and `Ship Date`, which participate in different relationships with the `Date` dimension. Each date represents a different role in our analysis. In other words, we can analyze the data using the `Date` dimension for different purposes. For instance, we can calculate `Sales Amount` by `Order Date`, which results in different values from the values; that is, either `Sales Amount` by `Due Date` or `Sales Amount` by `Ship Date`. But there is a small problem: the xVelocity engine does not support multiple active relationships at the same time. However, we can programmatically enable an inactive relationship using the `USERELATIONSHIP()` function in DAX, which activates the relationship for the duration of the calculation. We can use the `USERELATIONSHIP()` function within the `CALCULATE()` function for this.

Now that we understand what the roleplaying dimension is, let's implement it in a scenario using the `AdventureWorksDW2017.xlsx` sample file:

1. Open Power BI Desktop, connect to the `AdventureWorksDW2017.xlsx` file, and get data from the **Reseller_Sales** and **Dates** tables:

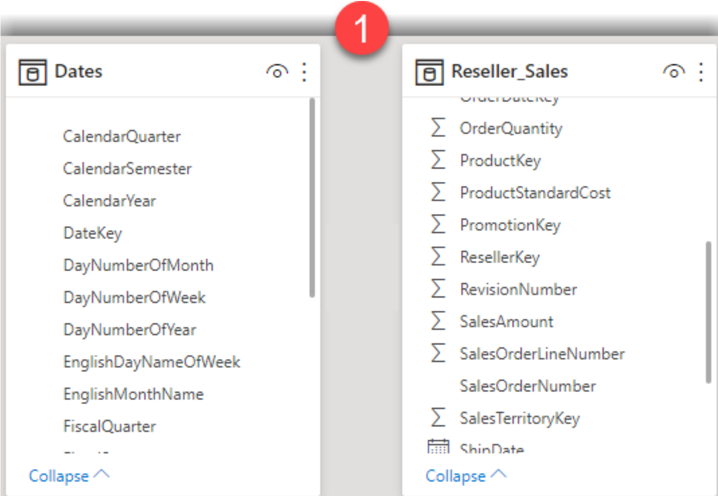


Figure 10.45 – Getting data from the Reseller_Sales and Dates tables

2. Create the following relationships between the `Reseller_Sales` and `Dates` tables. Keep the first relationship active:
3. `Reseller_Sales (OrderDateKey) => Dates (DateKey)`
4. `Reseller_Sales (DueDateKey) => Dates (DateKey)`
5. `Reseller_Sales (ShipDateKey) => Dates (DateKey):`

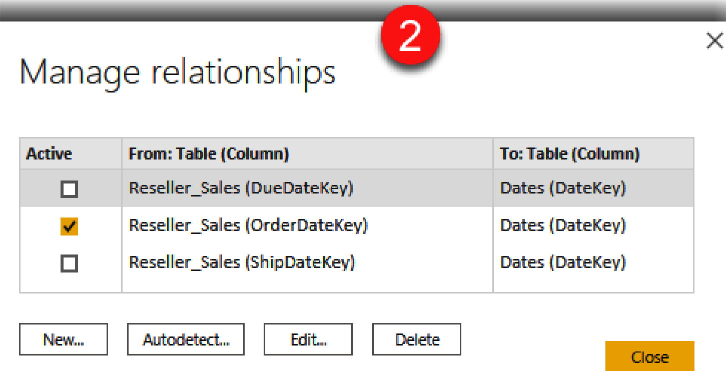


Figure 10.46 – Creating relationships between Reseller_Sales and Dates

We now need to create new measures, one for each role, and name the measures appropriately so they resemble the roles.

> **Note**
>
> Keep in mind that the active relationship between the two tables will remain the primary relationship. Therefore, all the measures that calculate values on top of the `Reseller_Sales` and `Dates` tables will use the active relationship. So, it is essential to make the relationship that makes the most sense to the business active. In our scenario, `Order Date` is the most important date for the business. Therefore, we will keep the `Reseller_Sales (OrderDateKey)` => `Dates (DateKey)` relationship active.

We can create the `Reseller Sales by Order Date` measure using the following DAX expression:

```
Reseller Sales by Order Date = SUM(Reseller_Sales[SalesAmount])
```

We can create the `Reseller Sales by Due Date` measure using the following DAX expression:

```
Reseller Sales by Due Date =
    CALCULATE([Reseller Sales by Order Date]
        , USERELATIONSHIP(Dates[DateKey], Reseller_
Sales[DueDateKey])
    )
```

We can create the `Reseller Sales by Ship Date` measure using the following DAX expression:

```
Reseller Sales by Ship Date =
    CALCULATE([Reseller Sales by Order Date]
        , USERELATIONSHIP(Dates[DateKey], Reseller_
Sales[ShipDateKey])
    )
```

As you can see, we did not use the `USERELATIONSHIP()` function in the `Reseller Sales by Order Date` measure as the active relationship between the `Reseller_Sales` and `Dates` tables is `Reseller_Sales (OrderDateKey)` => `Dates (DateKey)`.

Now, we can use the preceding measures in our data visualizations. The following screenshot shows that all the measures that were created to support roleplaying dimensions are used side by side in a matrix:

Year, Month
2010
2011
2012
January
February
March
April
May
June
July
August
September
October
November
December
2013
2014

EnglishMonthName	Reseller Sales by Order Date	Reseller Sales by Due Date	Reseller Sales by Ship Date
January	$3,601,190.7	$2,393,689.5	$2,393,689.5
February	$2,885,359.2	$3,601,190.7	$3,601,190.7
March	$1,802,154.2	$2,885,359.2	$2,885,359.2
April	$3,053,816.3	$1,802,154.2	$1,802,154.2
May	$2,185,213.2	$3,053,816.3	$3,053,816.3
June	$1,317,541.8	$2,185,213.2	$2,185,213.2
July	$2,384,846.6	$1,317,541.8	$1,317,541.8
August	$1,563,955.1	$2,384,846.6	$2,384,846.6
September	$1,865,278.4	$1,563,955.1	$1,563,955.1
October	$2,880,752.7	$1,865,278.4	$1,865,278.4
November	$1,987,872.7	$2,880,752.7	$2,880,752.7
December	$2,665,650.5	$1,987,872.7	$1,987,872.7
Total	**$28,193,631.5**	**$27,921,670.5**	**$27,921,670.5**

Figure 10.47 – Visualizing roleplaying dimensions

If you are coming from a SQL Server Multidimensional background, you may be thinking of creating multiple `Date` tables. While that is another approach to implementing roleplaying dimensions, we do not recommend going down that path. The following are some reasons against creating multiple `Date` tables to handle roleplaying dimensions:

- Having multiple `Date` tables in our model can confuse other content creators, even if we have only two `Date` tables.
- This approach unnecessarily increases data model size and memory consumption.
- This approach is tough to maintain. We have seen some businesses that have more than 10 roles; have 10 `Date` tables to handle roleplaying dimensions does not sound right.

Using calculation groups

Creating calculation groups is one of the most useful features for Power BI data modelers and developers. It reduces the number of measures you have to create. Calculation groups address the fact that we have to create many measures in larger and more complex data models that are somewhat redundant. Creating those measures takes a lot of development time. For instance, in a Sales data model, we can have `Sales Amount` as a base measure. In real-world scenarios, we usually have to create many time intelligence measures on top of the Sales Amount measure, such as Sales Amount YTD, Sales Amount QTD, Sales Amount MTD, Sales Amount LYTD, Sales Amount LQTD, Sales Amount LMTD, and so on. We have seen models with more than 20 time intelligence measures created on top of a single measure. In real-world scenarios, we have far more base measures and a business that requires all those 20 time intelligence measures for every single base measure. You can imagine how time-consuming it is to develop all those measures. We only need to create the referencing measures with calculation groups once. Then, we can use them with any base measures. In other words, the measures are now reusable. Calculation groups only used to be available in SQL Server Analysis Services Tabular 2019, Azure Analysis Service, and Power BI Premium. But it is now open to all Power BI licensing plans, including Power BI free. However, at the time of writing this book, we cannot implement calculation groups directly in Power BI Desktop; we have to use **Tabular Editor v.2**, a renowned free community tool built by the fantastic *Daniel Otykier*.

With this brief explanation and before we jump into the development process, let's get more familiar with some requirements and terminologies.

Requirements

As we mentioned earlier, at the time of writing this book, we cannot create calculation groups directly in Power BI Desktop:

- We need to download and install Tabular Editor v.2. You can download it from here: `https://tabulareditor.com/`.
- We must have Power BI Desktop July 2020 or earlier.

- This requirement only applies to Power BI Desktop versions before September 2020, so skip this point if you're using the latest Power BI Desktop version: Enable the **Store datasets using enhanced metadata format** setting from **Preview features** in the Power BI Desktop's **Options** window. The following screenshot illustrates how to enable this feature:

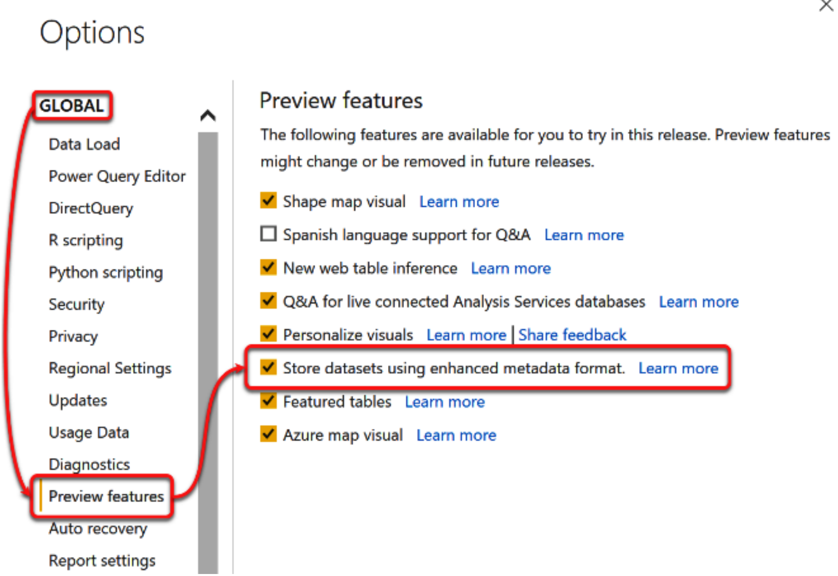


Figure 10.48 – Enabling the "Store datasets using enhanced metadata format" feature

- Disable **Implicit Measures** for the entire data model. We covered this in *Chapter 8*, *Data Modeling Components*, in the *Fields* section.
- The calculation groups do not support implicit measures; therefore, we must have at least one explicit measure in the data model.

Terminology

Let's go through the following terminology involved in calculation groups:

- **Calculation Group**: A calculation group is indeed a table like any other table that holds calculation items:

  Precedence: Each calculation group has a precedence property that specifies the order of evaluation if there is more than one calculation group. The calculation groups with higher precedence numbers will be evaluated before the calculation groups with lower precedence.

- **Calculation Item**: We create calculation items within a calculation group using DAX expressions. Calculation items are like template measures that can run over any explicit measures we have in our data model. Each calculation group has two columns: `Name` with a `Text` data type and `Ordinal` with a `Whole Number` data type. The `Name` column keeps the names of calculation items. The `Ordinal` column is a hidden column that keeps the sort order of the `Name` column. In other words, the `Ordinal` column sorts the `Name` column. We can create as many calculation items as required:

  Ordinal: Each calculation item has an ordinal property. The ordinal property dictates the order in which the calculation items appear in a visual within the report. When we set the ordinal property of the calculation items, we are entering the values of the `Ordinal` column. If the ordinal property is not set, the calculation items show up in alphabetical order within the report. Setting an ordinal property does not affect the precedence and the order in which the calculation items get evaluated.
- **Sideways Recursion**: The term sideways recursion refers to when a calculation item references other calculation items within the same calculation group. Sideways recursion is allowed unless we create infinite loops, such as when calculation item A refers to calculation item B and vice versa. An infinite loop can also occur when a calculation item references an explicit measure that refers to the first calculation item. It is best to avoid sideways recursion.

Now that we are more familiar with the requirements and terminology, it is time to implement calculation groups.

Implementing calculation groups to handle time intelligence

One of the most popular use cases for implementing calculation groups is to handle time intelligence measures. So, let's create a new calculation group and name it `Time Intelligence`. We must then define a series of time intelligence calculation items within the calculation group to meet the business requirements. This section and the next will use the `Chapter 10, Calculation Groups.pbix` sample file, which sources the data from the `AdventureWorksDW2017.xlsx` file. We've already loaded the data from the `Internet_Sales` and `Dates` tables. We then took some transformation steps by renaming the tables and columns.

We also removed unnecessary columns from the data model; then, we **Imported** the data into the data model and **marked Date table as Date**. We also downloaded and installed Tabular Editor v.2. With that, let's get started:

1. Click **Tabular Editor** from the **External Tool** tab:

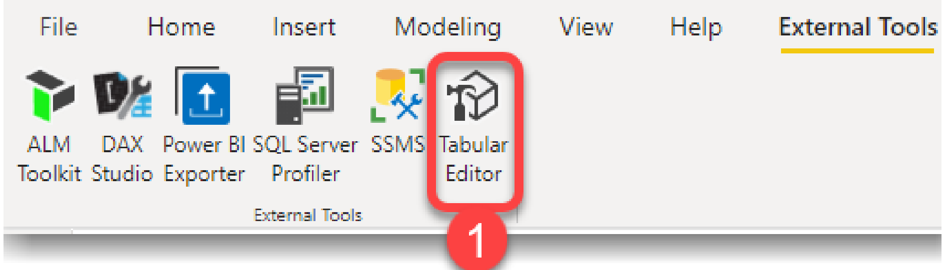


Figure 10.49 – Opening Tabular Editor from the External Tools tab

2. In the Tabular Editor, right-click the **Tables** node, hover over **Create New**, and click **Calculation Group**. We can also use the *Alt + 7* keyboard shortcut:

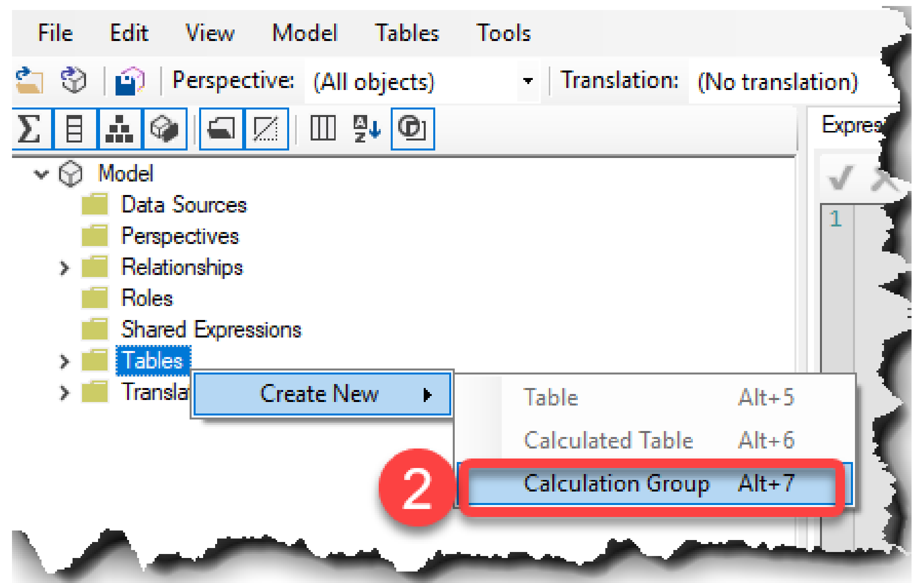


Figure 10.50 – Creating a new calculation group in Tabular Editor

3. Name the new calculation group **Time Intelligence**.
4. Set **Calculation Group Precedence** to `10`:

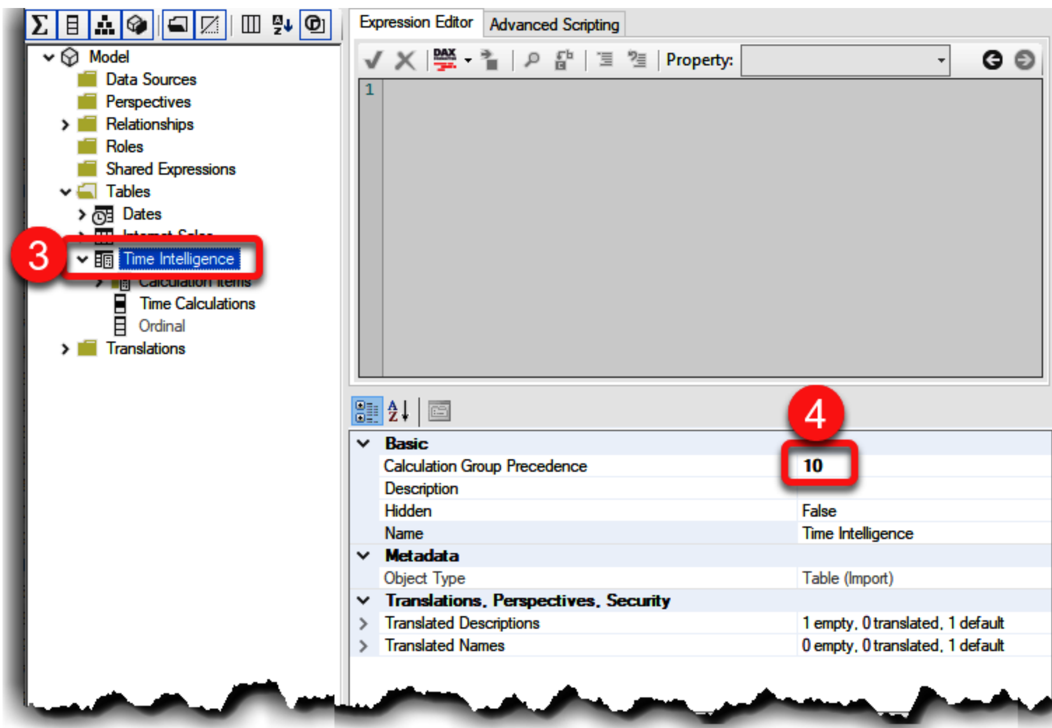


Figure 10.51 – Creating a calculation group

5. Rename the `Name` column to `Time Calculations`.
6. Right-click the `Calculation Items` node and click `New Calculation Item`:

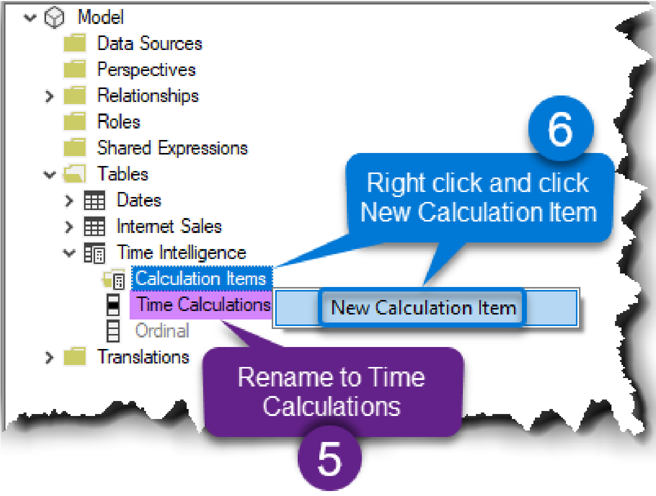


Figure 10.52 – Creating calculation items in Tabular Editor

7. Name the new calculation item `Current`. This calculation item will show the current value of a selected measure.
8. Type the `SELECTEDMEASURE()` expression into the **Expression Editor** box.
9. Set the `Ordinal` property to `0`.
10. Click the **Accept changes** button (✔):

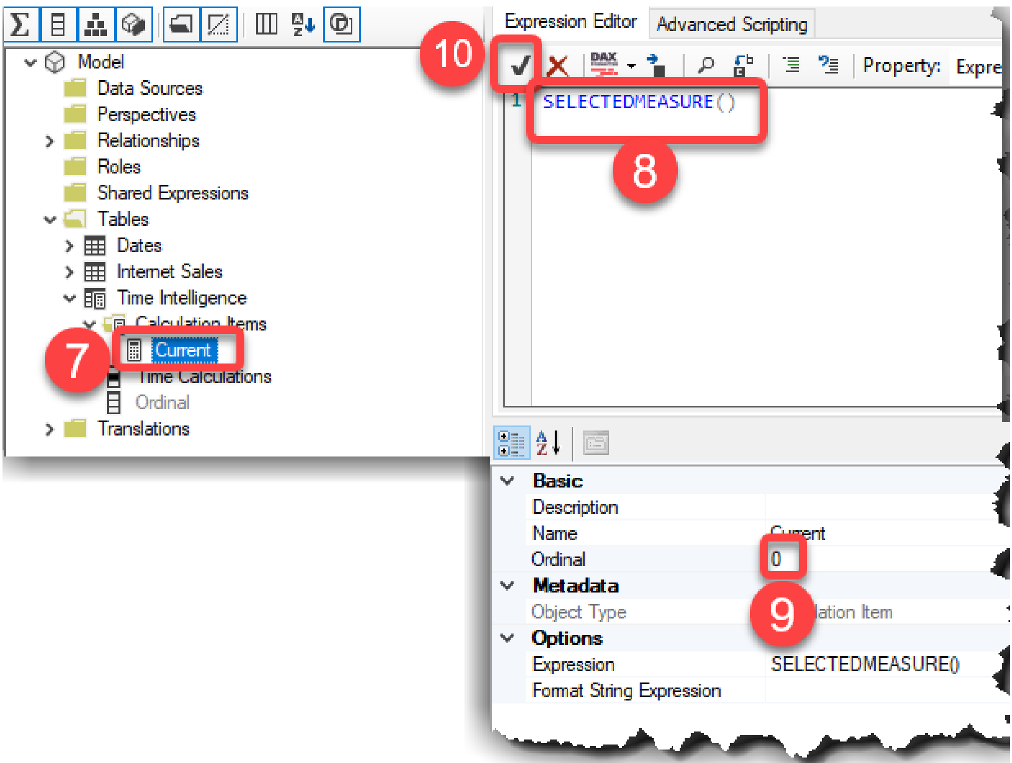


Figure 10.53 – Adding a DAX expression for a calculation item

11. Create another calculation item, name it `YTD` with the `TOTALYTD(SELECTEDMEASURE(), 'Dates'[Date])` expression, and set its `Ordinal` property to `1`:

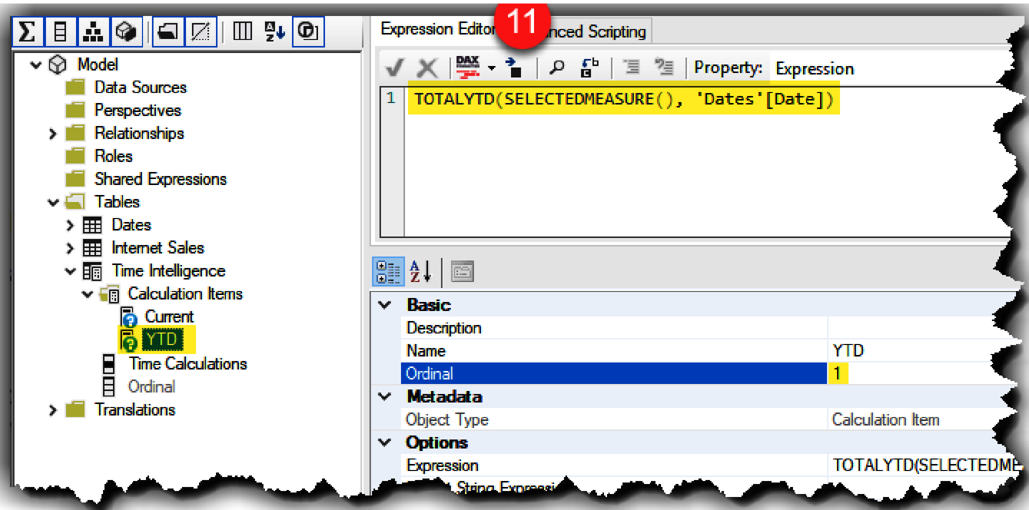


Figure 10.54 – Creating another calculation item

12. Click the **Save** ( ) button to save the changes back to our data model in Power BI Desktop:

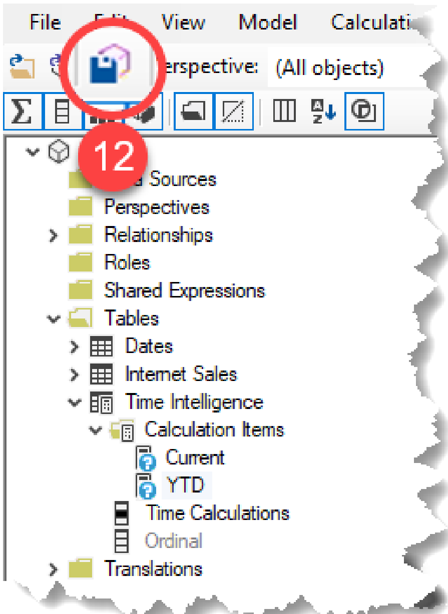


Figure 10.55 – Saving the changes from Tabular Editor back to Power BI Desktop

13. Go back to Power BI Desktop and click the **Refresh now** button on the yellow warning ribbon:

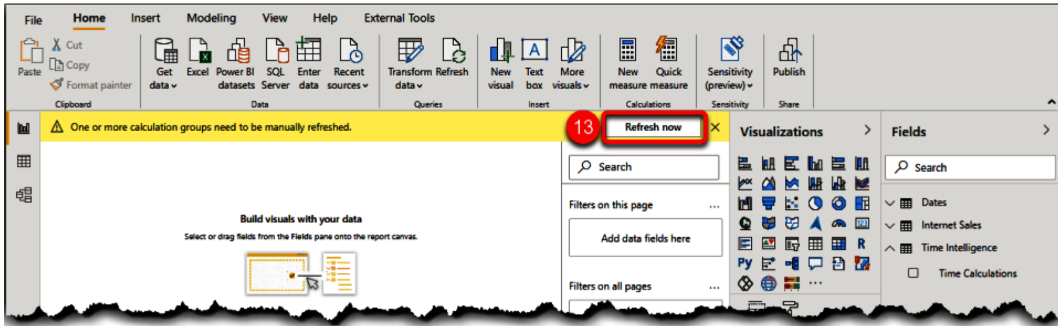


Figure 10.56 – Refreshing the calculation group after applying the changes back to Power BI Desktop

We can create as many calculation items as the business requires. In our example, we added seven calculation items, as shown in the following screenshot:

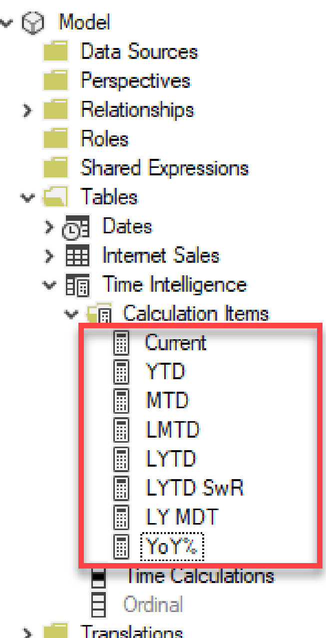


Figure 10.57 – Calculation items created in the sample file

Let's look at these expressions. `Ordinal` is used to create the preceding calculation items:

Calculation Item	DAX Expression	Ordinal	Description
Current	*SELECTEDMEASURE()*	0	Shows the current values of a selected measure
YTD	*TOTALYTD(SELECTEDMEASURE() , 'Dates'[Date])*	1	Year to Date
MTD	*TOTALMTD(SELECTEDMEASURE() , 'Dates'[Date])*	2	Month to Date
LMTD	*TOTALMTD(SELECTEDMEASURE() , DATEADD('Dates'[Date], -1, MONTH))*	3	Last Month to Date
LYTD	*TOTALYTD(SELECTEDMEASURE() , DATEADD('Dates'[Date], -1, YEAR))*	4	Last Year to Date
LYTD SwR	*CALCULATE(SELECTEDMEASURE() , DATEADD('Dates'[Date], -1, YEAR) , 'Time Intelligence'[Time Calculations] = "YTD")*	5	Last Year to Date with Sideways Recursion
LY MTD	*TOTALMTD(SELECTEDMEASURE() , SAMEPERIODLASTYEAR('Dates'[Date]))*	6	Last Year Month to Date
YoY%	*var _current = SELECTEDMEASURE() var _ly = CALCULATE(SELECTEDMEASURE() , DATEADD('Dates'[Date], -1, YEAR)) return DIVIDE(_current - _ly, _ly)*	7	Year Over Year Change %

Figure 10.58 – The expressions and ordinals used to create calculated items in the sample file

Look at the `LYTD SwR` calculation item in the preceding expressions. The `LYTD SwR` calculation item is an example of using **Sideways Recursion**. The results of the `LYTD SwR` calculation item are the same as the `LYTD` calculation item. We just wanted to show what **Sideways Recursion** looks like in action. Again, remember to avoid Sideways Recursions when possible. They can add unnecessary complexities to our code. Besides, Sideways Recursion can become problematic for report contributors who do not have any context on Sideways Recursion. Now that we've finished the implementation, let's test it out.

Testing calculation groups

As we mentioned previously, calculation groups only work with explicit measures. So, we must create at least one explicit measure in our sample to make the calculation groups work. We created the following measures in the sample file:

A measure to calculate `SalesAmount`:

```
Total Sales = SUM('Internet Sales'[SalesAmount])
```

A measure to calculate `OrderQuantity`:

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

Let's test the calculation group we created to see if it works as expected, as follows:

1. In Power BI Desktop, put a **Matrix** visual on the report page. Put the `Year`, `Month`, and `Date` columns from the `Dates` table into **Rows**.
2. Put the `Time Calculations` column from the Time Intelligence calculation group into **Columns**.
3. Put the `Total Sales` measure from the `Internet Sales` table into **Values**:

Year	Current	YTD	MTD	LMTD	LYTD	LYTD SwR	LY MDT	YoY%
⊞ **2010**	**$43,421.04**	**$43,421.04**	**$43,421.04**					
⊞ **2011**	**$7,075,525.93**	**$7,075,525.93**	**$669,431.50**	**$660,545.81**	**$43,421.04**	**$43,421.04**	**$43,421.04**	**$161.95**
⊟ **2012**	**$5,842,485.20**	**$5,842,485.20**	**$624,502.17**	**$537,955.52**	**$7,075,525.93**	**$7,075,525.93**	**$669,431.50**	**-$00.17**
⊟ **April**	**$400,335.61**	**$1,776,176.93**	**$400,335.61**	**$373,483.01**	**$1,923,431.32**	**$1,923,431.32**	**$502,073.85**	**-$00.20**
Sunday, 1 April 2012	$15,551.91	$1,391,393.22	$15,551.91	$11,554.46	$1,431,889.01	$1,431,889.01	$10,531.53	$00.48
Monday, 2 April 2012	$12,798.54	$1,404,191.77	$28,350.45	$28,368.59	$1,446,202.09	$1,446,202.09	$24,844.61	-$00.11
Tuesday, 3 April 2012	$5,035.97	$1,409,227.74	$33,386.42	$38,812.46	$1,453,358.63	$1,453,358.63	$32,001.15	-$00.30
Wednesday, 4 April 2012	$12,183.70	$1,421,411.45	$45,570.13	$47,538.71	$1,478,545.78	$1,478,545.78	$57,188.30	-$00.52
Thursday, 5 April 2012	$11,488.65	$1,432,900.10	$57,058.78	$56,526.75	$1,486,198.13	$1,486,198.13	$64,840.66	$00.50
Friday, 6 April 2012	$13,869.53	$1,446,769.62	$70,928.31	$59,753.09	$1,497,632.04	$1,497,632.04	$76,274.56	$00.21
Saturday, 7 April 2012	$6,958.12	$1,453,727.74	$77,886.42	$78,509.01	$1,530,878.11	$1,530,878.11	$109,520.63	-$00.79
Sunday, 8 April 2012	$8,593.79	$1,462,321.53	$86,480.21	$94,442.93	$1,560,000.09	$1,560,000.09	$138,642.61	-$00.70
Monday, 9 April 2012	$15,574.23	$1,477,895.76	$102,054.44	$111,299.03	$1,577,891.44	$1,577,891.44	$156,533.96	-$00.13
Tuesday, 10 April 2012	$17,916.64	$1,495,812.39	$119,971.08	$126,670.25	$1,605,907.76	$1,605,907.76	$184,550.28	-$00.36
Wednesday, 11 April 2012	$24,070.79	$1,519,883.19	$144,041.87	$136,454.20	$1,648,465.44	$1,648,465.44	$227,107.96	-$00.43
Thursday, 12 April 2012	$12,627.81	$1,532,511.00	$156,669.68	$154,916.92	$1,666,699.32	$1,666,699.32	$245,341.85	-$00.31
Friday, 13 April 2012	$10,856.45	$1,543,367.45	$167,526.13	$171,286.76	$1,677,434.13	$1,677,434.13	$256,076.66	$00.01
Saturday, 14 April 2012	$13,687.02	$1,557,054.47	$181,213.15	$180,142.33	$1,684,387.39	$1,684,387.39	$263,029.92	$00.97
Sunday, 15 April 2012	$10,945.13	$1,567,999.60	$192,158.28	$188,404.65	$1,702,571.28	$1,702,571.28	$281,213.81	-$00.40
Monday, 16 April 2012	$13,122.22	$1,581,121.82	$205,280.50	$198,131.33	$1,713,102.81	$1,713,102.81	$291,745.34	$00.25
Tuesday, 17 April 2012	$21,384.84	$1,602,506.66	$226,665.34	$216,799.44	$1,741,550.69	$1,741,550.69	$320,193.22	-$00.25
Total	**$29,358,677.22**	**$45,694.72**			**$16,351,550.34**	**$16,351,550.34**	**$1,874,360.29**	**$00.00**

Visualizations
Filters
Rows
Year
Month
Date
Columns
Time Calculations
Values
Total Sales
Drill through

Figure 10.59 – Visualizing calculation groups in a Matrix visual

As highlighted in the preceding screenshot, there is an issue with the format string of `YoY%`. In the next section, we will go through a simple process to fix this issue.

Fixing the format string issue

As shown in the preceding screenshot, all the calculation items we created earlier within the **Time Intelligence** calculation group are formatted as currency. But we did not set the format string for any of the calculation items. These calculation items inherit the format string from the selected measure. While inheriting the format string from the selected measure is a convenient feature, as highlighted in the preceding image, it may not work for all calculation items, such as `YoY%`.

The format string for `YoY%` must be a percentage, regardless of what the format string carries from the selected measure. We must set the format string for the `YoY%` calculation item to fix this issue, which overrides the selected measure's format string. Here, we must open Tabular Editor again and set the format string of `YoY%` to `"0.00%"`. The following screenshot shows the preceding fix:

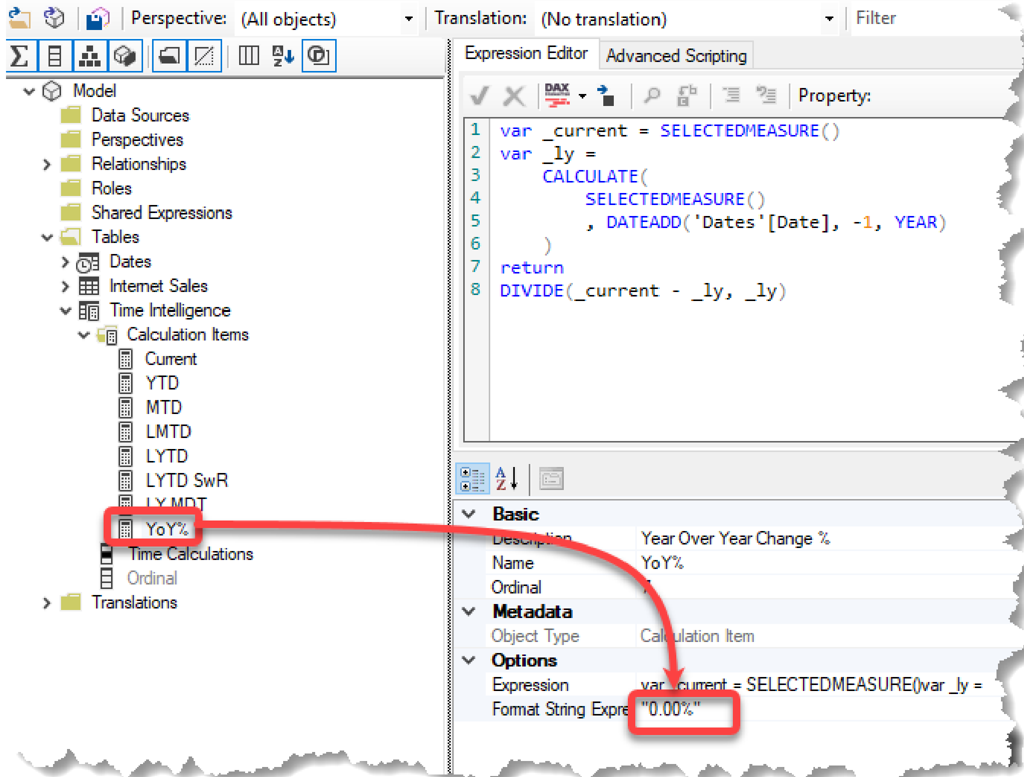


Figure 10.60 – Fixing the format string issue in Tabular Editor

Note

We discussed format strings in *Chapter 8*, *Data Modeling Components*, in the *Fields* section, under the *Custom formatting* subsection.

The following screenshot shows the Matrix visual after saving the changes back to the Power BI Desktop model:

Year	Current	YTD	MTD	LMTD	LYTD	LYTD SwR	LY MDT	YoY%
⊟ **April**	**$400,335.61**	**$1,776,176.93**	**$400,335.61**	**$373,483.01**	**$1,923,431.32**	**$1,923,431.32**	**$502,073.85**	**-20.26%**
Sunday, 1 April 2012	$15,551.91	$1,391,393.22	$15,551.91	$11,554.46	$1,431,889.01	$1,431,889.01	$10,531.53	47.67%
Monday, 2 April 2012	$12,798.54	$1,404,191.77	$28,350.45	$28,368.59	$1,446,202.09	$1,446,202.09	$24,844.61	-10.58%
Tuesday, 3 April 2012	$5,035.97	$1,409,227.74	$33,386.42	$38,812.46	$1,453,358.63	$1,453,358.63	$32,001.15	-29.63%
Wednesday, 4 April 2012	$12,183.70	$1,421,411.45	$45,570.13	$47,538.71	$1,478,545.78	$1,478,545.78	$57,188.30	-51.63%
Thursday, 5 April 2012	$11,488.65	$1,432,900.10	$57,058.78	$56,526.75	$1,486,198.13	$1,486,198.13	$64,840.66	50.13%
Friday, 6 April 2012	$13,869.53	$1,446,769.62	$70,928.31	$59,753.09	$1,497,632.04	$1,497,632.04	$76,274.56	21.30%
Saturday, 7 April 2012	$6,958.12	$1,453,727.74	$77,886.42	$78,509.01	$1,530,878.11	$1,530,878.11	$109,520.63	-79.07%
Sunday, 8 April 2012	$8,593.79	$1,462,321.53	$86,480.21	$94,442.93	$1,560,000.09	$1,560,000.09	$138,642.61	-70.49%
Monday, 9 April 2012	$15,574.23	$1,477,895.76	$102,054.44	$111,299.03	$1,577,891.44	$1,577,891.44	$156,533.96	-12.95%
Tuesday, 10 April 2012	$17,916.64	$1,495,812.39	$119,971.08	$126,670.25	$1,605,907.76	$1,605,907.76	$184,550.28	-36.05%
Wednesday, 11 April 2012	$24,070.79	$1,519,883.19	$144,041.87	$136,454.20	$1,648,465.44	$1,648,465.44	$227,107.96	-43.44%
Thursday, 12 April 2012	$12,627.81	$1,532,511.00	$156,669.68	$154,916.92	$1,666,699.32	$1,666,699.32	$245,341.85	-30.75%
Friday, 13 April 2012	$10,856.45	$1,543,367.45	$167,526.13	$171,286.76	$1,677,434.13	$1,677,434.13	$256,076.66	1.13%
Saturday, 14 April 2012	$13,687.02	$1,557,054.47	$181,213.15	$180,142.33	$1,684,387.39	$1,684,387.39	$263,029.92	96.84%
Sunday, 15 April 2012	$10,945.13	$1,567,999.60	$192,158.28	$188,404.65	$1,702,571.28	$1,702,571.28	$281,213.81	-39.81%
Monday, 16 April 2012	$13,122.22	$1,581,121.82	$205,280.50	$198,131.33	$1,713,102.81	$1,713,102.81	$291,745.34	24.60%
Tuesday, 17 April 2012	$21,384.84	$1,602,506.66	$226,665.34	$216,799.44	$1,741,550.69	$1,741,550.69	$320,193.22	-24.83%
Wednesday, 18 April 2012	$14,342.20	$1,616,848.86	$241,007.54	$231,056.66	$1,756,562.87	$1,756,562.87	$335,205.39	-4.46%
Thursday, 19 April 2012	$19,119.81	$1,635,968.66	$260,127.35	$240,925.59	$1,778,032.49	$1,778,032.49	$356,675.01	-10.94%
Friday, 20 April 2012	$12,820.56	$1,648,789.23	$272,947.91	$251,830.26	$1,788,767.30	$1,788,767.30	$367,409.82	19.43%
Total	**$29,358,677.22**	**$45,694.72**			**$16,351,550.34**	**$16,351,550.34**	**$1,874,360.29**	**0.16%**

Figure 10.61 – The Matrix visual after fixing the format string issue

As the preceding screenshot shows, the format string issue has been resolved.

DAX functions for calculation groups

There are many use cases for calculation groups that we haven't covered in this chapter. Therefore, we leave the rest for you to investigate. However, it is worthwhile mentioning the DAX functions that are currently available for calculation groups. The following list briefly explains those functions:

- `SELECTEDMEASURE()`: A reference to an explicit measure that's used on top of calculation items. You can learn more about the `SELECTEDMEASURE()` function here: `https://docs.microsoft.com/en-us/dax/selectedmeasure-function-dax?WT.mc_id=?WT.mc_id=DP-MVP-5003466`.
- `ISSELECTEDMEASURE( [Measure1], [Measure2], ... )`: Accepts a list of explicit measures that exist within the data model, and then determines if the measure that is currently selected within the visuals is one of the ones mentioned in the input list of parameters. It can be used to apply the calculation logic conditionally. You can learn more about `ISSELECTEDMEASURE( [Measure1], [Measure2], ... )` function here: `https://docs.microsoft.com/en-us/dax/isselectedmeasure-function-dax?WT.mc_id=?WT.mc_id=DP-MVP-5003466`.

- `SELECTEDMEASURENAME()`: Returns the selected measure's name. It can be used to apply the calculation logic conditionally. You can learn more about the `SELECTEDMEASURENAME()` function here: `https://docs.microsoft.com/en-us/dax/selectedmeasurename-function-dax?WT.mc_id=?WT.mc_id=DP-MVP-5003466`.
- `SELECTEDMEASUREFORMATSTRING()`: Returns the format string defined by the selected measure. It can be used to define the format string dynamically based on expressions. You can learn more about the `SELECTEDMEASUREFORMATSTRING()` function here: `https://docs.microsoft.com/en-us/dax/selectedmeasureformatstring-function-dax?WT.mc_id=?WT.mc_id=DP-MVP-5003466`.

Summary

In this chapter, we learned about some advanced data modeling techniques, as well as how to implement aggregations using big data in Power BI. We also learned how to configure incremental refresh, which also helps deal with the challenges of working with large data sources. Then, we looked at the concept of Parent-Child hierarchies and implemented one in Power BI Desktop. After that, we learned how to deal with roleplaying dimensions in Power BI. Last but not least, we implemented calculation groups.

In the next chapter, *Row-Level Security*, we will discuss a crucial part of data modeling that is essential for organizations that believe the right people must access the right data in the right way. See you there!

11
Row-Level Security

In the previous chapter, we learned some advanced data modeling techniques such as using aggregations, incremental refresh, implementing a parent-child hierarchy, role-playing dimensions, and calculation groups. In this chapter, we discuss an essential aspect of data modeling, **row-level security** (**RLS**). We will cover the following topics:

- What RLS means in data modeling
- RLS terminologies
- RLS implementation flow
- Common RLS implementation approaches

We try to cover the preceding topics with real-world scenarios, but keep in mind that each Power BI project may have specific requirements, so it is virtually impossible to cover all RLS possibilities and scenarios.

When it comes to Power BI security, many people immediately think it is something related to Power BI administrators, which is correct to some extent. RLS enables the filtering of data within an entire data model, to only show relevant data to the relevant users, so it is an access control mechanism we directly apply to the data model. In terms of this, RLS is directly relevant to data modelers, but at the end of the day we publish our data model into the Power BI service or Power BI Report Server, where other users—including contributors and end users—will see or use the data model. From this point onward, the duty of taking care of the security settings varies from organization to organization. In some organizations, it is purely an administrator's job to take care of the RLS-related security settings within the Power BI service or Power BI Report Server. At the same time, in some other organizations, it falls to data modelers to fully support all aspects of RLS, from development to configuration. The latter, though, is not very common except in small organizations where Power BI developers take care of development, deployment, and administration. Regardless of who takes care of RLS within an organization, we cover the end-to-end implementation and configuration of RLS in this chapter.

With that in mind, let's get started.

What RLS means in data modeling

As mentioned previously, RLS is a mechanism to control user access over data so that the relevant data is accessible only to the relevant user or group of users. This is merely possible by filtering the data based on the users' usernames and the role(s) assigned to them by writing simple **Data Analysis Expressions** (**DAX**) or, in more complex scenarios, by making changes in the data model. Therefore, the relationships between tables and the direction of cross-filtering within these relationships are vital.

At the time of writing this book, developing RLS is only possible within Power BI Desktop.

What RLS is not

As mentioned earlier, RLS is simply nothing but filtering data across an entire data model. At the time of writing this book, **object-level security** (**OLS**) is made available for public preview. We will look at OLS in more detail in *Chapter 12, Extra Options and Features Available for Data Modeling*.

> **Note**
>
> We cannot control the visibility of data to developers within Power BI Desktop with RLS— this is something that must happen within the source system. Regardless of having RLS in place or not, developers have access to all data available in the source systems, so it is a mechanism to restrict data access, not a permission configuration.

RLS terminologies

There are some terminologies that we should be familiar with before implementing RLS. The following sections introduce these terminologies.

Roles

A role is a name that identifies the characteristic of a security rule over tables in our data model. It is best to pick a meaningful name that quickly describes the underlying security rules. We can define roles in Power BI Desktop as follows:

1. Click the **Modeling** tab.
2. Click the **Manage roles** button.
3. Click the **Create** button to add a new role.

The preceding steps are highlighted in the following screenshot:

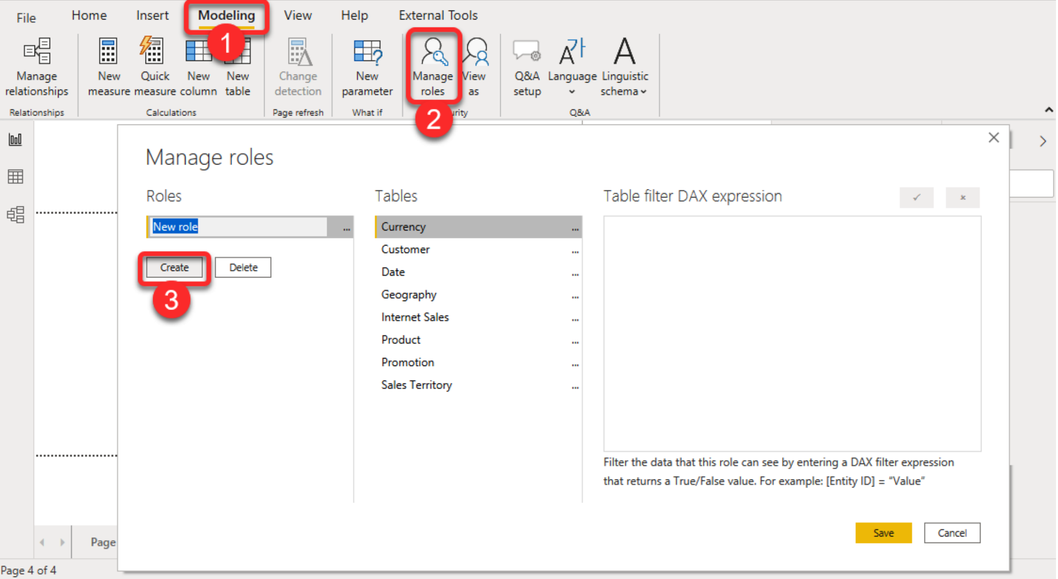


Figure 11.1 – Roles in Power BI Desktop

Rules

Security rules (or, in short, rules) are DAX—the expressions defining the data that a role can see. A DAX expression defining a rule returns a value of `true` or `false`. Follow the next steps to add a new rule:

1. Click the **Manage roles** button from the **Modeling** tab of the ribbon.
2. Type in a DAX expression that returns `true` or `false`. We can also click the ellipsis button of a table that we want to apply the rule to in order to select a specific column.
3. Click the **Verify DAX expression** button.

The following screenshot shows a rule that filters out the `Currency` table's data to only show sales with a currency of `AUD`. We defined the role under the **AUD Sales Only** role:

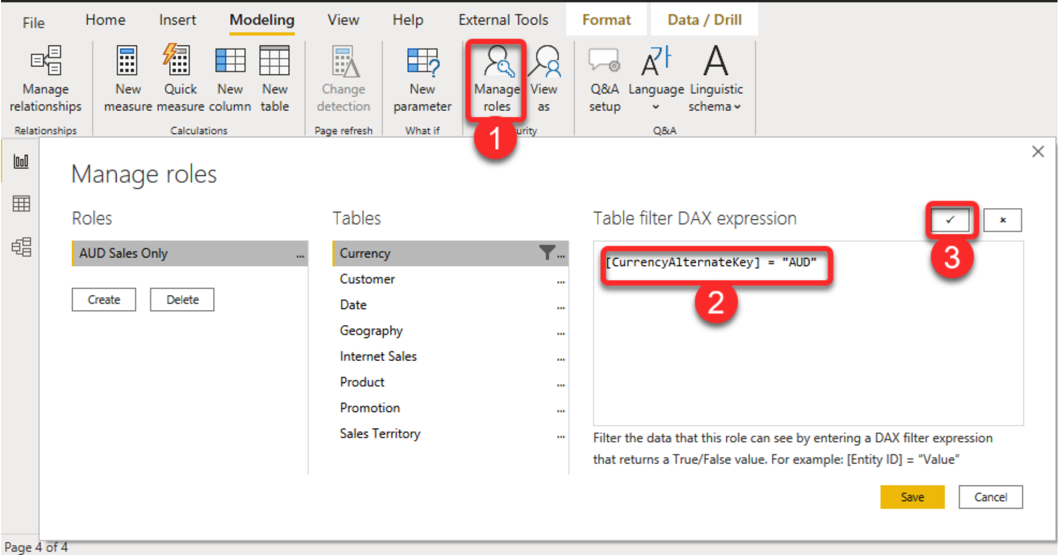


Figure 11.2 – Defining RLS rules

Validating roles

When we create roles and rules, we need to test them. Testing roles is also referred to as validating roles. We can validate roles in both Power BI Desktop and the Power BI service. The following steps show role validation in Power BI Desktop:

1. Click the **View as** button from the **Modeling** tab.
2. Select a role to validate.
3. Click **OK**.

The preceding steps are highlighted in the following screenshot:

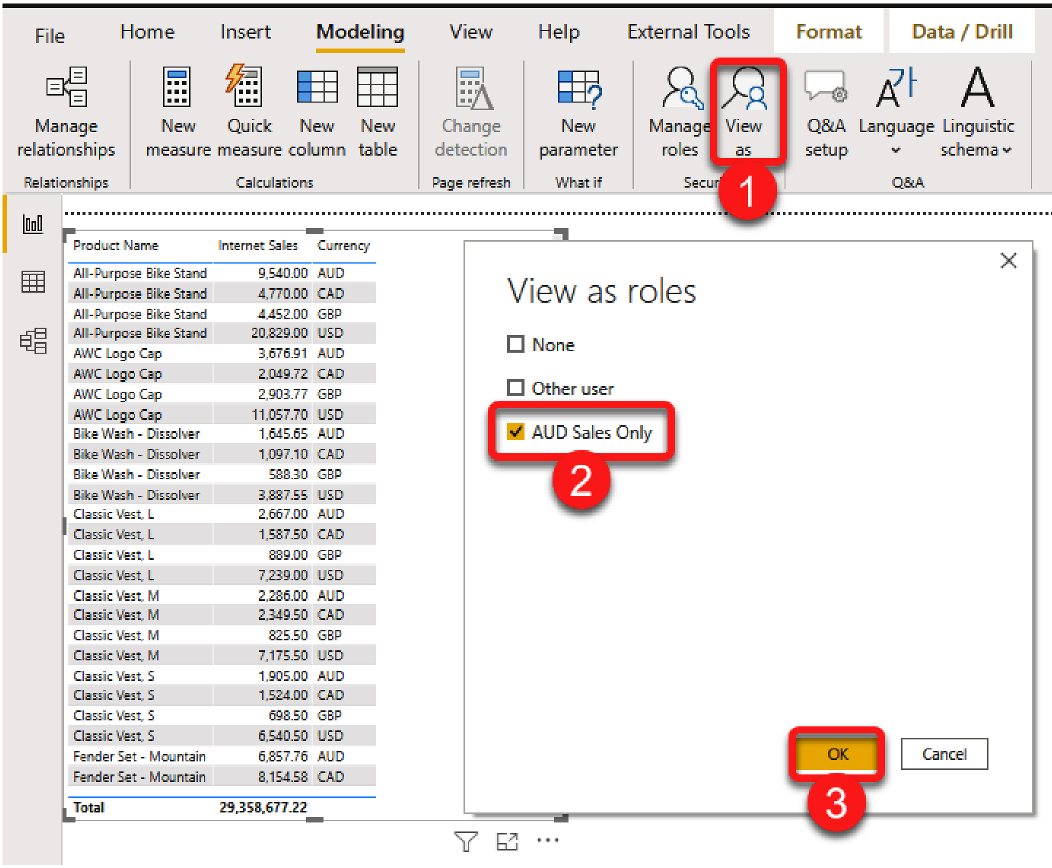


Figure 11.3 – Validating RLS roles

The following screenshot shows that the results of the validation only include `Internet Sales` with a currency of `AUD`. We can click the **Stop viewing** button to terminate the validation:

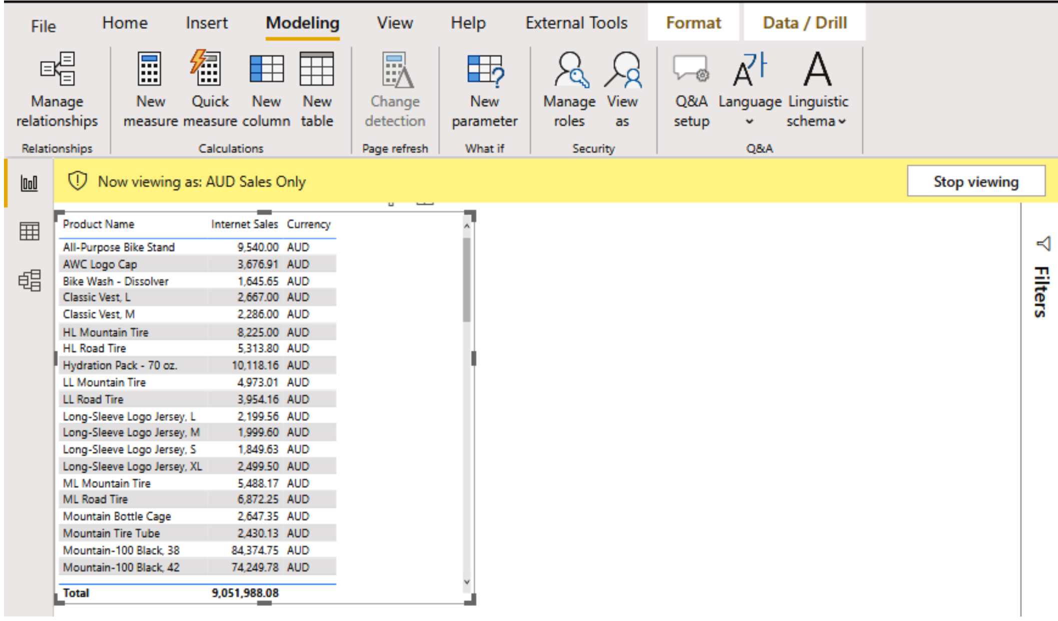


Figure 11.4 – RLS role validation results

Assigning members to roles in the Power BI service

After having implemented RLS, we need to assign users or groups to roles. Managing members is a part of RLS security management within the Power BI service or Power BI Report Server. The following steps show how to assign members to a role in the Power BI service after publishing a report to a workspace:

1. Click the **RLS** button.
2. Click the ellipses button next to **Dataset.**
3. Click **Security**.
4. Select a role and type the name of a user of a group.
5. Click the **Add** button.

The preceding steps are highlighted in the following screenshot:

Figure 11.5 – Assigning members to RLS roles in the Power BI service

In the next section, we look at an RLS implementation flow in Power BI.

Assigning members to roles in Power BI Report Server

The following steps show how to assign members to a role in Power BI Report Server after you have published a report to the server:

1. Open a web browser and navigate to **Power BI Report Server**.
2. Click the ellipsis button of the desired report.
3. Click **Manage**.

The preceding steps are highlighted in the following screenshot:

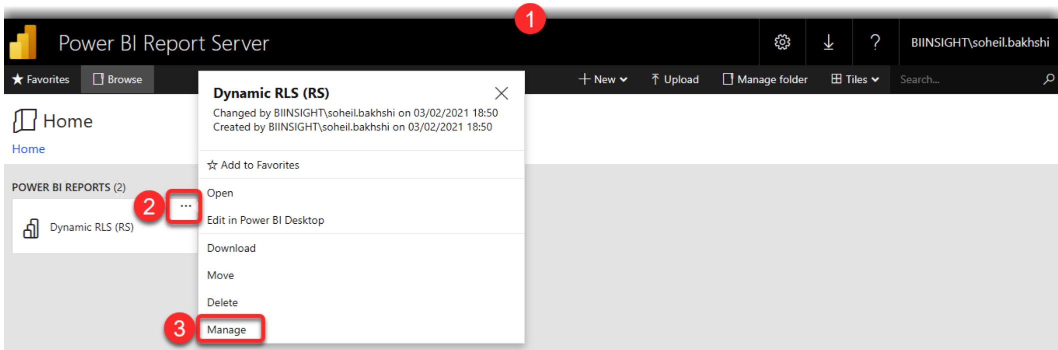


Figure 11.6 – Managing a report in Power BI Report Server

4. Click **Row-level security**.
5. Click the **Add Member** button, as highlighted in the following screenshot:

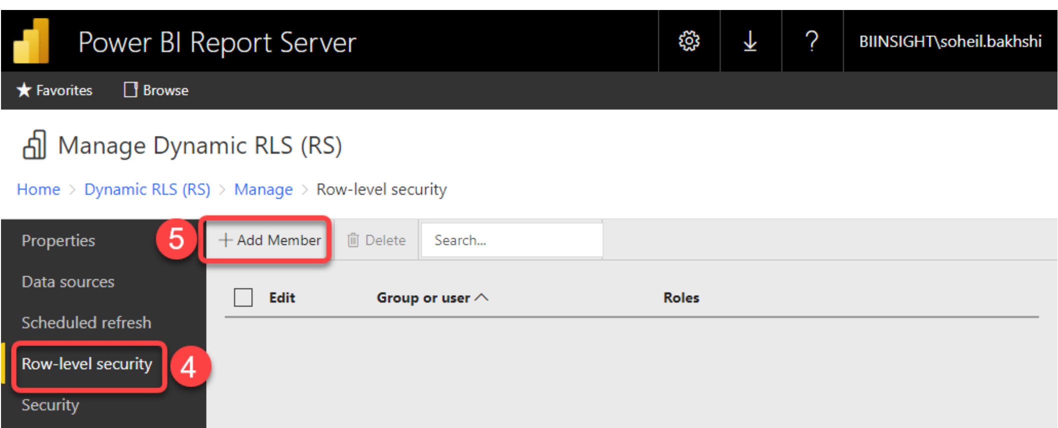


Figure 11.7 – Managing RLS in Power BI Report Server

6. Type in a username or group name.
7. Select roles to assign to the user.
8. Click **OK**.

The preceding steps are highlighted in the following screenshot:

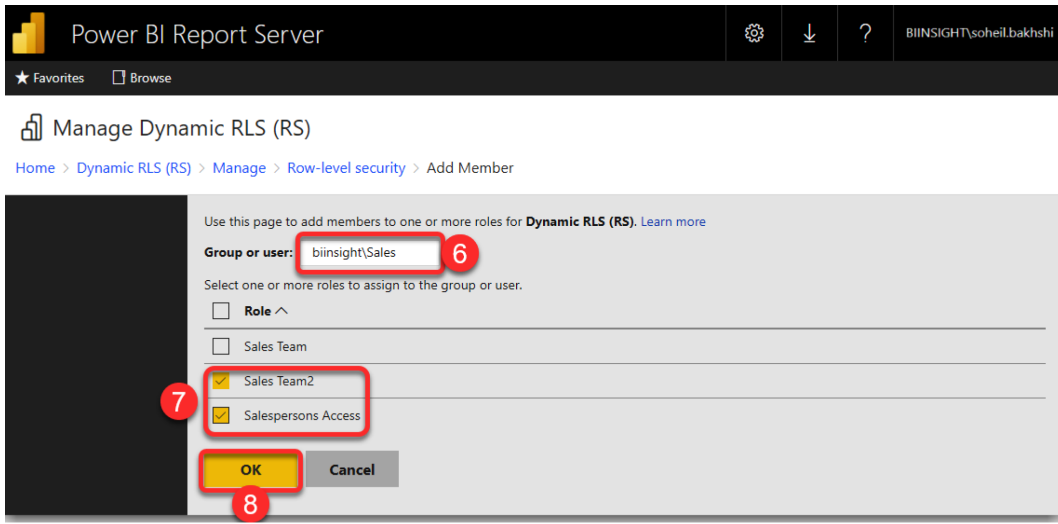


Figure 11.8 – Assigning users or groups to roles in Power BI Report Server

After we have assigned users or groups to roles, users can only see data that is relevant to them.

RLS implementation flow

Implementing RLS in Power BI always follows the same flow, which applies to all implementation approaches and all supported storage modes. We can implement RLS in the data model; therefore, the dataset's storage mode must be in **Import mode**, **DirectQuery mode** or **Composite mode** (**Mixed** mode). Once we have a data model, we have to go through the following flow:

1. Creating security roles.
2. Defining rules within the roles.
3. Validating roles in Power BI Desktop.
4. Publishing a report to the Power BI service or Power BI Report Server.
5. Assigning members to roles within the service or Power BI Report Server.
6. Validating roles in the Power BI service (role validation is not available in Power BI Report Server).

The following diagram illustrates the preceding flow:

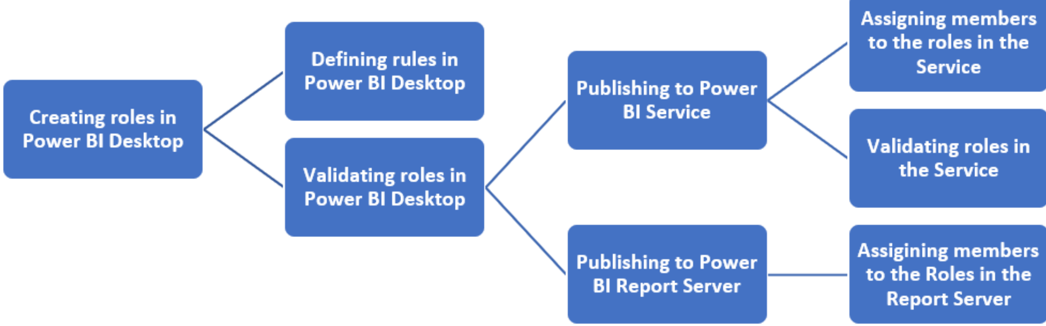


Figure 11.9 – RLS implementation flow

In the next section, we look at different implementation approaches.

Common RLS implementation approaches

There are usually two different approaches to implementing RLS in Power BI Desktop: static RLS and dynamic RLS. In the following few sections, we look at both approaches by implementing real-world scenarios.

Implementing static RLS

A static RLS approach is used when we define rules that statically apply filters to the data model—for example, in *Figure 11.2*, we created a static RLS rule to filter the `Internet Sales` amounts by currency when the currency equals `AUD`. Static RLS is simple to implement, but it can get quite expensive to maintain and support. However, in some cases, static RLS is just enough to satisfy the business requirements—for instance, when the data model does not include data to support dynamic RLS, implementing and supporting dynamic RLS can be more expensive than implementing and supporting static RLS. A good example is in an international organization with a few security groups within **Azure Active Directory** (**Azure AD**) or Microsoft 365, separating users based on their geographical location. In our sample, the *Adventure Works* organization has two security groups: one for Australia and another for the rest of the world. The business requires RLS implementation so that users from Australia can see only their `Internet Sales` amount. In contrast, the rest of the world can see all `Internet Sales` amounts except Australia's.

The following sections show the implementation of the preceding requirement.

Creating roles and defining rules: Follow these steps to create roles and for defining rules:

1. Click the **Manage roles** button from the **Modeling** tab of the ribbon.
2. Click **Create**.
3. Type in `AUD Sales Only` as the role name.
4. Click the ellipsis button of the `Currency` table.
5. Hover over **Add filter…**.
6. Click **[CurrencyAlternateKey]**.

The preceding steps are highlighted in the following screenshot:

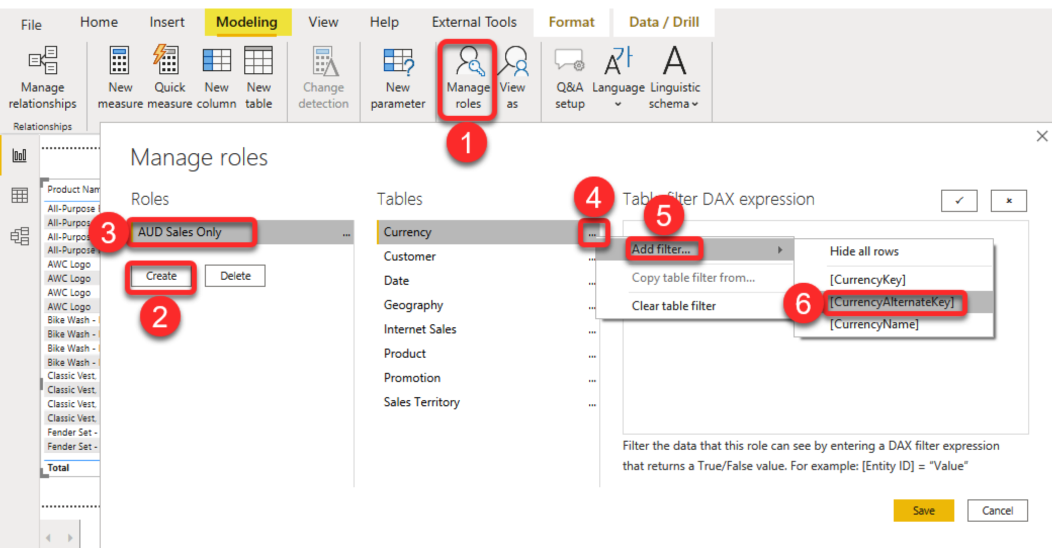


Figure 11.10 – Creating an AUD Sales Only RLS role in Power BI Desktop

7. This automatically creates a `[CurrencyAlternateKey] = "Value"` DAX expression. Replace the `Value` field with `AUD`.
8. Click the **Verify DAX Expression** button.

The preceding steps are highlighted in the following screenshot:

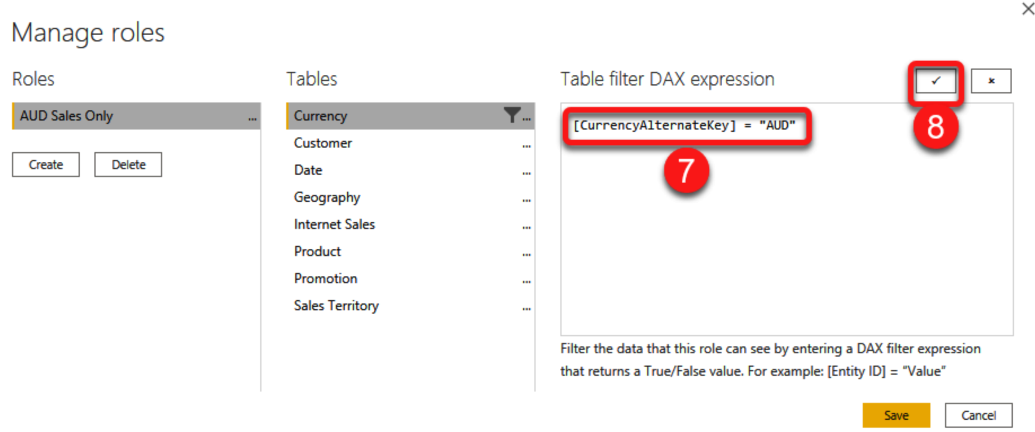


Figure 11.11 – Defining a new RLS rule

9. Click the **Create** button again.
10. Type in `Non-AUD Sales` as the role name.
11. Click the ellipsis button of the `Currency` table.
12. Hover over **Add filter…**.
13. Click **[CurrencyAlternateKey]**.
14. Change the generated DAX expression to `[CurrencyAlternateKey] <> "AUD"`.
15. Click the **Validate DAX Expression** button.
16. Click **Save**.

The preceding steps are highlighted in the following screenshot:

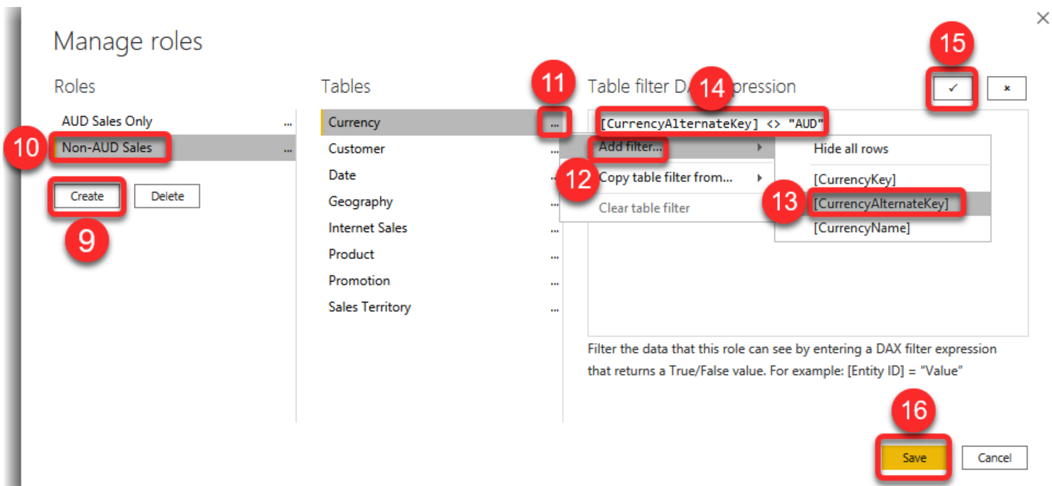


Figure 11.12 – Creating a Non-AUD Sales RLS role in Power BI Desktop

So far, we have created the roles. We now need to validate them.

Validating roles: The following steps explain role validation within Power BI Desktop:

17. Click the **View as** button from the **Modeling** tab.
18. Select a role to validate.
19. Click **OK**.
20. Click the **Stop viewing** button after finishing with the validation.

The following screenshot shows the validation of the **Non-AUD Sales** role:

Figure 11.13 – Validating the Non-AUD Sales role

Now that we have validated the roles and are sure they work as expected, we need to publish a report to the Power BI service model in order to assign members to roles.

Publishing a report to the Power BI service: Follow these next steps to publish a report to the Power BI service:

21. Click the **Publish** button from the **Home** tab of the ribbon.
22. Select the desired workspace from the list.
23. Click the **Select** button.

The preceding steps are highlighted in the following screenshot:

Figure 11.14 – Publishing a Power BI report to the Power BI service

After the report is successfully published, we need to log in to the Power BI service from a web browser to assign members to roles.

Assigning members to roles: After we log in to the Power BI service, we need to navigate to the workspace containing the report we published earlier. The following steps show how to assign members to roles:

24. Click the **More options** button of the desired dataset.
25. Click **Security**.

The preceding steps are highlighted in the following screenshot:

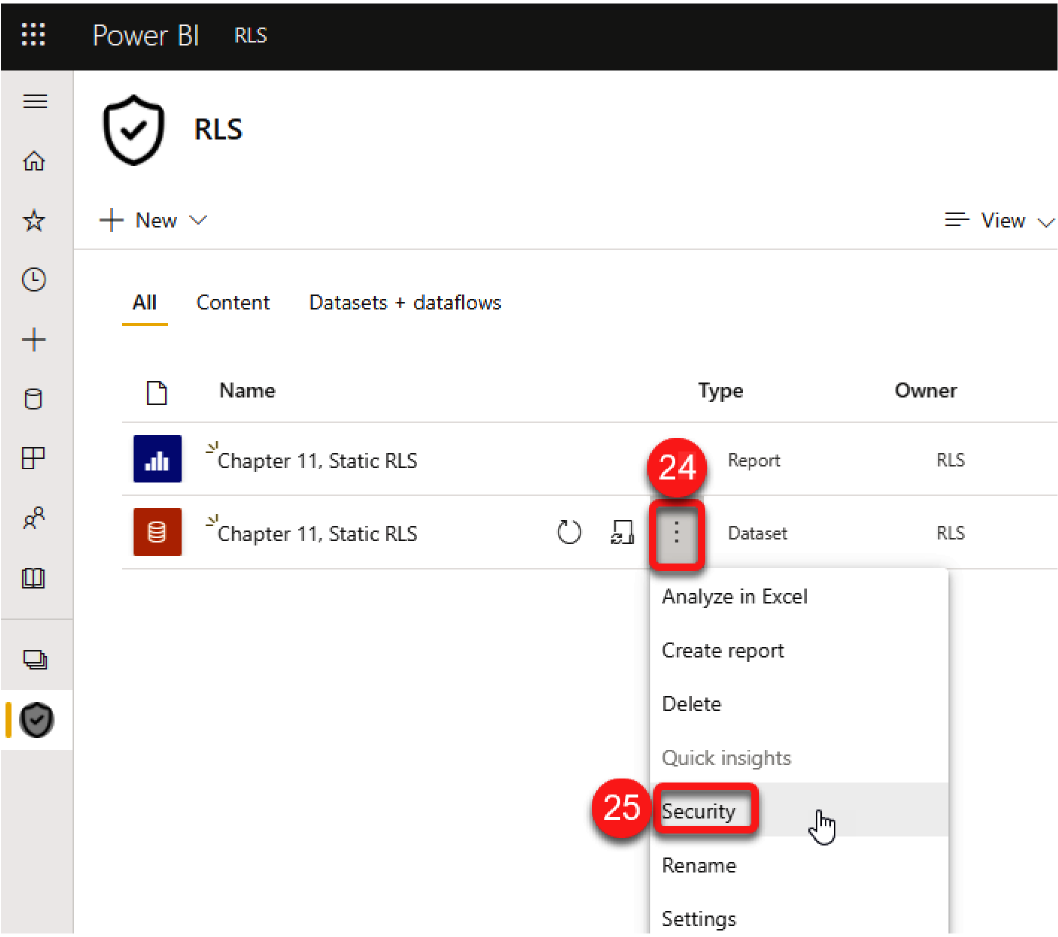


Figure 11.15 – Managing RLS of datasets in the Power BI service

26. Select the desired role.
27. Type in a user or a group; I have two security groups defined in my environment (**AU Users** and **Non-AU Users**), so I assign those two groups to the corresponding roles.
28. Click **Add**.
29. Click **Save**.

The preceding steps are highlighted in the following screenshot:

Figure 11.16 – Assigning members to roles

> **Note**
> We need to have either **Admin** or **Member** access right on the workspace to assign members to roles or validate roles. If we do not have one of the aforementioned access rights, then the **Security** option does not show up in the menu (number **26** in *Figure 11.16*).

We have now successfully implemented RLS to show sales amounts in **Australian Dollars** (**AUD**) to our Australian users and to show non-AUD sales to the rest of the world. As mentioned earlier, we can also validate roles in the Power BI service. To validate roles from the service, we do not need to assign members to roles. Now, let's validate the roles in the Power BI service.

Validating roles in the Power BI service: The following steps will help you to validate roles in the Power BI service:

30. Click the ellipsis button of a role.
31. Click **Test as role**.

The preceding steps are highlighted in the following screenshot:

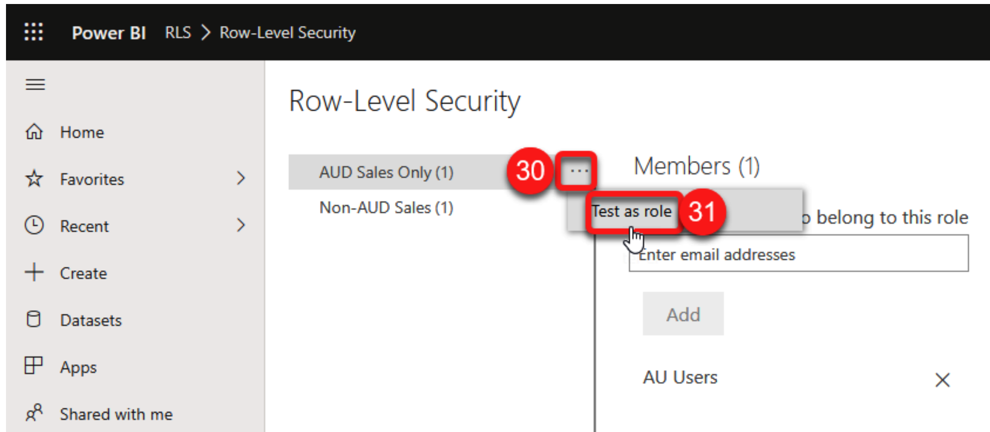
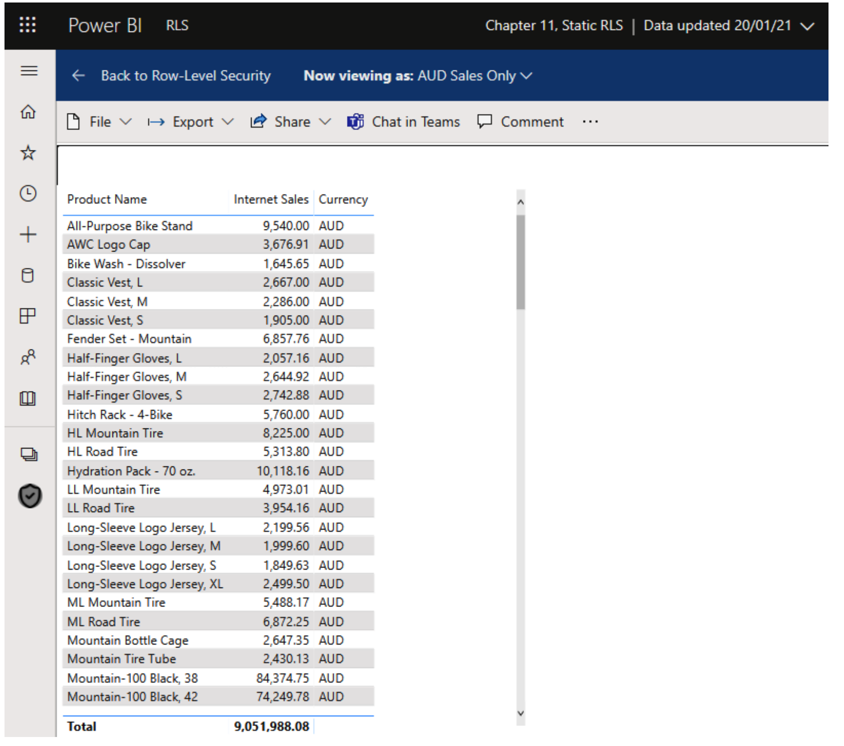


Figure 11.17 – Validating RLS roles in the Power BI service

This opens a report with the selected RLS role-applied filters. The following screenshot shows the validation results for the **AUD Sales Only** role:



Product Name	Internet Sales	Currency
All-Purpose Bike Stand	9,540.00	AUD
AWC Logo Cap	3,676.91	AUD
Bike Wash - Dissolver	1,645.65	AUD
Classic Vest, L	2,667.00	AUD
Classic Vest, M	2,286.00	AUD
Classic Vest, S	1,905.00	AUD
Fender Set - Mountain	6,857.76	AUD
Half-Finger Gloves, L	2,057.16	AUD
Half-Finger Gloves, M	2,644.92	AUD
Half-Finger Gloves, S	2,742.88	AUD
Hitch Rack - 4-Bike	5,760.00	AUD
HL Mountain Tire	8,225.00	AUD
HL Road Tire	5,313.80	AUD
Hydration Pack - 70 oz.	10,118.16	AUD
LL Mountain Tire	4,973.01	AUD
LL Road Tire	3,954.16	AUD
Long-Sleeve Logo Jersey, L	2,199.56	AUD
Long-Sleeve Logo Jersey, M	1,999.60	AUD
Long-Sleeve Logo Jersey, S	1,849.63	AUD
Long-Sleeve Logo Jersey, XL	2,499.50	AUD
ML Mountain Tire	5,488.17	AUD
ML Road Tire	6,872.25	AUD
Mountain Bottle Cage	2,647.35	AUD
Mountain Tire Tube	2,430.13	AUD
Mountain-100 Black, 38	84,374.75	AUD
Mountain-100 Black, 42	74,249.78	AUD
Total	**9,051,988.08**	

Figure 11.18 – The validation results for the AUD Sales Only role

Now that we have learned how to implement a static RLS scenario, let's look at some more complex scenarios that require a dynamic RLS implementation.

Implementing dynamic RLS

When we design and implement RLS, there are many cases where we need to consider dynamic RLS when static RLS does not make that much sense. In this section, we look at several real-world scenarios that require the implementation of dynamic RLS.

Each user can only access their own data

Imagine that we have a sales data model. We need to implement RLS for salespersons so that each salesperson can only see their sales data. Implementing static RLS for such a scenario does not make any sense, primarily when many salespersons work for the business. Are we going to create one static role per salesperson in Power BI Desktop and assign a member to each role in the Power BI service? The answer is obviously no. We can create only one role that works dynamically based on the salesperson's username, one of the easiest and yet most common dynamic RLS requirements to implement. In this section, we use the `Chapter 11, Dynamic RLS.pbix` sample file supplied with this book.

Implementing the preceding scenario is relatively easy. We need to use one of the following DAX functions to retrieve the current username and use it in an RLS role:

- `USERNAME()`: Returns the current user's login name in the form of `DOMAIN_NAME\USER_NAME` when used in Power BI Desktop, such as `biinsight\soheil`. The `USERNAME()` function returns the user's **User Principal Name** (**UPN**) when published to the Power BI service as well as to Power BI Report Server—for example, `soheil@biinsight.com`.
- `USERPRINCIPALNAME()`: Returns the user's UPN at connection time. The UPN is in email format—for example, `soheil@biinsight.com`. The `USERPRINCIPALNAME()` function does not accept any parameters.

Implementing RLS in Power BI only makes sense when we publish the model to the service or Power BI Report Server, therefore using the `USERPRINCIPALNAME()` function is the preferred function.

> **Note**
>
> If we embed Power BI reports to our proprietary application and have to use the user login name, we should use the `USERNAME()` function.

Now, it's time to implement the solution. The steps to create roles and rules, validate roles, publish to the service, and assign members to roles are all the same as we learned before, so we will skip an explanation of those steps in this section. To implement RLS to work dynamically for this scenario, we only need to find the matching value in the `EmailAddress` column from the `Employee` table.

The following steps explain the implementation:

1. Create a role and name it `Salespersons Access`.
2. Create a rule on the `Employee` table over the `EmailAddress` column.
3. Use the `[EmailAddress] = USERPRINCIPALNAME()` DAX expression.
4. Click **Save**.

The preceding steps are highlighted in the following screenshot:

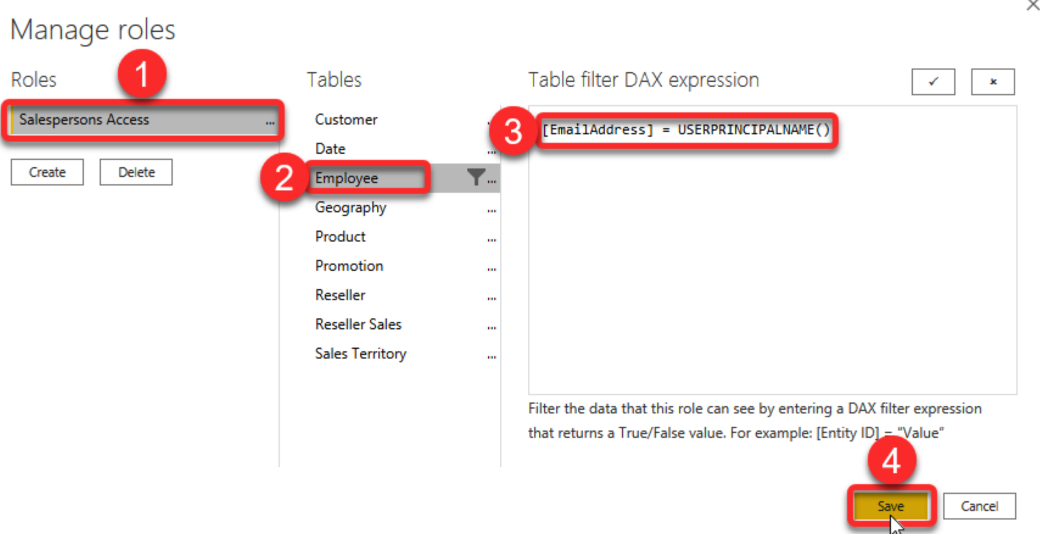


Figure 11.19 – Implementing dynamic RLS to filter the data based on the EmailAddress column

> **Note**
>
> To validate the RLS roles, we do not have to have data visualizations. We can switch to **Data** view then validate the roles.

5. Switch to **Data** view.
6. Click the `Employee` table to see the data within **Data** view.
7. Click the **View as** button from the **Home** tab of the ribbon.
8. Tick the **Other user** option.
9. Type in an email account to test the role.
10. Check the **Salespersons Access** role.
11. Click **OK**.

The following screenshot shows the role validation steps:

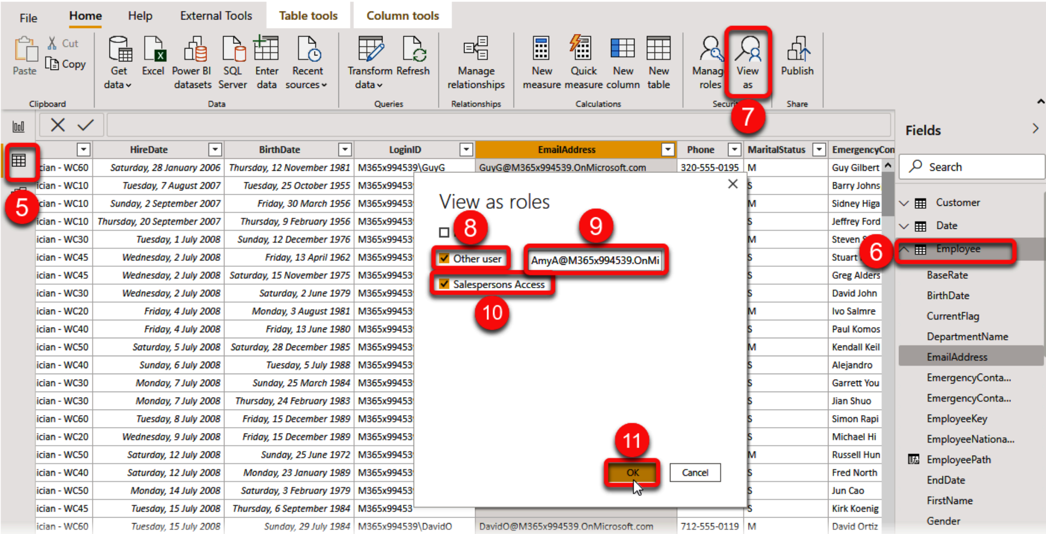


Figure 11.20 – Validating dynamic RLS roles within Data view

The following screenshot shows the result after validating the **Salespersons Access** role:

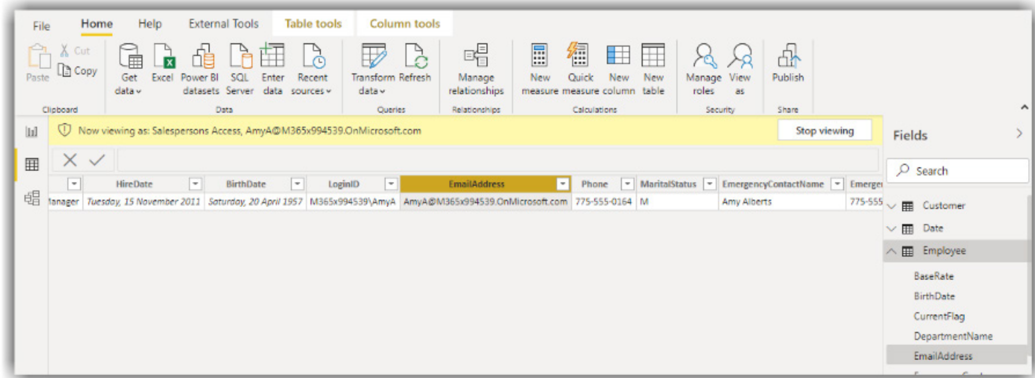


Figure 11.21 – RLS role validation results within Data view

12. If you would like to see how the data changes in your visuals, click **Report** view to see the changes, as highlighted in the following screenshot:

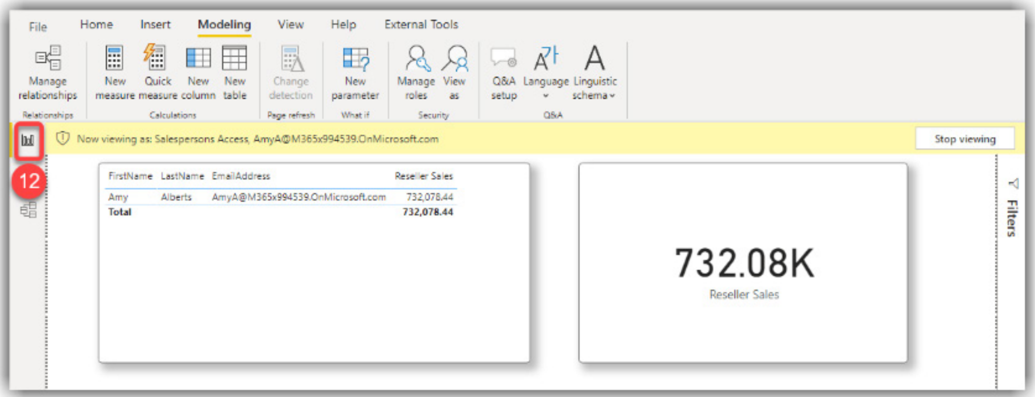


Figure 11.22 – Validating RLS roles in Report view

Now that we are sure that the RLS role works as expected, we can publish the Power BI service report. Suppose we have the correct RLS security settings in the service. In that case, users must see only their sales data when opening the report. The following screenshot shows a couple of examples of this:

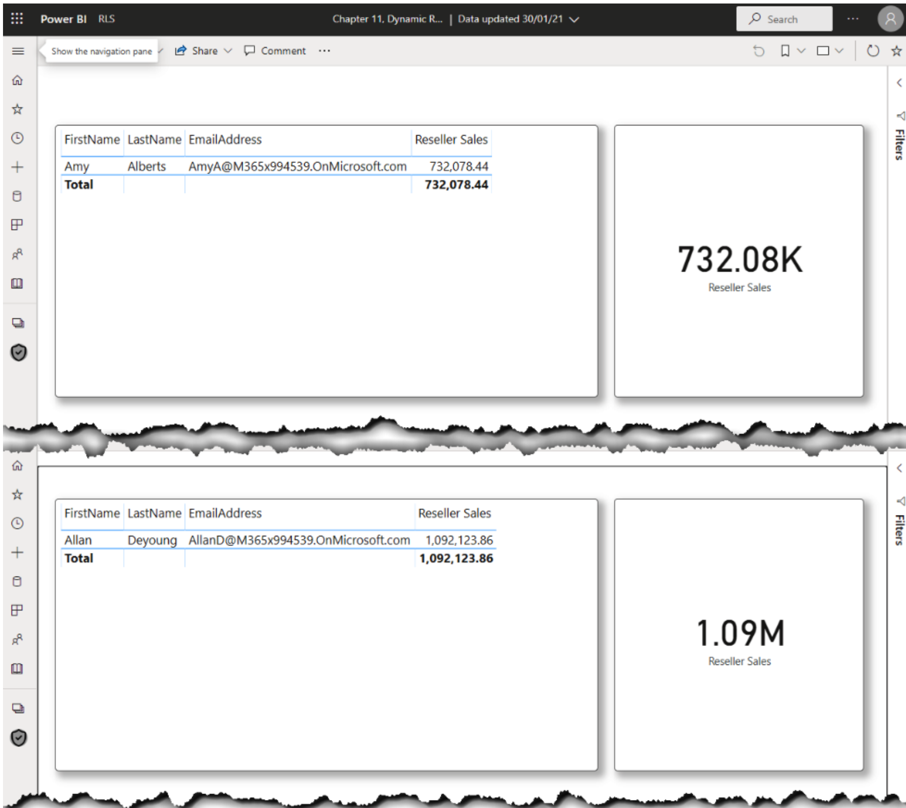


Figure 11.23 – Dynamic RLS automatically kicks in when users open the report

As shown in *Figure 11.23*, when **Allan** opens the report he sees his sales data, and when **Amy** opens the report she only sees her sales data.

Managers can access their team members' data in parent-child hierarchies

So far, we have implemented a straightforward scenario with dynamic RLS to enable salespersons to see the sales data that's relevant to them. Let's now have a look at a more complex scenario. Suppose the business started using the report and everyone is happy apart from the sales managers and—more importantly—the organization's **chief executive officer** (**CEO**). The feedback from sales managers is that they can only see their own data and not that of their team members. More significant is the feedback from the CEO, who cannot see any sales data. The CEO cannot see any data because they are not a salesperson; while they do not sell any products, they still require access to everyone's sales data. The business needs us to implement a dynamic RLS role so that every salesperson can see their sales data. Managers can also see their team members' sales.

Moreover, the CEO can see everyone's sales. To implement this scenario, we need to have an organizational chart handy to create a parent-child hierarchy. The `Employee` table in the data model contains the necessary data we require to implement the preceding scenario, the `EmployeeKey` and the `ParentEmployeeKey` columns to create the **parent-child hierarchy**, and the `EmailAddress` column to identify the users' UPN.

To implement RLS for this scenario, we use the `EmployeeKey` and `ParentEmployeeKey` columns to create a calculated column to identify the parent-child hierarchy **path** using the following DAX expression:

```
EmployeePath = PATH(Employee[EmployeeKey],
Employee[ParentEmployeeKey])
```

The results of the preceding code snippet are shown in the following screenshot:

```
1 EmployeePath = PATH(Employee[EmployeeKey], Employee[ParentEmployeeKey])
```

seRate	VacationHours	SickLeaveHours	CurrentFlag	SalesPersonFlag	DepartmentName	StartDate	EndDate	Status	EmployeePath
12.45	21	30	True	False	Production	Saturday, 28 January 2006		Current	112|23|18|1
13.45	88	64	True	False	Production	Tuesday, 7 August 2007		Current	112|23|189|12
13.45	84	62	True	False	Production	Sunday, 2 September 2007		Current	112|23|189|15
13.45	85	62	True	False	Production	Thursday, 20 September 2007		Current	112|23|189|17
9.5	41	40	True	False	Production	Tuesday, 1 July 2008		Current	112|23|177|22
10	84	62	True	False	Production	Wednesday, 2 July 2008		Current	112|23|201|24
10	85	62	True	False	Production	Wednesday, 2 July 2008		Current	112|23|201|25
9.5	25	32	True	False	Production	Wednesday, 2 July 2008		Current	112|23|188|26
14	9	24	True	False	Production	Friday, 4 July 2008		Current	112|23|111|28
15	68	54	True	False	Production	Friday, 4 July 2008		Current	112|23|89|29
11	11	25	True	False	Production	Saturday, 5 July 2008		Current	112|23|16|31
15	52	46	True	False	Production	Sunday, 6 July 2008		Current	112|23|214|33
9.5	34	37	True	False	Production	Monday, 7 July 2008		Current	112|23|188|34
9.5	36	38	True	False	Production	Monday, 7 July 2008		Current	112|23|138|35
12.45	38	39	True	False	Production	Tuesday, 8 July 2008		Current	112|23|9|39
14	20	30	True	False	Production	Wednesday, 9 July 2008		Current	112|23|186|41
11	6	23	True	False	Production	Saturday, 12 July 2008		Current	112|23|76|45
15	47	43	True	False	Production	Saturday, 12 July 2008		Current	112|23|214|47
11	90	65	True	False	Production	Monday, 14 July 2008		Current	112|23|40|50
10	74	57	True	False	Production	Tuesday, 15 July 2008		Current	112|23|126|54
12.45	33	36	True	False	Production	Tuesday, 15 July 2008		Current	112|23|20|55
14	15	27	True	False	Production	Wednesday, 16 July 2008		Current	112|23|147|57
15	58	49	True	False	Production	Thursday, 17 July 2008		Current	112|23|214|58
12.45	17	28	True	False	Production	Thursday, 17 July 2008		Current	112|23|18|59
11	93	66	True	False	Production	Thursday, 17 July 2008		Current	112|23|40|60
11	1	20	True	False	Production	Saturday, 19 July 2008		Current	112|23|76|62
9.5	26	33	True	False	Production	Monday, 21 July 2008		Current	112|23|188|64
15	69	54	True	False	Production	Tuesday, 22 July 2008		Current	112|23|89|65

Fields
Search
Customer
Date
Employee
BaseRate
BirthDate
CurrentFlag
DepartmentName
EmailAddress
EmergencyConta...
EmergencyConta...
EmployeeKey
EmployeeNationa...
EmployeePath
EndDate
FirstName
Gender
HireDate
LastName
LoginID
MaritalStatus
MiddleName
NameStyle
ParentEmployeeK...

Figure 11.24 – Creating the EmployeePath calculated column in the Employee table

The next step is to identify the `EmployeeKey` value based on the user's UPN.

The following DAX expression retrieves the `EmployeeKey` value of the current user based on their UPN:

```
CALCULATETABLE(
      VALUES(Employee[EmployeeKey])
      , FILTER(Employee, Employee[EmailAddress] =
USERPRINCIPALNAME())
      , FILTER(Employee, Employee[Status] = "Current")
)
```

> **Note**
> The `Employee` table keeps a record of an employee's employment history, so we always want to get the `EmployeeKey` value for employees when their status is `Current`. Therefore, we need to check the `Status` column as well.

To test the previous code, we can create a measure using this then validate it, just like when validating any other RLS roles. The following screenshot shows the result of using the same code in a measure used in a card visual:

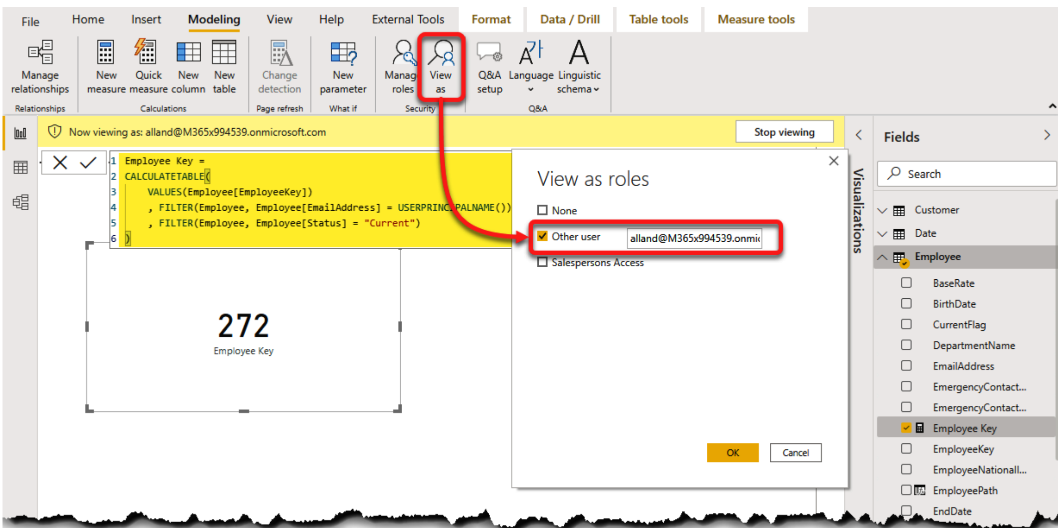


Figure 11.25 – Validating DAX expressions for RLS in a measure

> **Note**
> Remember to remove the `EmployeeKey` measure. We created it for testing purposes only.

Now that we have retrieved the `EmployeeKey` value, we can use the result to find all rows where an `EmployeeKey` value appears within the `EmployeePath` calculated column. To understand the scenario, let's have a closer look at the data. The following screenshot shows all salespersons' sales, their `EmployeeKey` values, names, email addresses, and `EmployeePath` values:

EmployeeKey	FirstName	LastName	Title	EmailAddress	EmployeePath	Reseller Sales
272	Allan	Deyoung	North American Sales Manager	AllanD@M365x994539.OnMicrosoft.com	112|277|272	1,092,123.86
290	Amy	Alberts	European Sales Manager	AmyA@M365x994539.OnMicrosoft.com	112|277|290	732,078.44
289	David	Campbell	Sales Representative	DavidC@M365x994539.OnMicrosoft.com	112|277|272|289	3,729,945.35
284	Garrett	Vargas	Sales Representative	GarrettV@M365x994539.OnMicrosoft.com	112|277|272|284	3,609,447.22
291	Jae	Pak	Sales Representative	JaeP@M365x994539.OnMicrosoft.com	112|277|290|291	8,503,338.65
283	Jillian	Carson	Sales Representative	JillianC@M365x994539.OnMicrosoft.com	112|277|272|283	10,065,803.54
288	José	Saraiva	Sales Representative	JoséS@M365x994539.OnMicrosoft.com	112|277|272|288	5,926,418.36
282	Linda	Mitchell	Sales Representative	LindaM@M365x994539.OnMicrosoft.com	112|277|272|282	10,367,007.43
296	Lynn	Tsoflias	Sales Representative	LynnT@M365x994539.OnMicrosoft.com	112|277|294|296	1,421,810.93
281	Michael	Blythe	Sales Representative	MichaelB@M365x994539.OnMicrosoft.com	112|277|272|281	9,293,903.01
286	Pamela	Ansman-Wolfe	Sales Representative	PamelaA@M365x994539.OnMicrosoft.com	112|277|272|286	3,325,102.60
295	Rachel	Valdez	Sales Representative	RachelV@M365x994539.OnMicrosoft.com	112|277|290|295	1,790,640.23
292	Ranjit	Varkey Chudukatil	Sales Representative	RanjitV@M365x994539.OnMicrosoft.com	112|277|290|292	4,509,888.93
287	Shu	Ito	Sales Representative	ShuI@M365x994539.OnMicrosoft.com	112|277|272|287	6,427,005.56
294	Syed	Abbas	Pacific Sales Manager	SyedA@M365x994539.OnMicrosoft.com	112|277|294	172,524.45
293	Tete	Mensa-Annan	Sales Representative	TeteM@M365x994539.OnMicrosoft.com	112|277|272|293	2,312,545.69
285	Tsvi	Reiter	Sales Representative	TsviR@M365x994539.OnMicrosoft.com	112|277|272|285	7,171,012.75
Total						**80,450,596.98**

Figure 11.26 – Salespersons' sales

As the preceding screenshot shows, **Allan**, **Amy**, and **Syed** are sales managers. The sales managers are also salespersons themselves, so they also sell products. The `EmployeePath` column's values reveal that the sales managers are in the third level of the organizational chart, as their employee keys appear in the third position within the `EmployeePath` values, so they must see all sales data of their team members in which their employee keys appear after the sales managers. Needless to mention that the person with an employee key of `277`, which is one level below the person with an employee key of `112`, must see all sales data. The person with an employee key of `112` is the CEO. To implement this scenario, we must find each person's employee key within the `EmployeePath` value for each row. The good news is that there is a specific function in DAX to find values in a **parent-child path**: `PATHCONTAINS(<Path to lookup>, <value to be found within the Path>)`. The `PATHCONTAINS()` function returns `True` if the specified value appears within the path. We already have the path values within the `EmployeePath` column. We also authored the DAX expression to retrieve the employee key of the current user. This is the value to be found within the path. So, the DAX expression looks like this:

```
VAR _key = CALCULATETABLE (
    VALUES ( Employee[EmployeeKey] ),
```

```
    FILTER ( Employee, Employee[EmailAddress] =
USERPRINCIPALNAME()),
    FILTER ( Employee, Employee[Status] = "Current" )
)
RETURN
PATHCONTAINS (
            Employee[EmployeePath], _key
)
```

The only remaining part is to create a role and use the preceding expression as a rule. The following screenshot shows that we created a new role, **Sales Team**, with the preceding DAX expression as its rule:

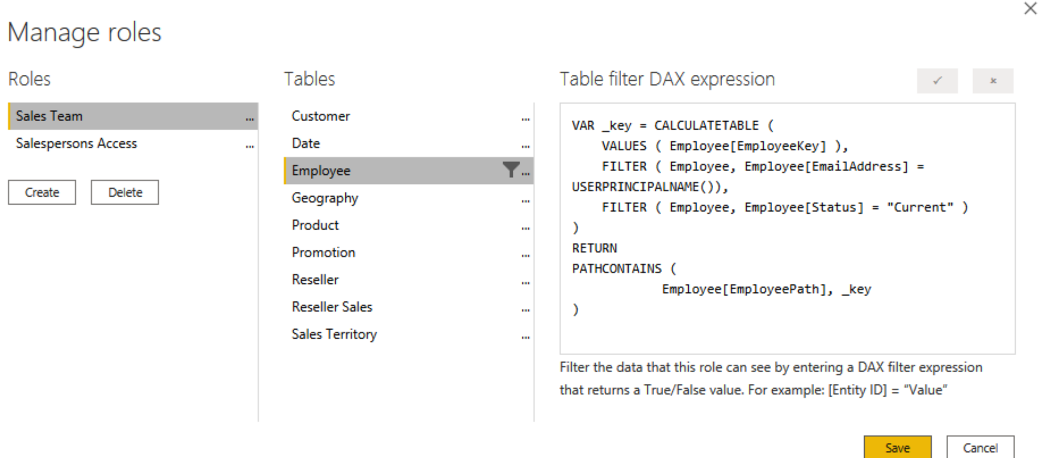
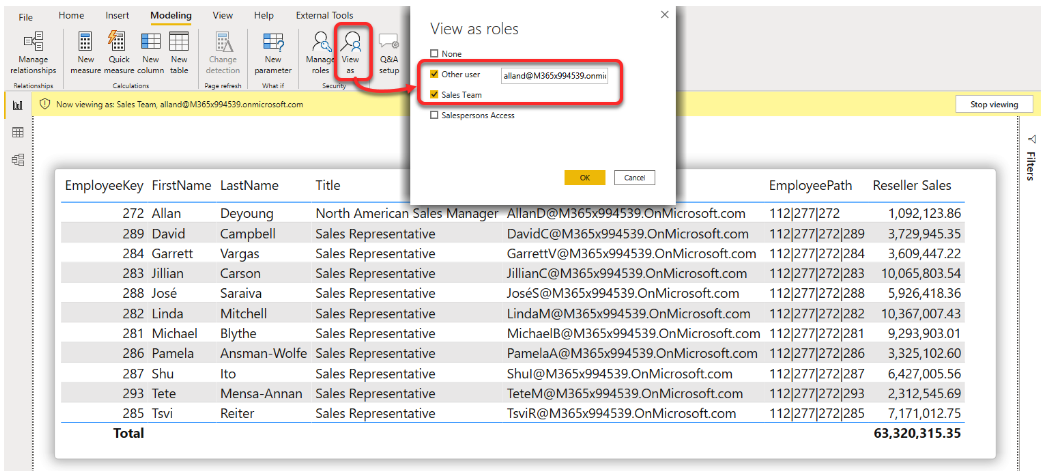


Figure 11.27 – Creating a Sales Team role for dynamic RLS

Now, we validate the **Sales Team** role for one of the managers. The following screenshot shows the validation results when **Allan** uses the report:



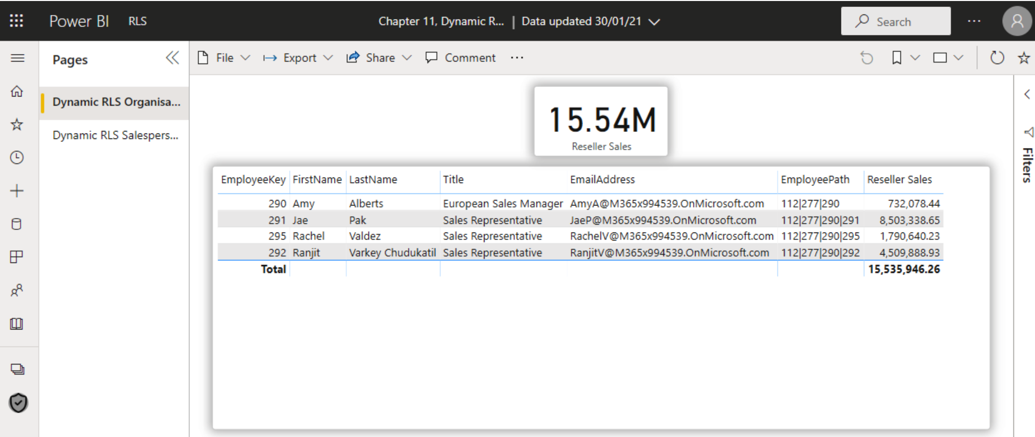
EmployeeKey	FirstName	LastName	Title		EmployeePath	Reseller Sales
272	Allan	Deyoung	North American Sales Manager	AllanD@M365x994539.OnMicrosoft.com	112|277|272	1,092,123.86
289	David	Campbell	Sales Representative	DavidC@M365x994539.OnMicrosoft.com	112|277|272|289	3,729,945.35
284	Garrett	Vargas	Sales Representative	GarrettV@M365x994539.OnMicrosoft.com	112|277|272|284	3,609,447.22
283	Jillian	Carson	Sales Representative	JillianC@M365x994539.OnMicrosoft.com	112|277|272|283	10,065,803.54
288	José	Saraiva	Sales Representative	JoséS@M365x994539.OnMicrosoft.com	112|277|272|288	5,926,418.36
282	Linda	Mitchell	Sales Representative	LindaM@M365x994539.OnMicrosoft.com	112|277|272|282	10,367,007.43
281	Michael	Blythe	Sales Representative	MichaelB@M365x994539.OnMicrosoft.com	112|277|272|281	9,293,903.01
286	Pamela	Ansman-Wolfe	Sales Representative	PamelaA@M365x994539.OnMicrosoft.com	112|277|272|286	3,325,102.60
287	Shu	Ito	Sales Representative	ShuI@M365x994539.OnMicrosoft.com	112|277|272|287	6,427,005.56
293	Tete	Mensa-Annan	Sales Representative	TeteM@M365x994539.OnMicrosoft.com	112|277|272|293	2,312,545.69
285	Tsvi	Reiter	Sales Representative	TsviR@M365x994539.OnMicrosoft.com	112|277|272|285	7,171,012.75
Total						**63,320,315.35**

Figure 11.28 – Validating dynamic RLS, enabling sales managers to see their team members' sales

We can now publish the report to the service.

> **Remember**
>
> If you are a Power BI tenant administrator, you need to assign members to the new **Sales Team** role. Otherwise, ask your Power BI administrator to do so.

Now, when staff use the report, they can access all their team members' sales if they are a sales manager. The following screenshot shows what **Amy** sees after opening the report:



EmployeeKey	FirstName	LastName	Title	EmailAddress	EmployeePath	Reseller Sales
290	Amy	Alberts	European Sales Manager	AmyA@M365x994539.OnMicrosoft.com	112|277|290	732,078.44
291	Jae	Pak	Sales Representative	JaeP@M365x994539.OnMicrosoft.com	112|277|290|291	8,503,338.65
295	Rachel	Valdez	Sales Representative	RachelV@M365x994539.OnMicrosoft.com	112|277|290|295	1,790,640.23
292	Ranjit	Varkey Chudukatil	Sales Representative	RanjitV@M365x994539.OnMicrosoft.com	112|277|290|292	4,509,888.93
Total						**15,535,946.26**

Figure 11.29 – When users open the report in the Power BI service, the Sales Team RLS role kicks in

Now, let's see what the CEO can see after opening the report. As a reminder, the CEO can see everyone's sales data, as illustrated in the following screenshot:

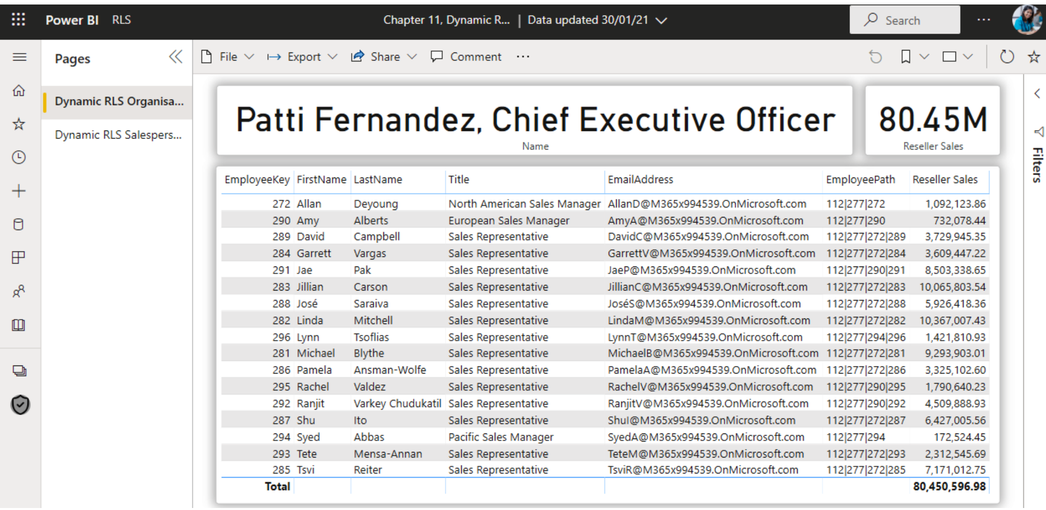


Figure 11.30 – The CEO can see everyone's data

While the preceding scenario was rather more complex than the first one, there is always a trickier scenario. You must have noted that the `Employee` table contains the users' login data to implement the previous scenarios, such as the `EmailAddress` value. In the next section, we learn how to implement scenarios when the email address data does not exist in the `Employee` table.

Getting the user's login data from another source

Suppose we have a scenario where the business has the same requirement as outlined previously; however, the source system does not contain the `EmailAddress` column in the `Employee` table. In that case, the scenario is different. We need to get the users' login data from a different source. While the new data source provides users' login data, it may not necessarily have an `EmployeeKey` column to relate employees to their login data. Depending on the source system, we may get a very different set of data. In our scenario, we asked the system administrators to give us an extract of the organization's **Azure AD** users. They provided a **JavaScript Object Notation** (**JSON**) file containing a list of all users' UPNs. You can find the file in this book's resources (`Chapter 11, Adventure Works, AAD UPNs.json`).

> **Note**
>
> In real-world scenarios, we have to think about automating the process of generating the JSON file to keep it up to date.
>
> The generated JSON file contains sensitive data, therefore it must be stored in a secured location accessible by a restricted number of users.

For convenience, we use the same Power BI file as in the previous scenario. We just ignore the `EmailAddress` column in the `Employee` table.

The following steps explain how to overcome this challenge:

1. In Power BI Desktop, get the data from the provided **JSON** file, as illustrated in the following screenshot:

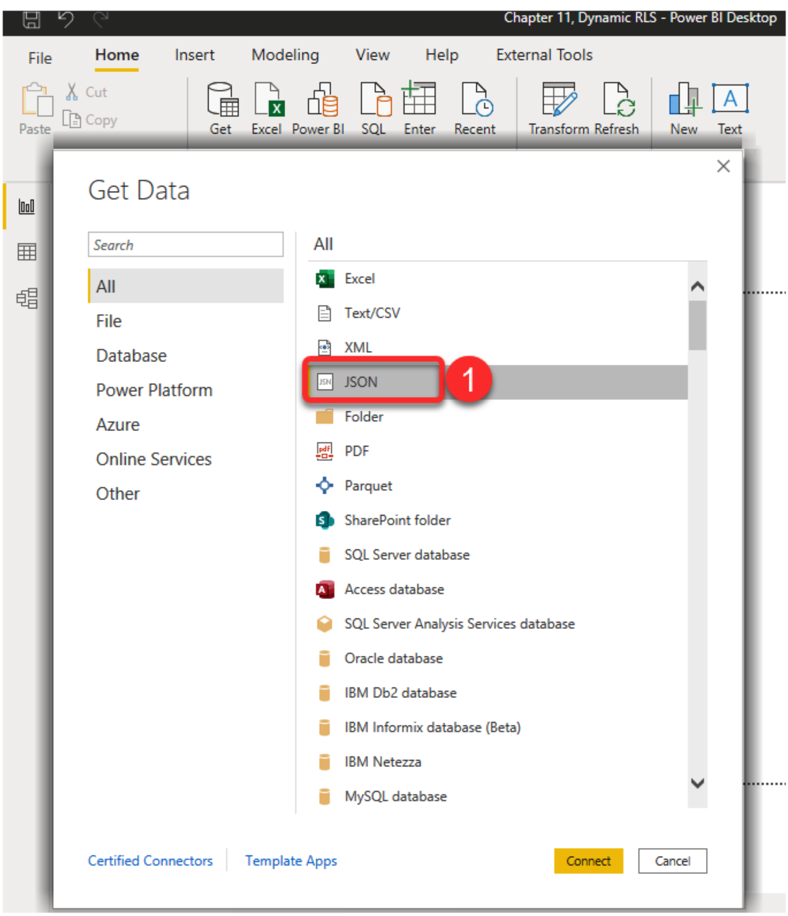


Figure 11.31 – Getting data from JSON file

2. Power BI detects the contents automatically and expands the values, so the results look like this:

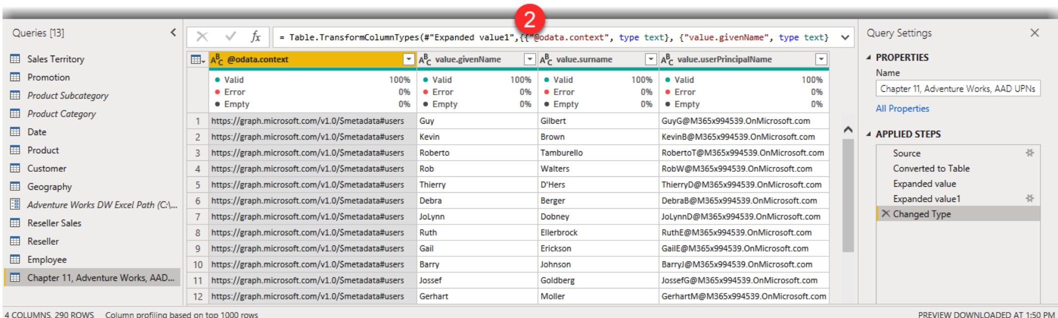


Figure 11.32 – Power BI automatically detects the JSON file's contents

3. Rename the query as `Users`.
4. Remove the `@odata.context` column.
5. Rename the `value.givenName` column as `First Name`, the `value.surname` column as `Last Name`, and the `value.userPrincipalName` column as `Email Address`. The results look like this:

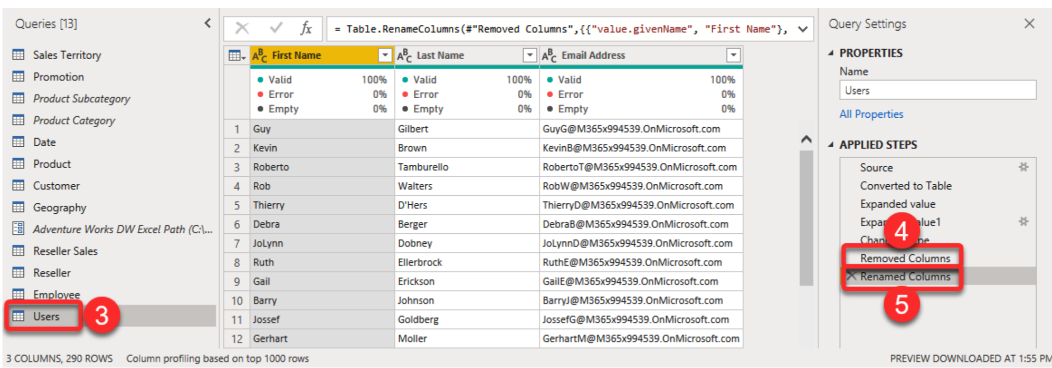


Figure 11.33 – Preparing the Users table

As you see, we do not have the `EmployeeKey` column in the `Users` table. However, we can get the `EmployeeKey` value from the `Employee` table by finding the matching values based on the `First Name` and `Last Name` columns. The following steps show how to do this:

6. **Merge** the `Users` table with the `Employee` table on the `First Name` and `Last Name` columns from the `User` and `FirstName` and `LastName` columns from the `Employee` table. Set the **Join Kind** value to **Left Outer (all from first, matching second)**, as illustrated in the following screenshot:

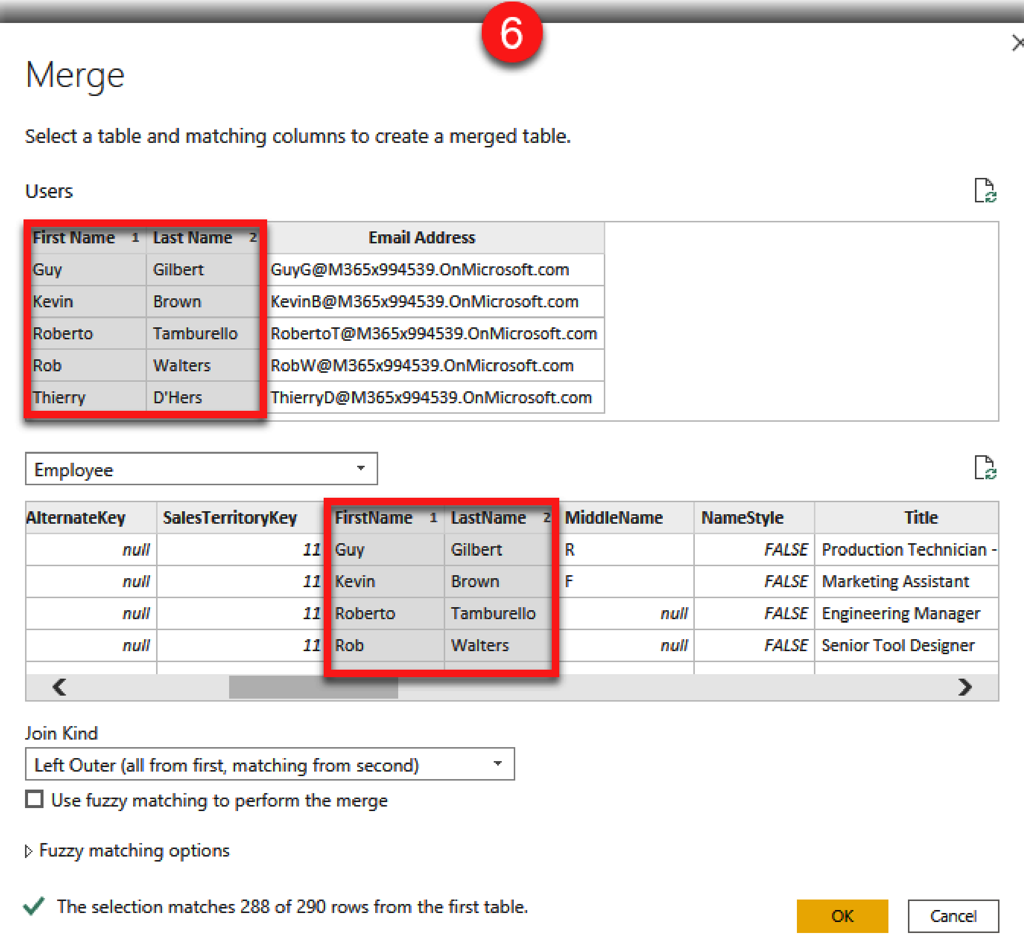


Figure 11.34 – Merging the Users table with the Employee table

7. Expand the `Employee` structured column to keep the `EmployeeKey` and `Status` columns, as illustrated in the following screenshot:

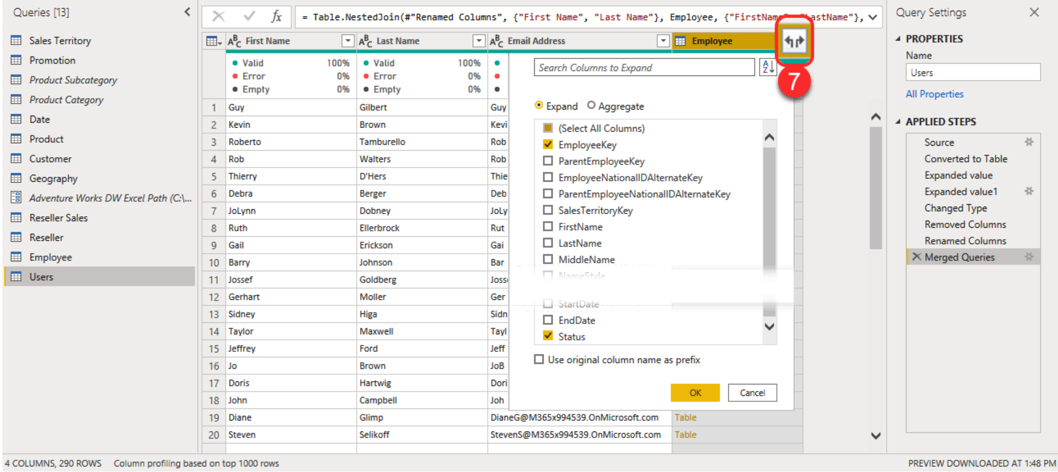


Figure 11.35 – Expanding the Employee structured column

> **Note**
>
> Remember—the `Employee` table keeps the employees' history, which means we can potentially duplicate values. Therefore, we need to keep the `Status` column to make sure all employees' statuses are `Current`.

8. Filter the **Status** column to only show rows with a `Current` status, as illustrated in the following screenshot:

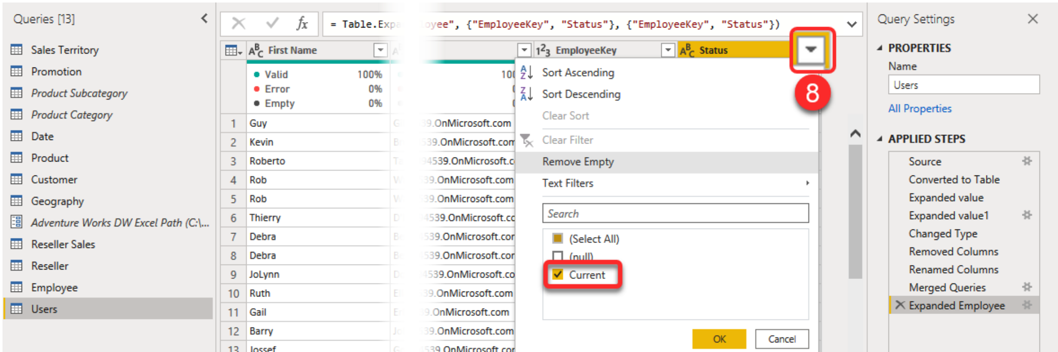


Figure 11.36 – Filtering the Employee table to show the rows with a Current status

9. Click the **Close & Apply** button to load the `Users` table into the data model, as illustrated in the following screenshot:

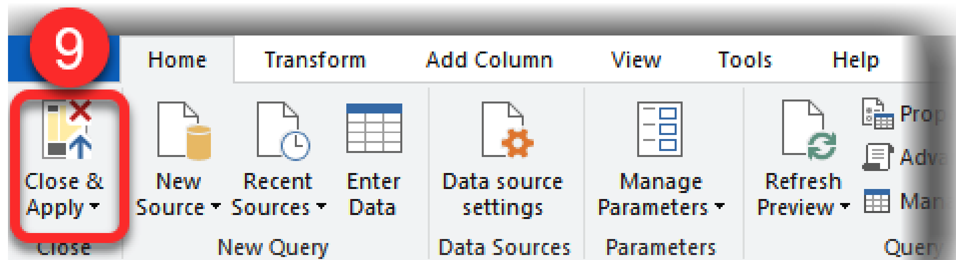


Figure 11.37 – Applying the changes and loading the Users table into the data model

10. Switch to **Model** view.

11. Create a new relationship between the `Users` table and the `Employee` table if Power BI Desktop has not automatically detected the relationship, as illustrated in the following screenshot:

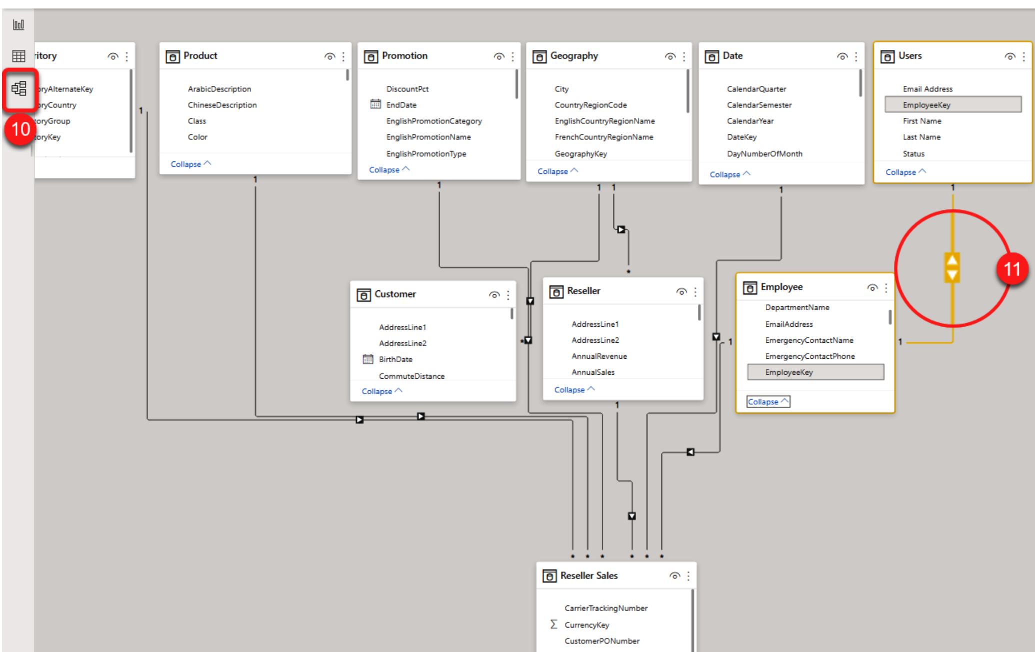


Figure 11.38 – Creating a relationship between the Users and Employee tables

As you see in the preceding screenshot, the relationship between the `Users` and the `Employee` tables is a one-to-one relationship, which is precisely what we are after. Now, we can create a new role on the `Employee` table, as explained in the following steps:

12. Create a new RLS role and name it `Sales Team2`, then create a new rule on the `Employee` table using the following DAX expression, as shown in *Figure 11.39*:

```
VAR _key = CALCULATETABLE (
            VALUES ( Users[EmployeeKey] ),
            FILTER ( Users, Users[Email Address] =
USERPRINCIPALNAME())
)
RETURN
PATHCONTAINS (
              Employee[EmployeePath], _key)
```

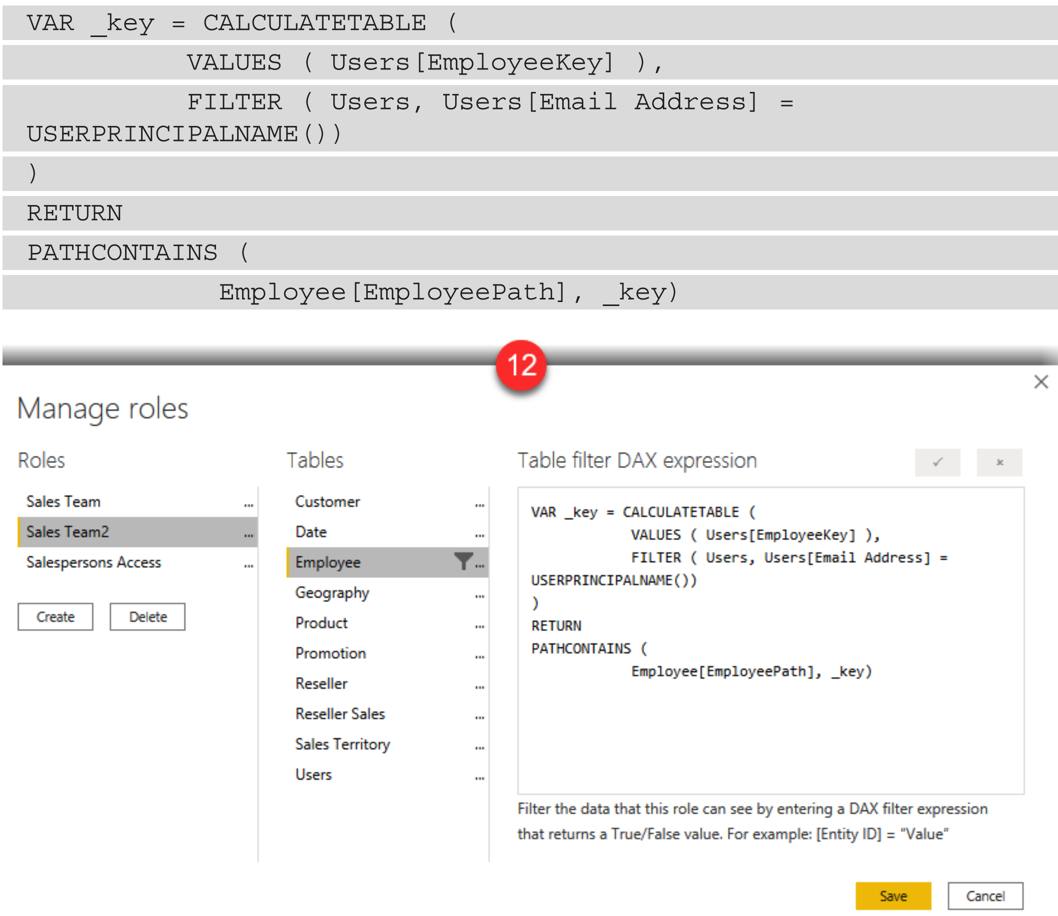


Figure 11.39 – Creating a new RLS role and rule on the Employee table

> **Note**
>
> As we already filtered all rows in the `Users` table to only show the current users, we do not need to add an additional `FILTER()` function in the `CALCULATETABLE()` code block.

13. Last, but not least, is to validate the role by clicking the **View as** button from the **Modeling** tab of the ribbon, as illustrated in the following screenshot:

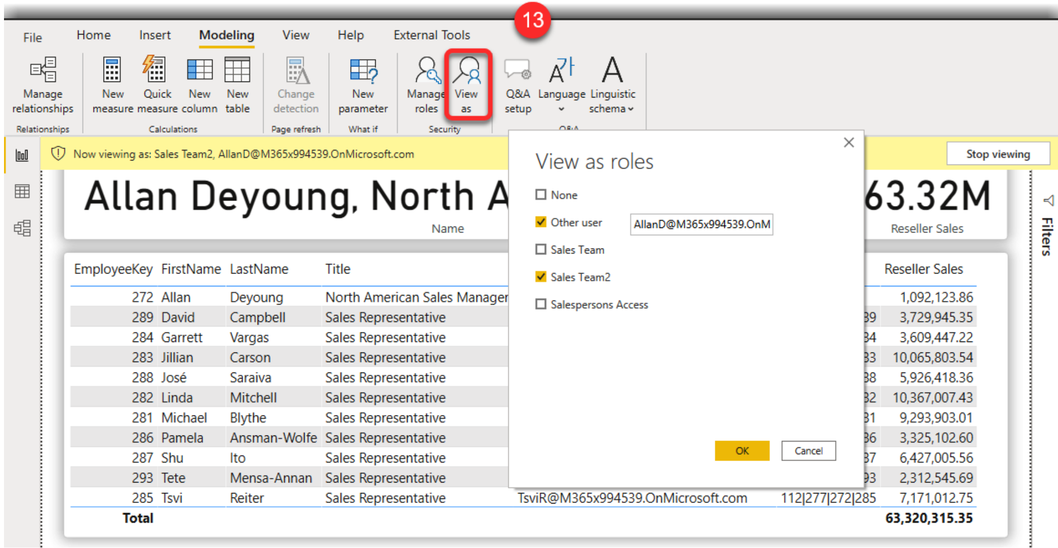


Figure 11.40 – Validating the new RLS role

So far, we have learned how to deal with business security requirements. We learned that to implement some requirements, we have to make some changes in our data model. While we covered some common scenarios, there are many more complex scenarios that we cannot cover in a single chapter, so we leave it to you to investigate more.

Summary

In this chapter, we learned how to implement RLS in Power BI Desktop and manage the roles within the Power BI service as well as in Power BI Report Server. We learned what static RLS and dynamic RLS approaches are and how to implement them in our data model in order to make the relevant data available only to the relevant people. In the next chapter, *Extra Options and Features Available for Data Modeling*, we look at slowly changing dimensions, OLS, dataflows, and composite models.

12
Extra Options and Features Available for Data Modeling

In the previous 11 chapters, we learned many aspects of data modeling to model many different scenarios. This chapter provides brief introductions to some more options available in the Power BI platform that can come in handy in many real-world scenarios. All of the topics discussed in this chapter are too extensive to go through each one in too much detail, but it is nonetheless worthwhile having exposure to them all. In this chapter, we cover the following areas:

- Dealing with **slowly changing dimensions** (**SCDs**)
- Introduction to **object-level security** (**OLS**)
- Introduction to dataflows
- Introduction to composite models

Dealing with SCDs

The term **slowly changing dimension**, which we usually refer to with its short form, **SCD**, is a data warehousing concept introduced by the amazing Ralph Kimball. You can learn more about Ralph Kimball here: `https://www.kimballgroup.com/about-kimball-group/`.

The SCD concept deals with moving a specific set of data from one state to another. Imagine we have a **human resources** (**HR**) system; `Stephen Jiang` is a `Sales Manager`, having 10 sales representatives in his team. The following screenshot shows the sample data for our scenario:

Full Name	Title	Manager
Stephen Jiang	North American Sales Manager	Brian Welcker
David Campbell	Sales Representative	Stephen Jiang
Garrett Vargas	Sales Representative	Stephen Jiang
Jillian Carson	Sales Representative	Stephen Jiang
José Saraiva	Sales Representative	Stephen Jiang
Linda Mitchell	Sales Representative	Stephen Jiang
Michael Blythe	Sales Representative	Stephen Jiang
Pamela Ansman-Wolfe	Sales Representative	Stephen Jiang
Shu Ito	Sales Representative	Stephen Jiang
Tete Mensa-Annan	Sales Representative	Stephen Jiang
Tsvi Reiter	Sales Representative	Stephen Jiang

Figure 12.1 – Stephen Jiang is the sales manager of a team of 10 sales representatives

Today, `Stephen Jiang` has been promoted to `Vice President of Sales`, so his team grows in size from 10 to 17. The following screenshot shows the changes:

Full Name	Title	Manager
Stephen Jiang	Vice President of Sales	Patti Fernandez
Amy Alberts	European Sales Manager	Stephen Jiang
Roger Hamilton	North American Sales Manager	Stephen Jiang
Syed Abbas	Pacific Sales Manager	Stephen Jiang
David Campbell	Sales Representative	Roger Hamilton
Garrett Vargas	Sales Representative	Roger Hamilton
Jae Pak	Sales Representative	Amy Alberts
Jillian Carson	Sales Representative	Roger Hamilton
José Saraiva	Sales Representative	Roger Hamilton
Linda Mitchell	Sales Representative	Roger Hamilton
Lynn Tsoflias	Sales Representative	Syed Abbas
Michael Blythe	Sales Representative	Roger Hamilton
Pamela Ansman-Wolfe	Sales Representative	Roger Hamilton
Rachel Valdez	Sales Representative	Amy Alberts
Ranjit Varkey Chudukatil	Sales Representative	Amy Alberts
Shu Ito	Sales Representative	Roger Hamilton
Tete Mensa-Annan	Sales Representative	Roger Hamilton
Tsvi Reiter	Sales Representative	Roger Hamilton

Figure 12.2 – Stephen's team after he was promoted to Vice President of Sales

Another example is when a customer's address changes in the sales system. Again, the customer is the same, but their address is now different. Depending on the business requirements, we have some options to deal with such situations from a data warehousing standpoint—this leads us to different types of SDCs. Just keep in mind that the data changes are happening in the source systems (in our examples, the HR system or a sales system), which are transactional. Then, we transform and move the data from the transactional systems via **extract, transform, and load** (**ETL**) processes and land the transformed data into a data warehouse, which is where the concept of SCD kicks in. SCD is about how changes in the source systems reflect the data in the data warehouse. These kinds of changes in the source system are not something that happens very often, hence the term slowly changing. Many SCD types have been developed over the years, which means that covering them is out the scope of this book, but for your reference, we cover the first three types as follows.

SCD type zero (SCD 0)

With this type of SCD, we ignore all changes in a dimension. So, when a person's residential address changes in the source system (an HR system, in our example), we do not change the landing dimension in our data warehouse. In other words, we ignore the changes within the data source. SCD 0 is also referred to as **fixed dimensions**.

SCD type 1 (SCD 1)

With a SCD 1 type, we overwrite the old data with the new. An excellent example of an SCD 1 type is when the business does not need to have the customer's old address and only needs to keep the customer's current address.

SCD type 2 (SCD 2)

With this type of SCD, we keep the history of data changes in the data warehouse when the business needs to keep the customer's old address and the current address. In an SCD 2 scenario we need to maintain history, so we insert a new row of data into the data warehouse whenever a change happens in the transactional system. Inserting a new row of data causes data duplications in the data warehouse, which means that we cannot use the `CustomerKey` column as the primary key of the dimension. Hence, we need to introduce a new set of columns, as follows:

- A new key column that guarantees rows' uniqueness in the `Customers` dimension. This new key column is simply an index representing each row of data stored in a data warehouse dimension. The new key is a so-called **surrogate key**. While the Surrogate Key guarantees each row in the dimension is unique, we still need to maintain the source system's primary key. By definition, the source system's primary keys are now called **business keys** or **alternate keys** in the data warehousing world.
- A `Start Date` and an `End Date` column to represent the timeframe during which a row of data is in its current state.
- Another column that shows the status of each row of data.

SCD 2 is the most common type of SCD.

Let's revisit our previous example when `Stephen Jiang` was promoted from `Sales Manager` to `Vice President of Sales`. The following screenshot shows the data before Stephen got the promotion:

EmployeeKey	Full Name	Title	Manager	EmployeeBusinessKey	Start Date	End Date	Status
272	Stephen Jiang	North American Sales Manager	Stephen Jiang	502097814	4/08/2010		Current
273	Wanida Benshoof	Marketing Assistant	Debra Berger	323403273	7/08/2010		Current
274	Sharon Salavaria	Design Engineer	Roberto Tamburello	56920285	18/08/2010		Current
275	John Wood	Marketing Specialist	Debra Berger	222969461	7/09/2010		Current
276	Mary Dempsey	Marketing Assistant	Debra Berger	52541318	14/09/2010		Current

Figure 12.3 – The employee data before Stephen was promoted

In the preceding screenshot, the `EmployeeKey` column is the **Surrogate Key** of the dimension and the `EmployeeBusinessKey` column is the **Business Key** (the primary key of the customer in the source system), the `Start Date` column shows the date `Stephen Jiang` started his job as `North American Sales Manager`, the `End Date` column has been left blank (null), and the `Status` column shows `Current`. Now, let's have a look at the data after Stephen gets the promotion, which is illustrated in the following screenshot:

EmployeeKey	Full Name	Title	Manager	EmployeeBusinessKey	Start Date	End Date	Status
272	Stephen Jiang	North American Sales Manager	Brian Welcker	502097814	4/08/2010	12/10/2012	
273	Wanida Benshoof	Marketing Assistant	Debra Berger	323403273	7/08/2010		Current
274	Sharon Salavaria	Design Engineer	Roberto Tamburello	56920285	18/08/2010		Current
275	John Wood	Marketing Specialist	Debra Berger	222969461	7/09/2010		Current
276	Mary Dempsey	Marketing Assistant	Debra Berger	52541318	14/09/2010		Current
277	Stephen Jiang	Vice President of Sales	Patti Fernandez	502097814	13/10/2012		Current

Figure 12.4 – The employee data after Stephen gets promoted

As *Figure 12.4* shows, `Stephan Jiang` started his new role as `Vice President of Sales` on `13/10/2012` and finished his job as `North American Sales Manager` on `12/10/2012`.

Let's see what SCD 2 means when it comes to data modeling in Power BI. The first question is: *Can we implement SCD 2 directly in Power BI Desktop without having a data warehouse?* To answer this question, we have to remember that we create a semantic layer when we build a data model in Power BI. The semantic layer, by definition, is a view of the source data (usually a data warehouse), optimized for reporting and analytical purposes. The semantic layer is not there to replace a data warehouse or another version of a data warehouse. Suppose the business needs to keep a history of changes. In that case, we either need to have a data warehouse, or the transactional system has to find a way to maintain historical data, such as a **temporal** mechanism. A temporal mechanism is a feature that some relational database management systems such as SQL Server offer to provide information about the data kept in a table at any point in time, instead of keeping the current data only. To learn more **temporal tables** in SQL Server, check out the following link: `https://docs.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables?view=sql-server-ver15&WT.mc_id=5003466`.

After we load the data into the data model in Power BI Desktop, we have all current and historical data in the dimension tables. Therefore, we have to be careful when dealing with SCDs. For instance, the following screenshot shows reseller sales for employees:

EmployeeKey	Full Name	Manager	Sales
290	Amy Alberts	Stephen Jiang	$732,078.4446
289	David Campbell	Roger Hamilton	$3,729,945.3501
284	Garrett Vargas	Roger Hamilton	$3,609,447.2163
291	Jae Pak	Amy Alberts	$8,503,338.6472
283	Jillian Carson	Roger Hamilton	$10,065,803.5429
288	José Saraiva	Roger Hamilton	$5,926,418.3574
282	Linda Mitchell	Roger Hamilton	$10,367,007.4286
296	Lynn Tsoflias	Syed Abbas	$1,421,810.9252
281	Michael Blythe	Roger Hamilton	$9,293,903.0055
286	Pamela Ansman-Wolfe	Roger Hamilton	$3,325,102.5952
295	Rachel Valdez	Amy Alberts	$1,790,640.2311
292	Ranjit Varkey Chudukatil	Amy Alberts	$4,509,888.933
287	Shu Ito	Roger Hamilton	$6,427,005.5556
272	Stephen Jiang	Brian Welcker	$1,092,123.8562
294	Syed Abbas	Stephen Jiang	$172,524.4515
293	Tete Mensa-Annan	Roger Hamilton	$2,312,545.6905
285	Tsvi Reiter	Roger Hamilton	$7,171,012.7514
Total			**$80,450,596.9823**

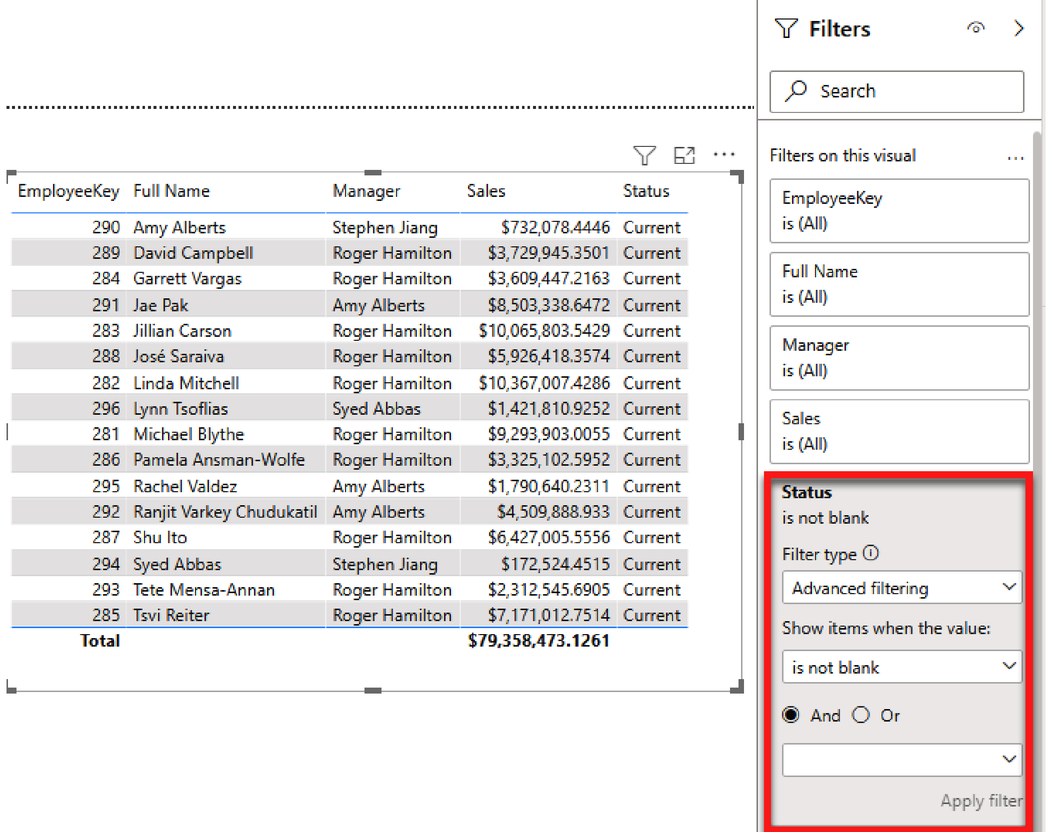
Figure 12.5 – Reseller sales for employees without considering SCD

At a first glance, the numbers seem to be right. Well, they may be right; they may be wrong. It depends on what the business expects to see on a report. We did not consider SCD when we created the preceding table, which means we consider Stephen's sales values (`EmployeeKey 277`). But is this what the business requires? Does the business expect to see all employees' sales without considering their status? For more clarity, let's add the `Status` column to the table. The following screenshot shows the same values as those displayed in *Figure 12.5*:

EmployeeKey	Full Name	Manager	Sales	Status
290	Amy Alberts	Stephen Jiang	$732,078.4446	Current
289	David Campbell	Roger Hamilton	$3,729,945.3501	Current
284	Garrett Vargas	Roger Hamilton	$3,609,447.2163	Current
291	Jae Pak	Amy Alberts	$8,503,338.6472	Current
283	Jillian Carson	Roger Hamilton	$10,065,803.5429	Current
288	José Saraiva	Roger Hamilton	$5,926,418.3574	Current
282	Linda Mitchell	Roger Hamilton	$10,367,007.4286	Current
296	Lynn Tsoflias	Syed Abbas	$1,421,810.9252	Current
281	Michael Blythe	Roger Hamilton	$9,293,903.0055	Current
286	Pamela Ansman-Wolfe	Roger Hamilton	$3,325,102.5952	Current
295	Rachel Valdez	Amy Alberts	$1,790,640.2311	Current
292	Ranjit Varkey Chudukatil	Amy Alberts	$4,509,888.933	Current
287	Shu Ito	Roger Hamilton	$6,427,005.5556	Current
272	Stephen Jiang	Brian Welcker	$1,092,123.8562	
294	Syed Abbas	Stephen Jiang	$172,524.4515	Current
293	Tete Mensa-Annan	Roger Hamilton	$2,312,545.6905	Current
285	Tsvi Reiter	Roger Hamilton	$7,171,012.7514	Current
Total			**$80,450,596.9823**	

Figure 12.6 – Reseller sales for employees and their status without considering SCD

What if the business needs to only show sales values only for employees when their status is `Current`? In that case, we would have to factor the SCD into the equation and filter out Stephen's sales values. Depending on the business requirements, we might need to add the `Status` column as a filter in the visualizations, while in other cases, we might need to modify the measures by adding the `Start Date`, `End Date`, and `Status` columns to filter the results. The following screenshot shows the results when we use visual filters to take out Stephen's sales:



EmployeeKey	Full Name	Manager	Sales	Status
290	Amy Alberts	Stephen Jiang	$732,078.4446	Current
289	David Campbell	Roger Hamilton	$3,729,945.3501	Current
284	Garrett Vargas	Roger Hamilton	$3,609,447.2163	Current
291	Jae Pak	Amy Alberts	$8,503,338.6472	Current
283	Jillian Carson	Roger Hamilton	$10,065,803.5429	Current
288	José Saraiva	Roger Hamilton	$5,926,418.3574	Current
282	Linda Mitchell	Roger Hamilton	$10,367,007.4286	Current
296	Lynn Tsoflias	Syed Abbas	$1,421,810.9252	Current
281	Michael Blythe	Roger Hamilton	$9,293,903.0055	Current
286	Pamela Ansman-Wolfe	Roger Hamilton	$3,325,102.5952	Current
295	Rachel Valdez	Amy Alberts	$1,790,640.2311	Current
292	Ranjit Varkey Chudukatil	Amy Alberts	$4,509,888.933	Current
287	Shu Ito	Roger Hamilton	$6,427,005.5556	Current
294	Syed Abbas	Stephen Jiang	$172,524.4515	Current
293	Tete Mensa-Annan	Roger Hamilton	$2,312,545.6905	Current
285	Tsvi Reiter	Roger Hamilton	$7,171,012.7514	Current
Total			**$79,358,473.1261**	

Figure 12.7 – Reseller sales for employees considering SCD

Dealing with SCDs is not always as simple as this. In some cases, we need to make some changes to our data model. As mentioned at the beginning, we are not diving deep into various scenarios in this chapter, so we will leave it to you to investigate more. In the next section, we look at OLS, which is under public preview at the time of writing this book.

Introduction to OLS

In *Chapter 11*, *Row-Level Security*, we learned how to restrict data access for end users using RLS. In this section, we look at object-level security in Power BI. With OLS, we can hide the model objects based on the usernames and their roles, such as hiding an entire table or hiding some table columns for specific users. Therefore, if users use the **Analyse in Excel** feature to connect to a dataset, they can see only the tables and columns they have permission to see.

> **Notes**
>
> At the time of writing this book, OLS has just been released for public preview. Therefore, it is subject to change.
>
> At the time of writing this book, we cannot create OLS within Power BI Desktop. Instead, we have to use **Tabular Editor**.
>
> We cannot control the data's visibility to developers within Power BI Desktop with OLS or RLS. This is something that must happen within the source system. Regardless of whether OLS or RLS is in place or not, developers have access to all data available in the source systems. It is essential to know that RLS and OLS are not permission configurations; they only restrict users' access to specific rows in RLS or a specific object in OLS.

Implementing OLS

Let's go ahead and see how OLS works. Suppose the business needs to hide some objects from a specific group of people, as follows:

- Hide the `Internet Sales` table from everyone who is a member of the **Internet Sales Denied** security group.
- Hide the `OrderQuantity` column of the `Internet Sales` table from members of the **OrderQty Denied** security group.

We will use the `Chapter 12, OLS.pbix` sample file provided with the book.

In the preceding scenario, we need to have two security groups available. We assume you have those two security groups to hand. The following steps show how to implement the scenario:

1. Open the desired Power BI report and click the **Manage Roles** button.
2. Click the **Create** button to create a new role.
3. Type in **Internet Sales Denied** for the name of the new role.
4. Click the **Create** button again, and then type in **OrderQty Denied** for the name of the new role.
5. Click **Save**.

   The preceding steps are highlighted in the following screenshot:

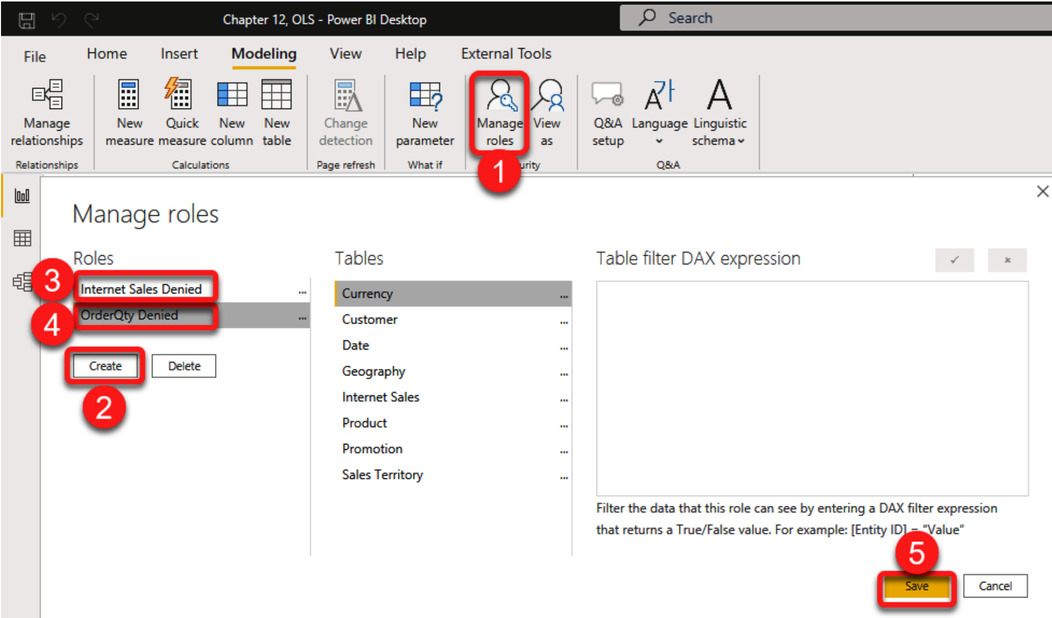


Figure 12.8 – Creating two new security roles

We now have to switch to **Tabular Editor** to implement the rest of the scenario. To do this, proceed as follows:

6. Click the **External Tools** tab on the ribbon.
7. Click **Tabular Editor**.

The process is illustrated in the following screenshot:

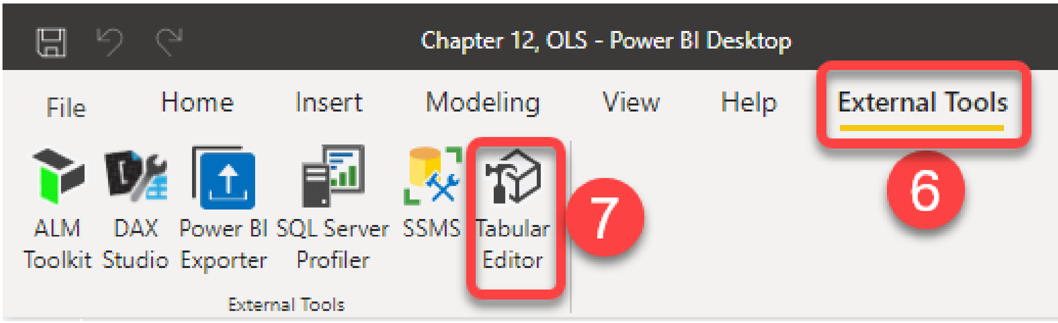


Figure 12.9 – Opening Tabular Editor from the External Tools tab

> **Note**
> Make sure you have the latest version of **Tabular Editor** installed on your machine.

8. In **Tabular Editor**, expand the **Tables** folder.
9. Click the `Internet Sales` table.
10. Expand **Object Level Security** from the **Properties** pane.
11. From the **Internet Sales Denied** drop-down menu, select **None**.

The preceding steps are illustrated in the following screenshot:

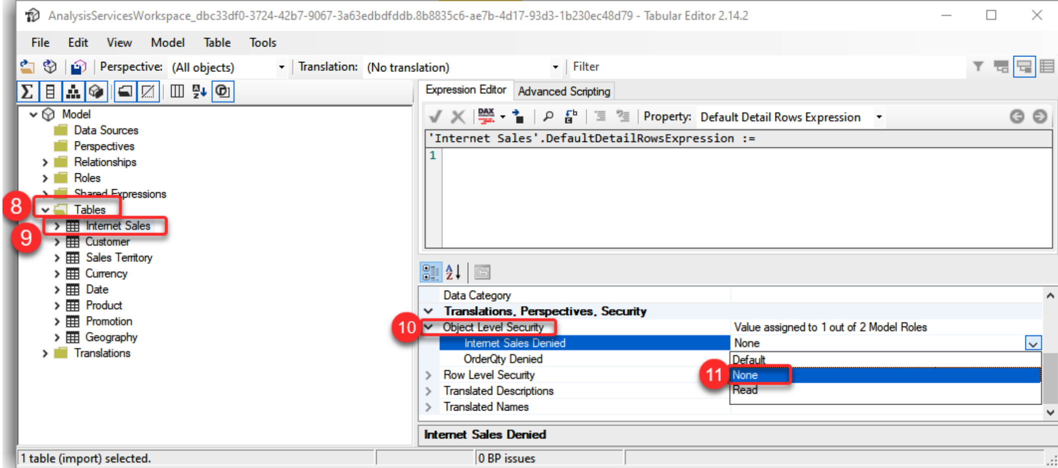


Figure 12.10 – Setting up OLS for tables in Tabular Editor

So far, we set OLS for the `Internet Sales` table, so whoever is a member of the **Internet Sales Denied** security group could not see the `Internet Sales` table, as if it didn't exist. In the next few steps, we implement the second part of the requirements within **Tabular Editor**, as follows:

12. Expand the `Internet Sales` table.
13. Click the `OrderQuantity` column.
14. Expand **Object Level Security** from the **Properties** pane.
15. From the **OrderQty Denied** drop-down menu, select **None**.
16. Save the changes to the model.

    The preceding steps are highlighted in the following screenshot:

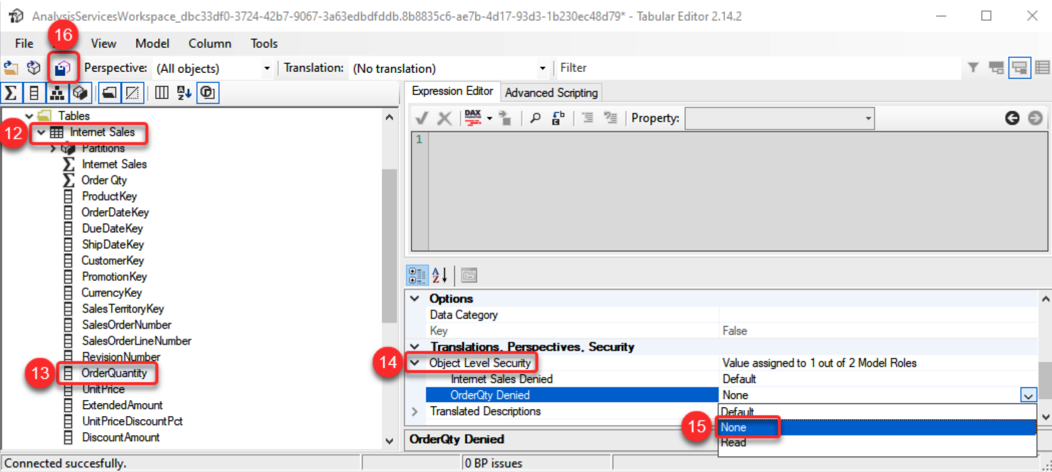


Figure 12.11 – Setting up OLS for columns

We have now implemented all OLS settings required by the business. In the next section, we test the roles.

Validating roles

Role validation for OLS is the same as what we did for RLS. The following steps show how to validate roles:

1. In Power BI Desktop, click the **View as** button from the **Modeling** tab.
2. Check the **Internet Sales Denied** role.
3. Click **OK**.

The preceding steps are highlighted in the following screenshot:

Figure 12.12 – Validating the Internet Sales Denied role

The result of the validation is shown in the following screenshot. Please note that the `Internet Sales` table has disappeared from the **Fields** pane, and also, all visuals linked to the `Internet Sales` table are broken:

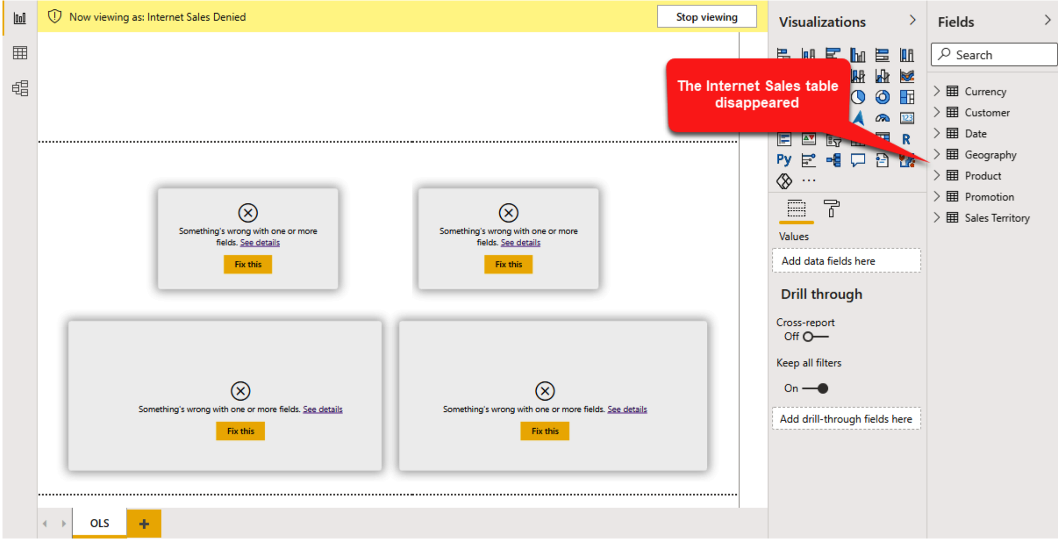


Figure 12.13 – Viewing the report as Internet Sales Denied role

We can validate the **OrderQty Denied** role in the same way. The following screenshot shows the result of the validation:

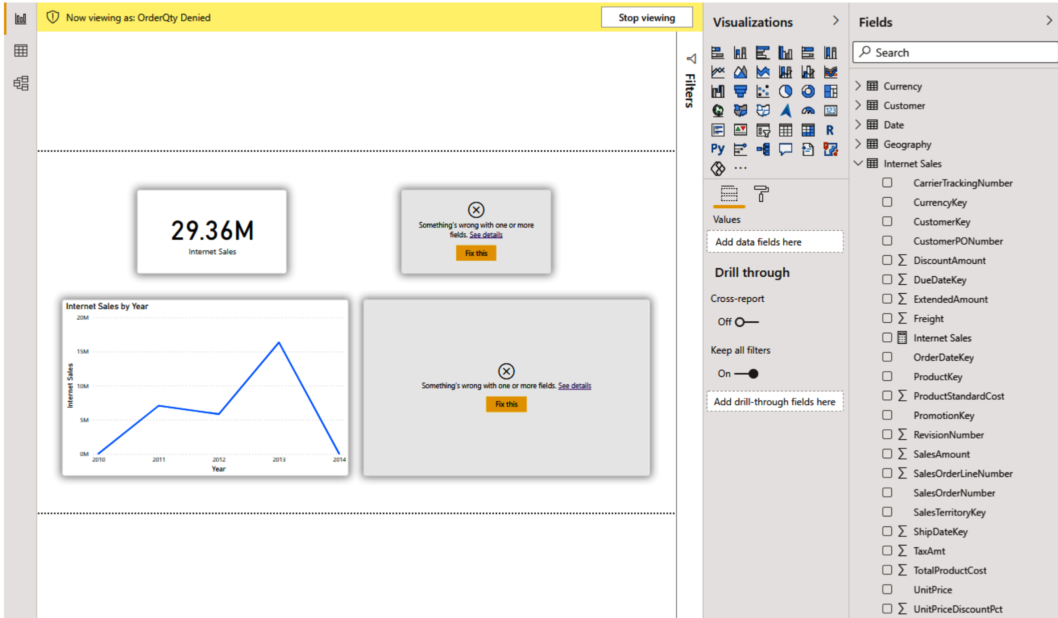


Figure 12.14 – Validating the OrderQty Denied role

As the preceding screenshot shows, only the visuals linked to the `OrderQuantity` column from the `Internet Sales` table are broken. Here are other things that have happened:

- The `OrderQuantity` column has disappeared from the `Internet Sales` table.
- The **Order Qty** measure that was linked to the `OrderQuantity` column has also disappeared.

Assigning members to roles in the Power BI service

After we have implemented OLS, we need to publish a report to the Power BI service, and then we have to assign users or groups to roles within the service. The following steps show how to assign members to a role in the Power BI service after publishing a report to a workspace:

1. Click the ellipsis button next to the desired dataset.
2. Click **Security**.
3. Select a role.

4. Type the name of the user of a group (in this case, it is a security group).
5. Click the **Add** button.

   The preceding steps are highlighted in the following screenshot:

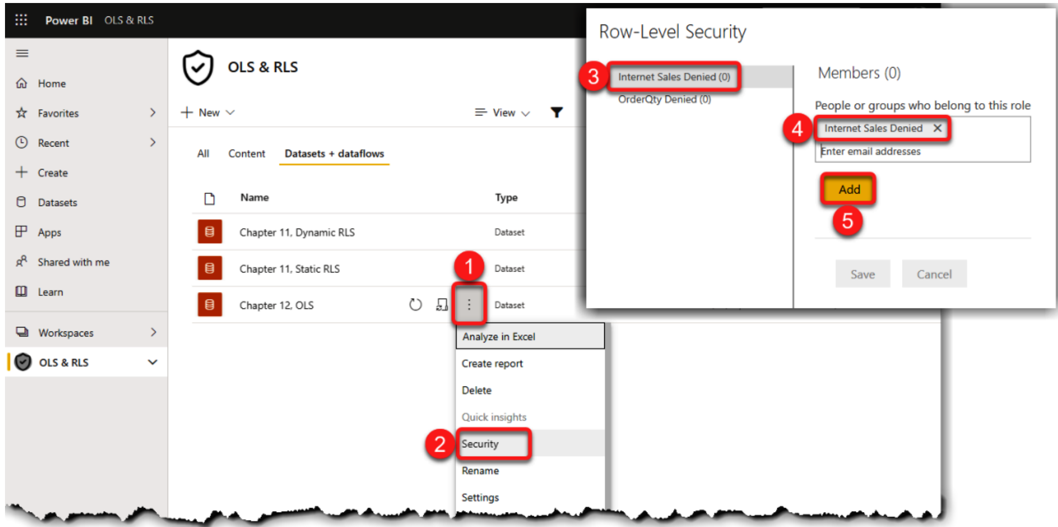


Figure 12.15 – Assigning members to roles in the Power BI service

> **Note**
> I already created two security groups with the same names as the roles (**Internet Sales Denied** and **OrderQty Denied**).

We will repeat the preceding steps to assign the **OrderQty Denied** security group to the **OrderQty Denied** role. In the next section, we look at validating security roles within the Power BI service to ensure everything works as expected.

Validating roles in the Power BI service

In this section, we validate the roles in the service by following these steps after navigating to the desired workspace:

1. Click the ellipsis button next to a dataset.
2. Click **Security**.
3. Click the ellipsis button of a role you want to validate.
4. Click **Test as role**.

The preceding steps are highlighted in the following screenshot:

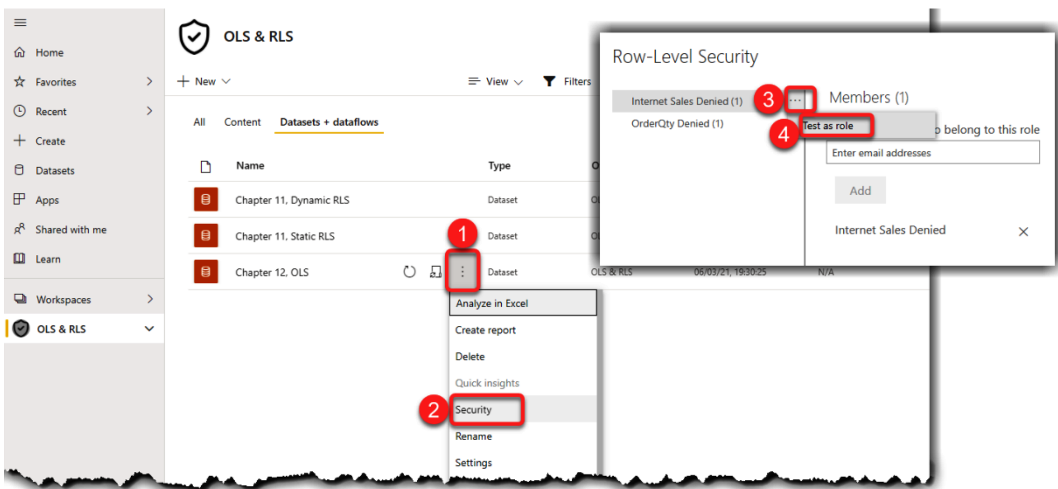


Figure 12.16 – Validating security roles in the service

The following screenshot shows the result of the validation:

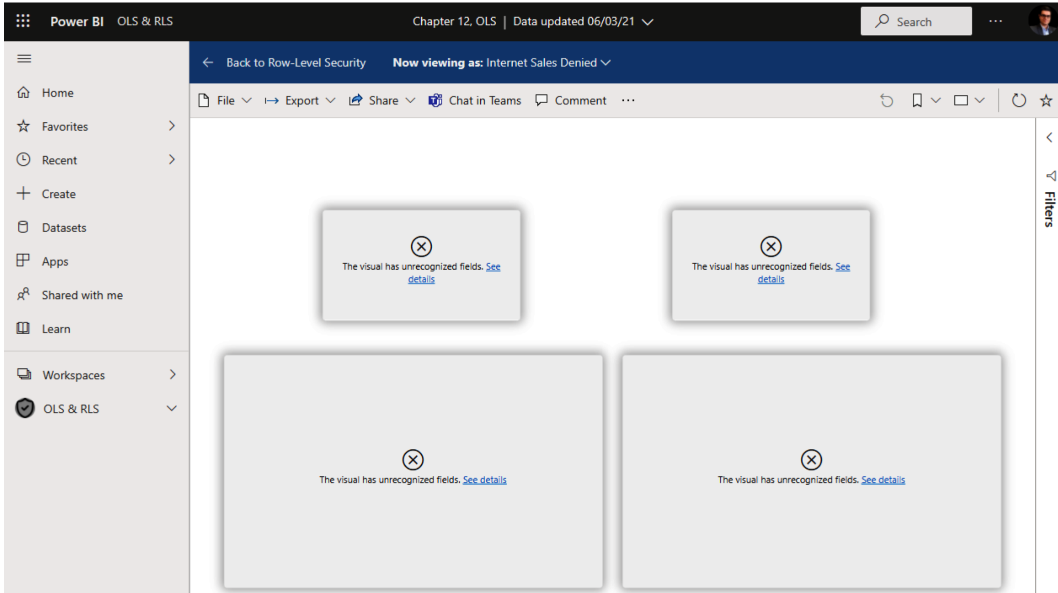


Figure 12.17 – The validation result in the Power BI service

In this section, we learned how to set up OLS in Power BI. In the next section, we look at some applications of **dataflows**.

Introduction to dataflows

Microsoft announced the public preview availability of dataflows in November 2018. Later on, in April 2019, dataflows became generally available. Back then, dataflows were a **Premium** feature. However, nowadays, we can leverage them in various Power Platform technologies as well as a Power BI **Pro** licensing plan. With that brief history, let's discuss what a dataflow is. In short, dataflows are a cloud version of Power Query, which is also known as Power Query Online. From a Power BI perspective, we can leverage Power Query's power to prepare and shape data within the Power BI service using dataflows.

> **Note**
>
> Dataflows are not only a component of Power BI. Other tools and services can also access data held by dataflows.

The prepared data is stored in the shape of files and folders, known as the **Common Data Model** (**CDM**) folder, in **Azure Data Lake Storage Gen2** (**ADLS Gen 2**) and managed in the Power BI service. If the organization owns an ADLS Gen 2 already, we can expose the prepared data to various tools and services. Moreover, with the use of dataflows, we can make the prepared data available for the organization's users. Dataflows are a self-service ETL tool available in a Power BI platform that makes data preparation processes less dependent on **Information Technology** (**IT**) departments. Dataflows enable users to use a set of consumption-ready data. As a result, in the long run, dataflows can decrease development costs by increasing reusability.

The main difference between data preparation in Power Query within Power BI Desktop and dataflows is that when we use Power BI Desktop to prepare the data, the results are only available within the data model. After we publish the Power BI service model, the entities are available within the published dataset. Some may think that we still can access the dataset's tables and data from other datasets in a **composite model** scenario. That is possible, but it is not the right design model to create a composite model for the sake of getting the prepared data into our data model. We will discuss composite models later in this chapter.

Scenarios for using dataflows

Having a centralized cloud-based self-service data preparation mechanism sounds so compelling that you might think of leaving Power Query in Power BI Desktop behind and move all data preparation activities to dataflows. Well, that doesn't sound a practical idea. The following are some scenarios where using dataflows makes more sense. We use dataflows in the following situations:

- We do not have a data warehouse in our organization. All our Power BI reports are getting data directly from the source systems, which affects the source systems' performance. Therefore, we need to create an organizational data warehouse. In this scenario, we follow the regular data warehouse architecture by separating activities into **staging** the data, **transforming** the data and building a **star schema**.
- We want to share the data preparation logic across the organization. As a result, many other datasets and reports inside Power BI—or, in general, Power Platform—can reuse the prepared data.
- The organization owns ADLS Gen 2, and we want to connect other Azure services to the prepared data.
- We want to create a **single source of truth** (**SSOT**) for our business analysts. Therefore, we can create dataflows that analysts can connect to rather than connecting to underlying transactional systems or dealing with disparate files.
- We need to prepare data from large data sources, and we own a Power BI **Premium** capacity; dataflows provide more flexibility and work more efficiently.
- We want to use the prepared data across various technologies in Power Platform. When we create the dataflows, we can make them available for other Power Platform products such as Power Apps, Power Automate, Power Virtual Agent, and Dynamics 365.
- We need a self-service data preparation tool that does not require a lot of IT or development background. Indeed, the dataflows' creators only need to have knowledge of Power Query.

Dataflow terminologies

As mentioned earlier, dataflows are also known as **Power Query Online**; therefore, they inherit many Power Query terminologies. The following terminologies are ones that are either applicable in dataflows or are another version of a term that is already available in Power Query:

- **Fields**: Fields are just like columns in Power Query.
- **Entities**: An entity consists of a set of fields or columns, similar to in Power Query. Some resources refer to an entity as a table, but there is a difference between an entity in dataflows and a query in Power Query in Power BI Desktop. In Power Query, in Power BI Desktop, all entities are called queries. But in dataflows, there are different types of entities, as follows:
    - **Normal entity**: Just like a table query in Power Query in Power BI Desktop. The icon currently used for normal entities is .
    - **Linked entity**: We have a linked entity when we reference an existing entity defined in another dataflow. When we create a linked entity, the data will not load into the new dataflow; only a link to the source dataflow exists in the new one. For that reason, the linked entities are read-only. As a result, we cannot create any further transformation steps. The icon currently used for linked entities is .
    - **Computed entity**: Computed entities are entities referencing other entities by taking more transformation steps. In this type of entity, data is processed for the source entity within the source dataflow. The data then flows through the transformation steps in the new entity. The transformed data then gets stored for the new dataflow. The icon currently used for computed entities is .

> **Note**
>
> Both linked entities and computed entities are only available in Power BI **Premium**, but the source entities for either linked and computed entities can be on a regular Power BI **Pro** workspace.
>
> The workspace keeping the linked entities must be a modern workspace (not the classic workspaces linked to Office 365 groups).
>
> We can link to entities in multiple dataflows residing in multiple modern workspaces.

Creating dataflows

To create a dataflow, we have to log in to our Power BI service in the browser of our choosing. The following steps show how to start creating a new dataflow:

1. Select any desired workspace.
2. Click the **New** button.
3. Click **Dataflow**.

   The preceding steps are highlighted in the following screenshot:

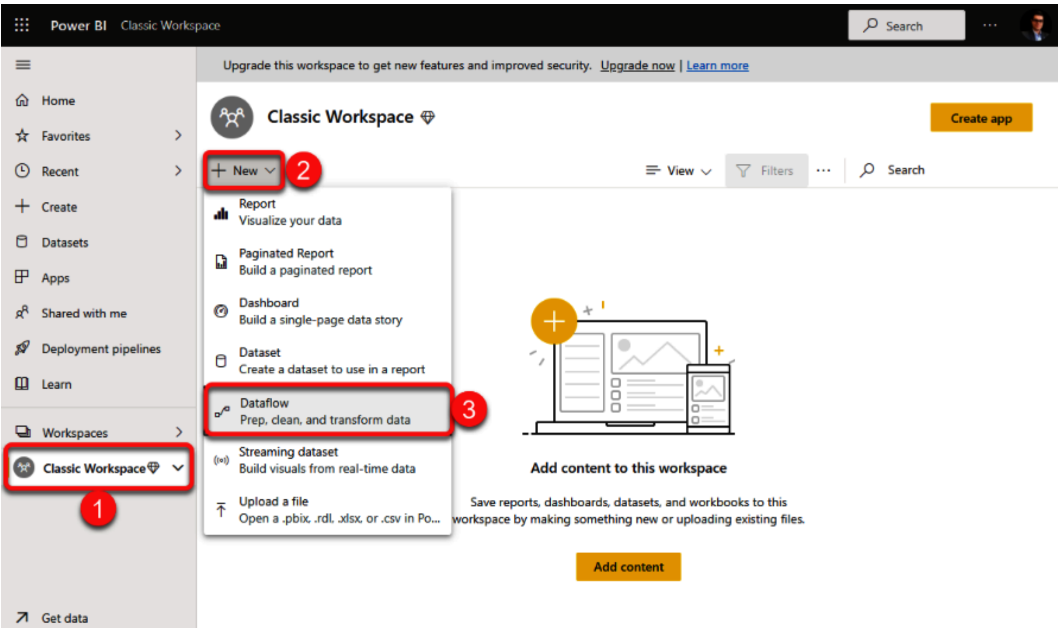


Figure 12.18 – Creating dataflows in the Power BI service

Now, depending on the type of workspace hosting the dataflow, we may have one of the following options:

- Option **A**: A regular classic workspace, as illustrated in the following screenshot:

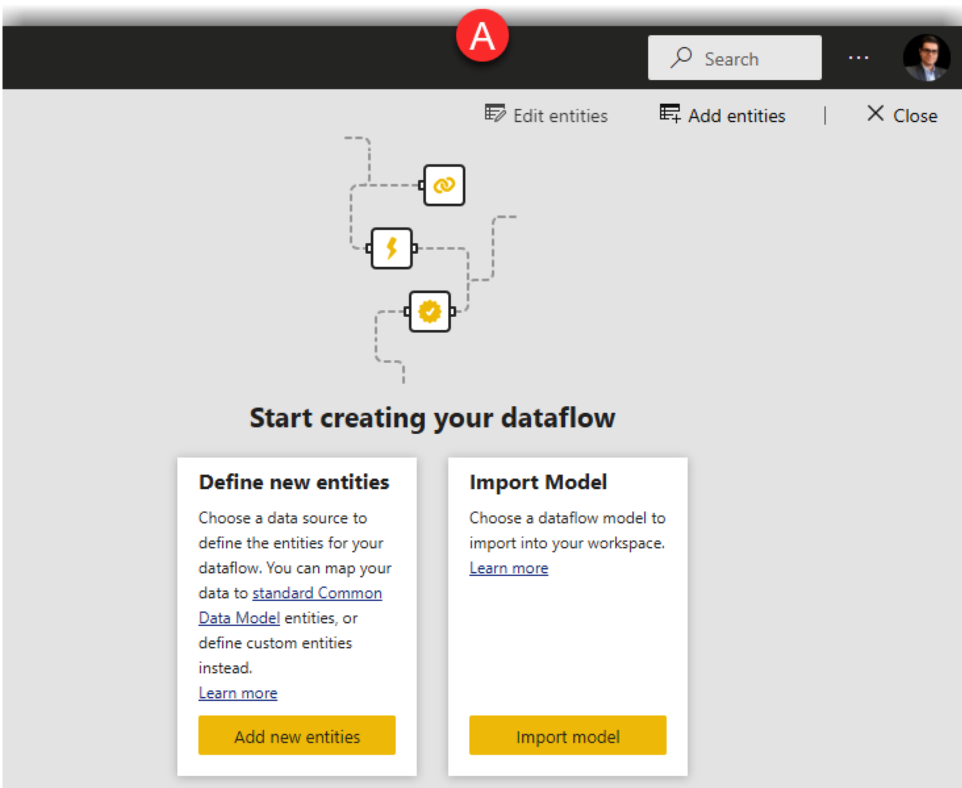


Figure 12.19 – Creating dataflows in a regular classic workspace

- Option **B**: A classic workspace backed by a Premium capacity, as illustrated in the following screenshot:

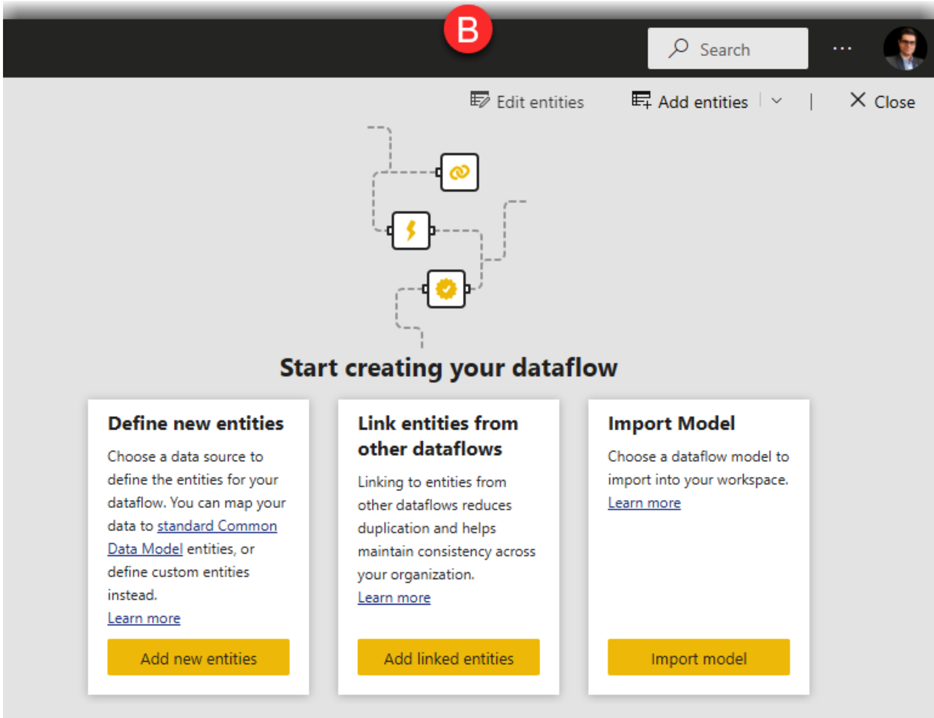


Figure 12.20 – Creating dataflows in a classic workspace backed by a Premium capacity

- Option **C**: A modern workspace, as illustrated in the following screenshot:

Figure 12.21 – Creating dataflows in a modern workspace

As you see in *Figure 12.21*, all options to create a new dataflow are available in a modern workspace. However, if the workspace is not a Premium workspace, we can still create a LinkedIn entity but we cannot refresh it. The following sections show how to create new entities, linked entities, and importing models. We avoid discussing the **Attach a Common Data Model folder** because it is in preview and has not been released yet at the time of writing this book.

Creating new entities

So far, we navigated to the desired workspace and started creating the entities in our dataflow. To create a new entity from the options available (as shown in *Figure 12.21*) we follow these next steps:

1. Click **Define new entities**, as illustrated in the following screenshot:

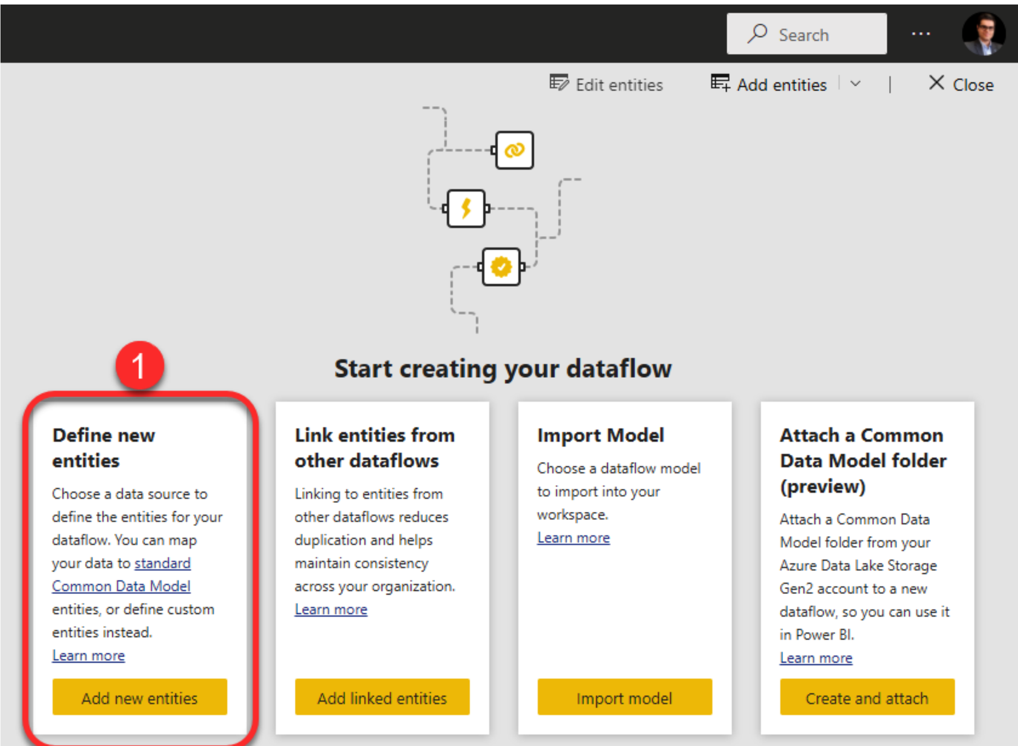


Figure 12.22 – Defining new entities within a dataflow

2. Select any desired data source connector, as illustrated in the following screenshot:

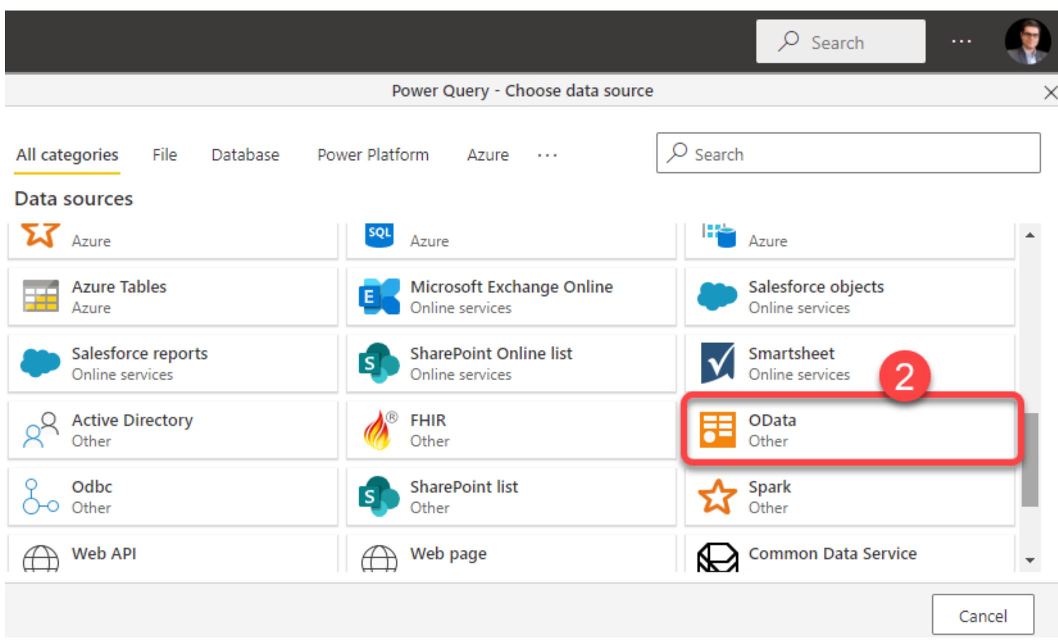


Figure 12.23 – Selecting a data source connector

3. Fill in the **Connection settings** fields.
4. Click **Next**.

The preceding steps are highlighted in the following screenshot:

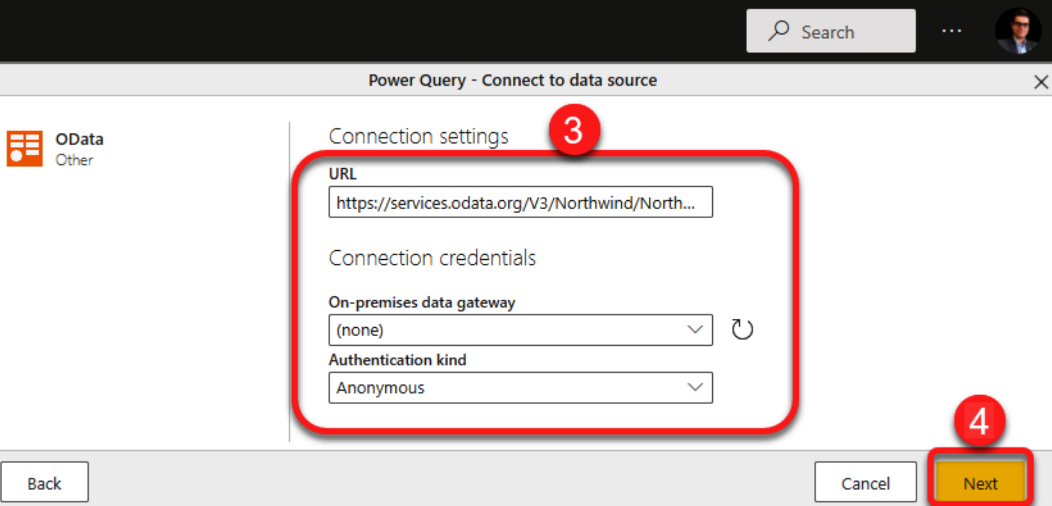


Figure 12.24 – Filling in the Connection settings fields

5. Select entities from the **Power Query - Choose data** form. (You may see a different form, depending on the data source you selected in the previous steps. We used a `Northwind OData` sample.)
6. Click **Transform data**.

   The preceding steps are highlighted in the following screenshot:

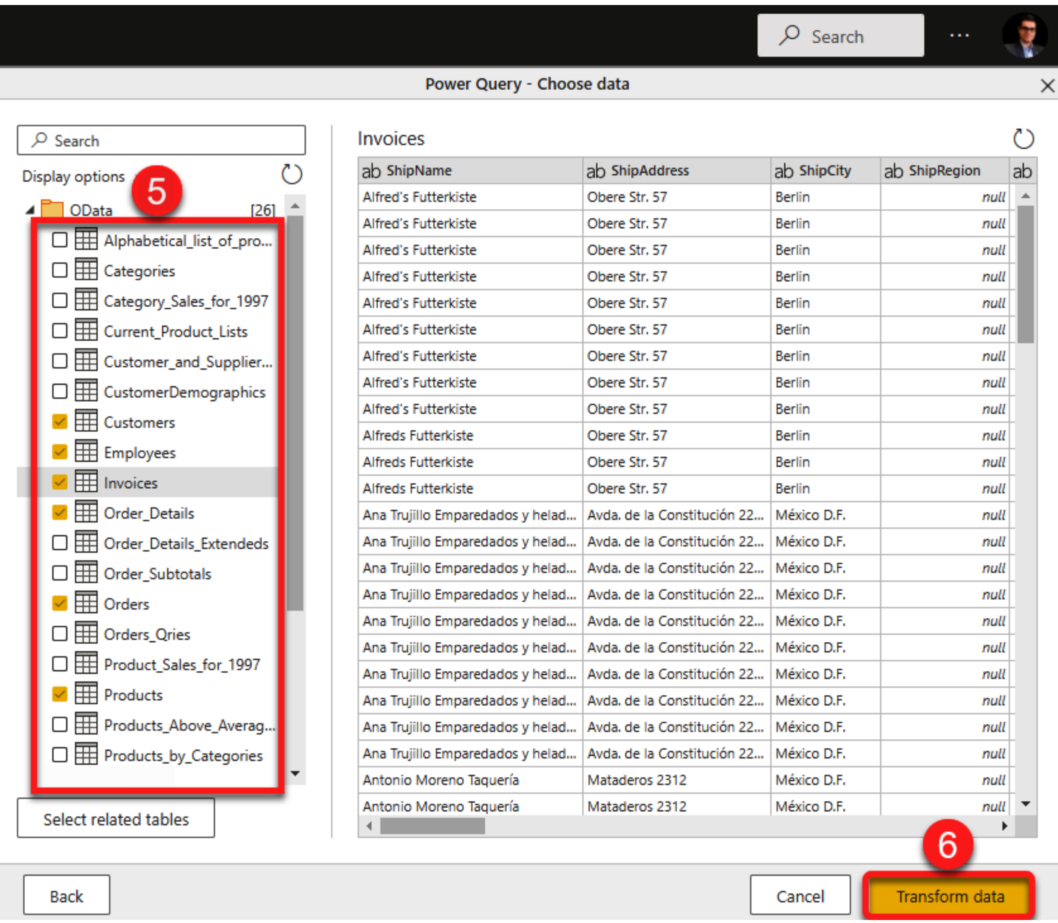


Figure 12.25 – Power Query - Choose data form when creating a new dataflow

We have now navigated to the **Power Query - Edit queries** form. This form is similar **Power Query Editor** window in Power BI Desktop. One point to notice in the **Power Query - Edit queries** form is the **warnings** section. As the following screenshot shows, if there are any issues in loading the data from some columns, a warning message shows up in the **Warnings** section. We can click on the **warnings** section to see more details:

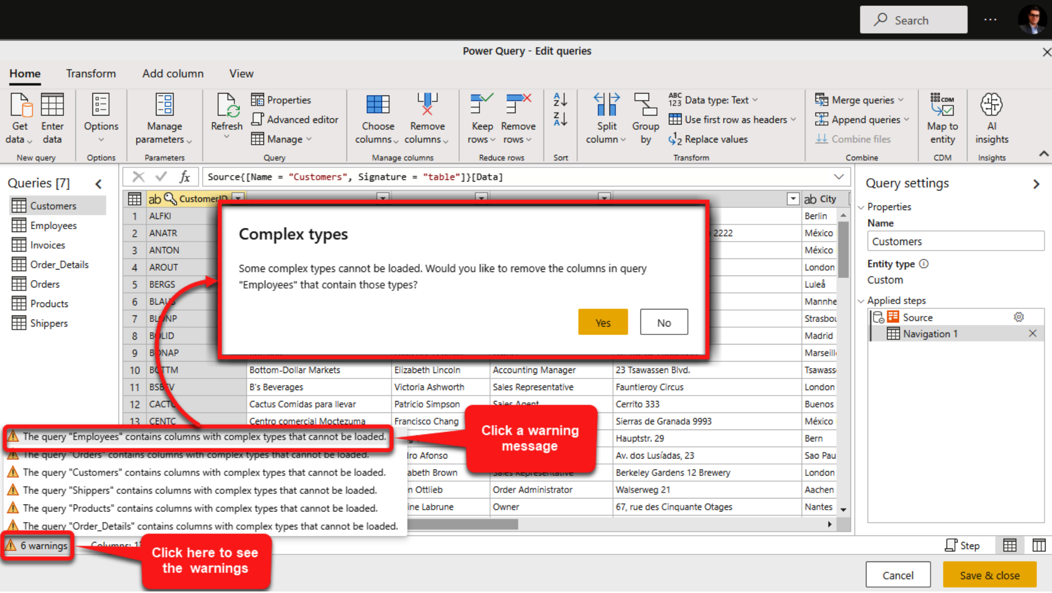


Figure 12.26 – Warning message after getting data from the source

As shown in *Figure 12.26*, we can click the warning message to take action if necessary. After we have finished all the required transformation steps, we can save the dataflow by clicking the **Save & close** button.

The next step is to give our dataflow a name and save it, as shown in the following screenshot:

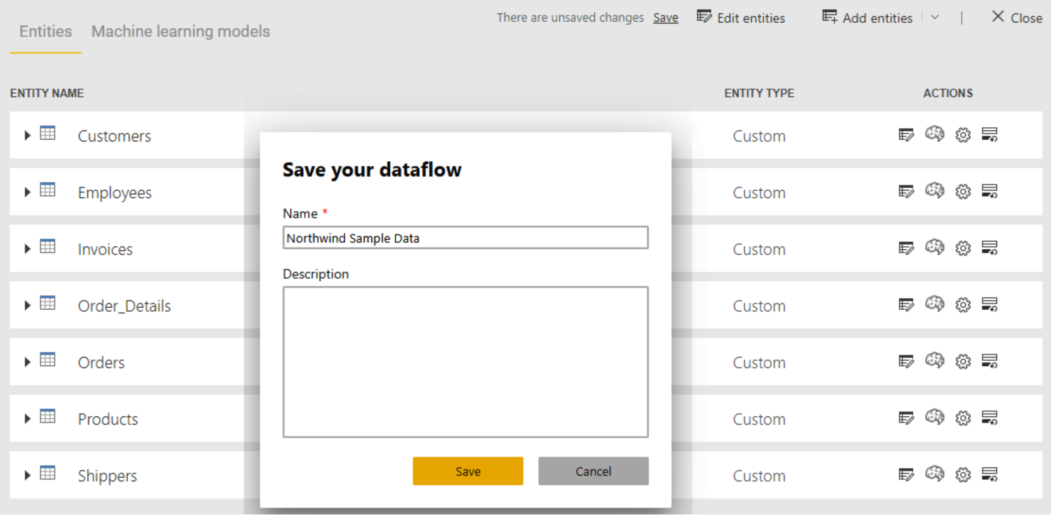


Figure 12.27 – Saving the dataflow

Unlike **Power Query Editor** in Power BI Desktop, which loads the data from the source immediately after we click the **Close & Apply** button, dataflows will not automatically load data from the source. Instead, we have an option to refresh the data or schedule a data refresh, as shown in the following screenshot:

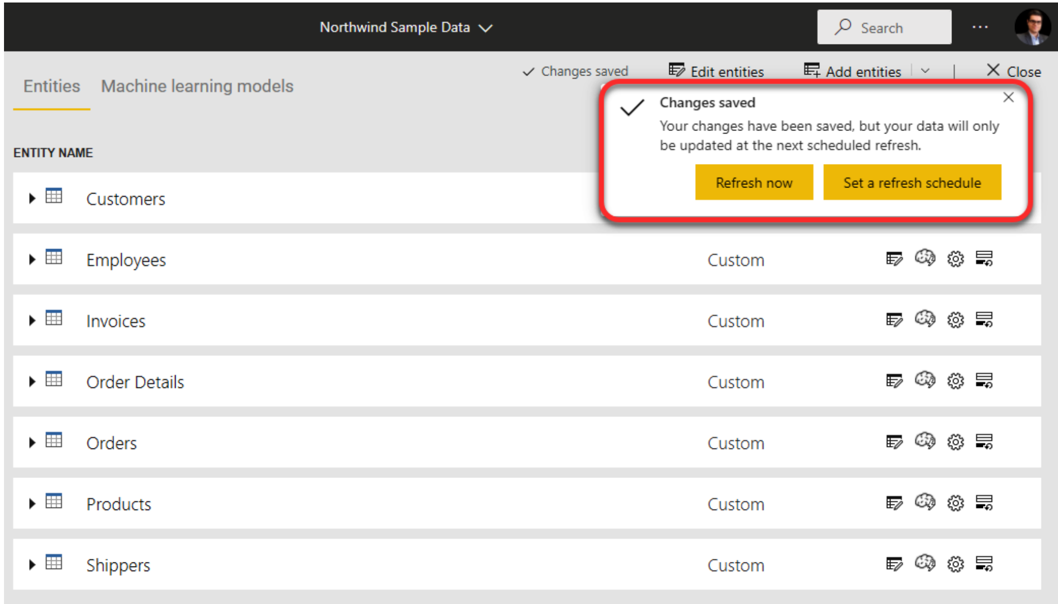


Figure 12.28 – Data refresh options after creating a new dataflow

We can populate the tables by clicking the **Refresh now** button or by setting a schedule to refresh the tables automatically.

Creating linked entities from other dataflows

We can create linked entities from other dataflows to create a new dataflow, or add a linked entity to an existing dataflow. The following steps show how to add linked entities as a new dataflow:

1. After navigating to the desired workspace and starting to create entities in our dataflow, click the **Add linked entities** button, as illustrated in the following screenshot:

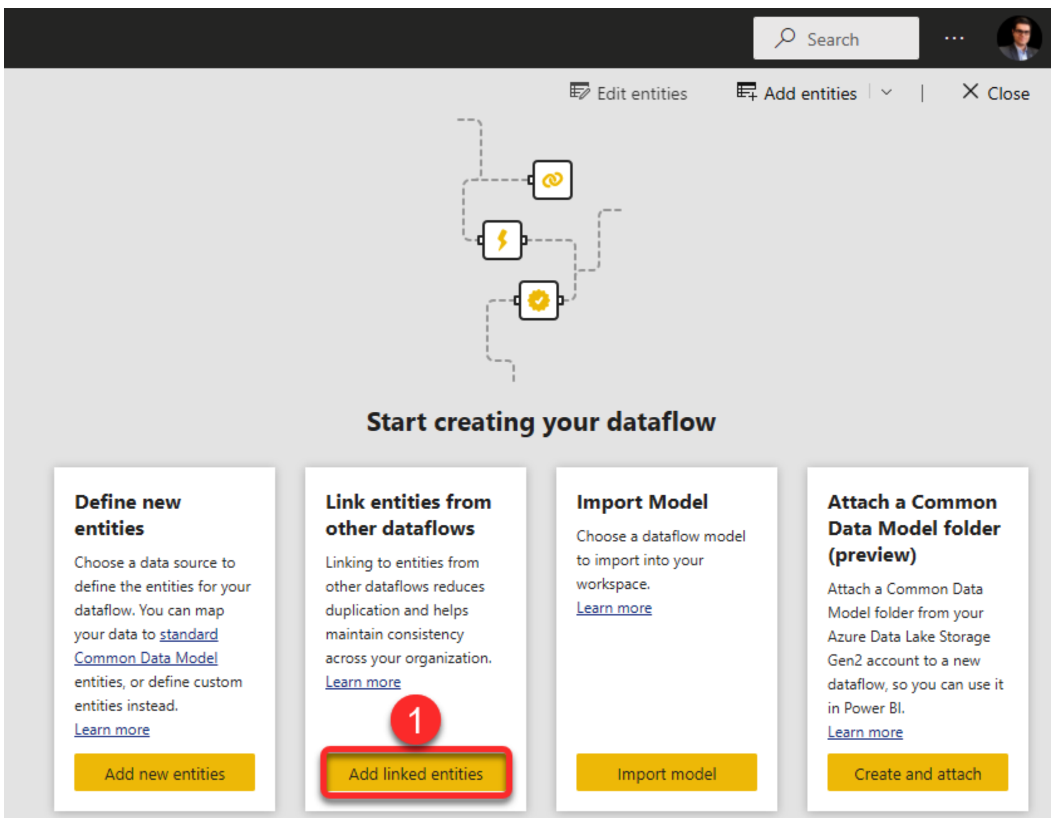


Figure 12.29 – Adding linked entities as a new dataflow

2. Go through the **Connection settings**.

> **Note**
>
> If the source data is an on-premises data source, you need to install an **on-premises data gateway**. On-premises data gateways are out of the scope of this book, but I encourage you to check out the following links:
>
> The Microsoft documentation: `https://docs.microsoft.com/en-us/power-bi/connect-data/service-gateway-onprem?WT.mc_id=5003466`
>
> *BI Insight*, *Implementing On-premises Data Gateway* (*Enterprise Mode*): `https://wp.me/p3L3Ff-1z0`

3. Click **Next**.

   The preceding steps are highlighted in the following screenshot:

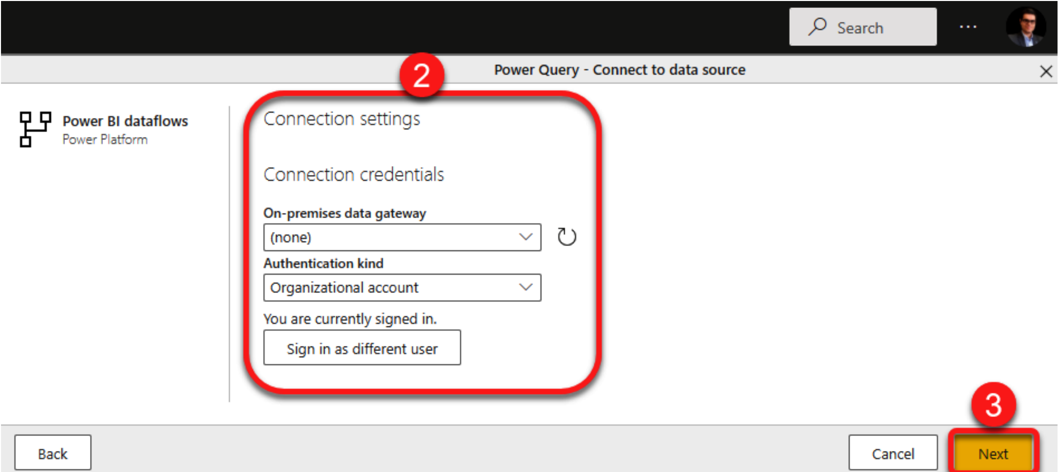


Figure 12.30 – Connection settings to create linked entities

4. In the **Power Query - Choose data** form, expand a workspace.
5. Select any desired dataflows.
6. Click the desired tables.
7. Click the **Transform data** button.

The preceding steps are highlighted in the following screenshot:

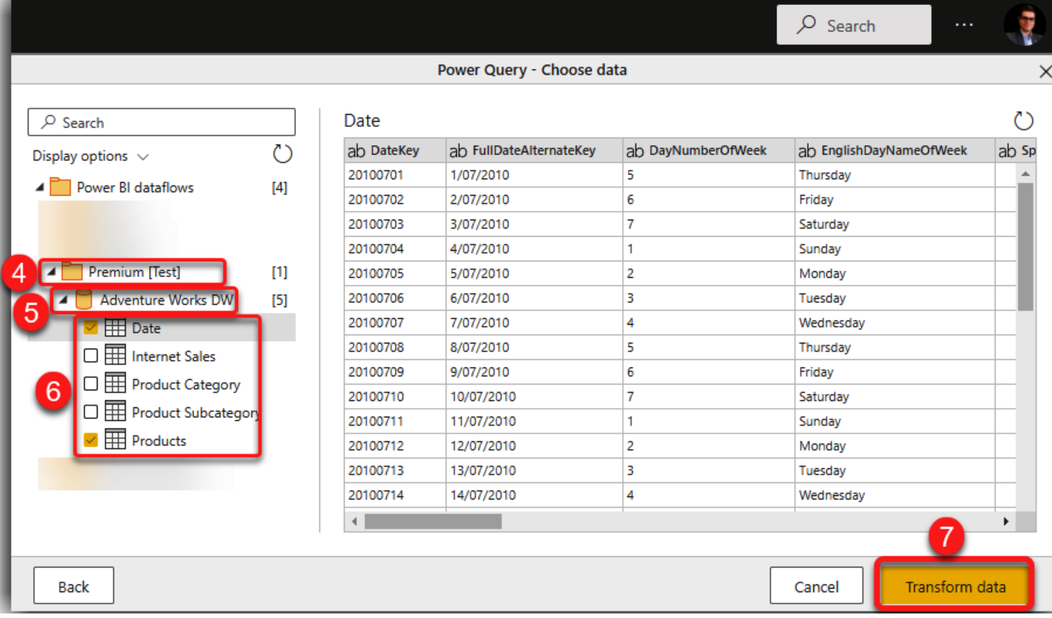


Figure 12.31 – Selecting tables from another dataflow to link

8. Click **Save & close**, as illustrated in the following screenshot:

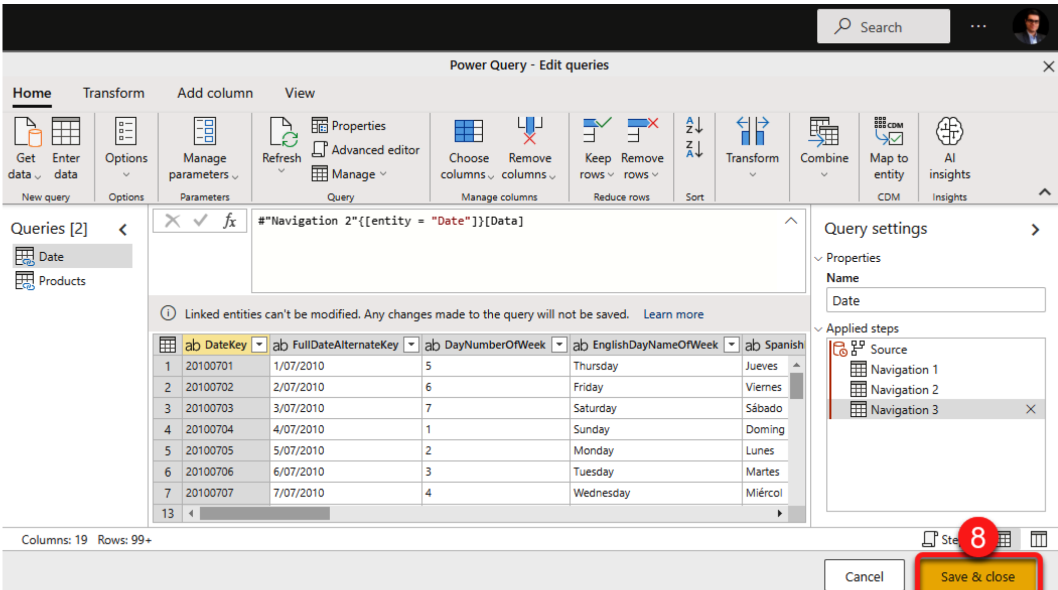


Figure 12.32 – Save & close changes for selected linked entities

9. Type in a name and then click the **Save** button, as illustrated in the following screenshot:

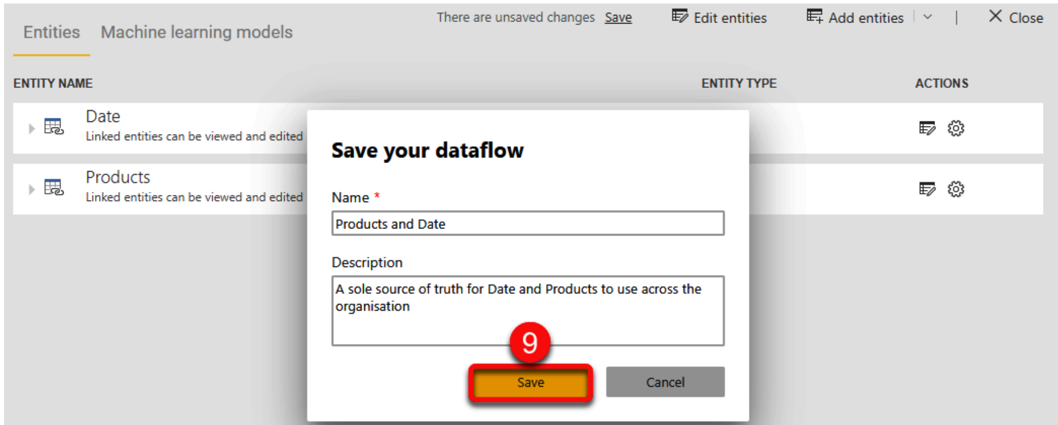


Figure 12.33 – Saving linked entities as a new dataflow

So far, we have learned how to create new dataflow entities or linked entities. In the next section, we learn how to create computed entities.

Creating computed entities

In this section, we learn how to create computed entities. The most common way to create a computed entity is within an existing dataflow by referencing another entity. The following steps show how to create a computed entity after navigating to the desired workspace and opening a dataflow:

1. Click the **Edit entities** button, as illustrated in the following screenshot:

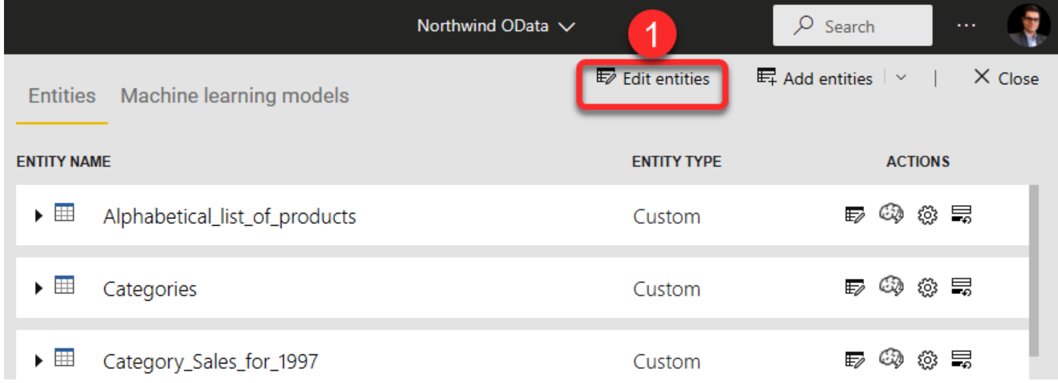


Figure 12.34 – Editing entities in a dataflow

2. Right-click an entity from the **Queries** pane.
3. Click **Reference**.

The preceding steps are highlighted in the following screenshot:

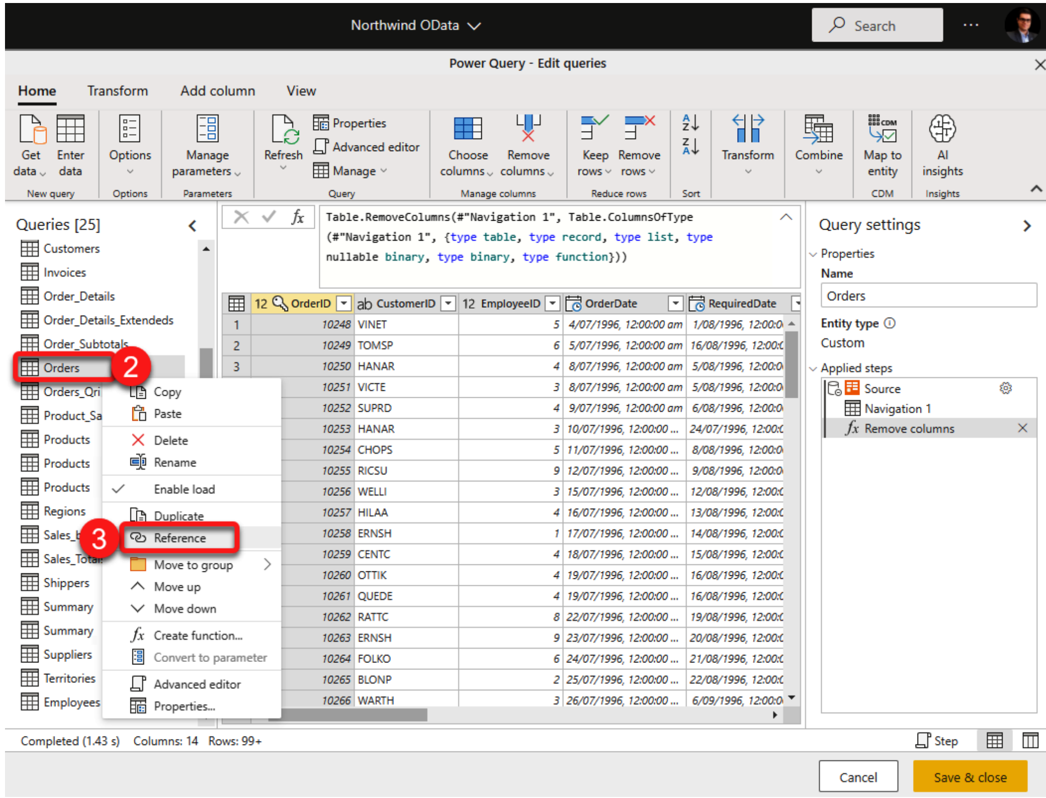


Figure 12.35 – Creating a computed entity

Our computed entity is created, as shown in the following screenshot:

Figure 12.36 – Computed entity

We can also create a computed entity from a linked entity by referencing it or adding some transformation steps. Remember: linked entities are read-only; therefore, if we modify the entity, the entity is no more a linked entity—it is now a computed entity.

Exporting and importing dataflows

We can export and import dataflows. In this section, we learn how to export a dataflow and how to import exported dataflows.

Exporting dataflows

Navigate to the desired workspace, then proceed as follows:

1. Hover over a dataflow and click the ellipsis button.
2. From the menu, click `Export.json`.
3. A message shows up when the file is ready.
4. Depending on your browser, the **JavaScript Object Notation** (**JSON**) file downloads automatically to your local machine.

The preceding steps are highlighted in the following screenshot:

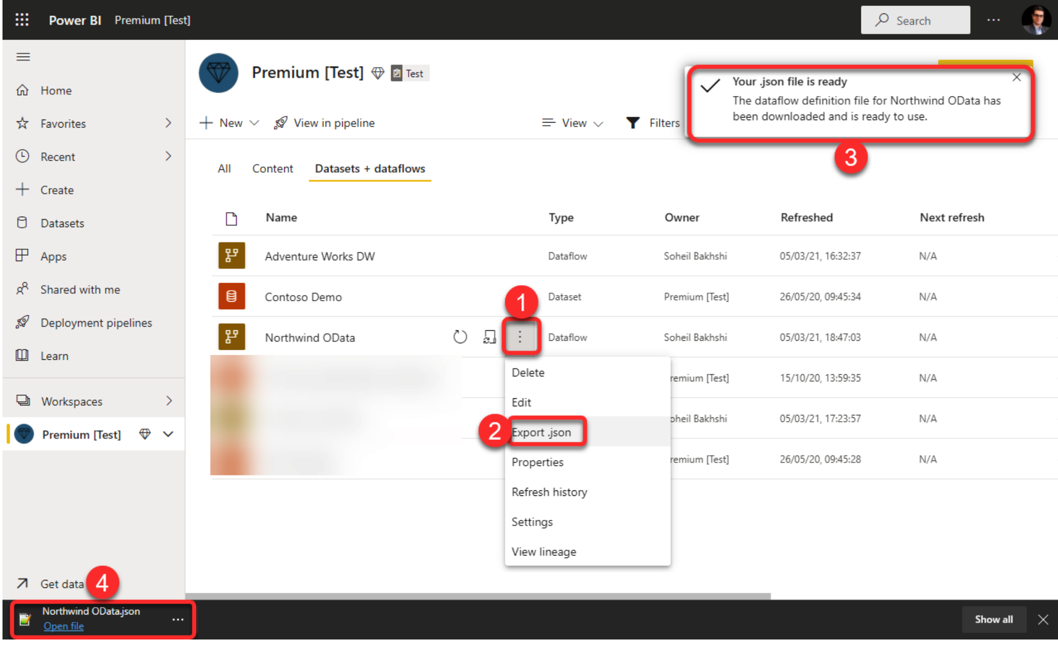


Figure 12.37 – Exporting dataflow definitions in JSON format

Importing dataflows

Importing a dataflow is simple. The following steps show how to import a dataflow after navigating to the desired workspace:

1. Click the **New** button.
2. Click **Dataflow**, as illustrated in the following screenshot:

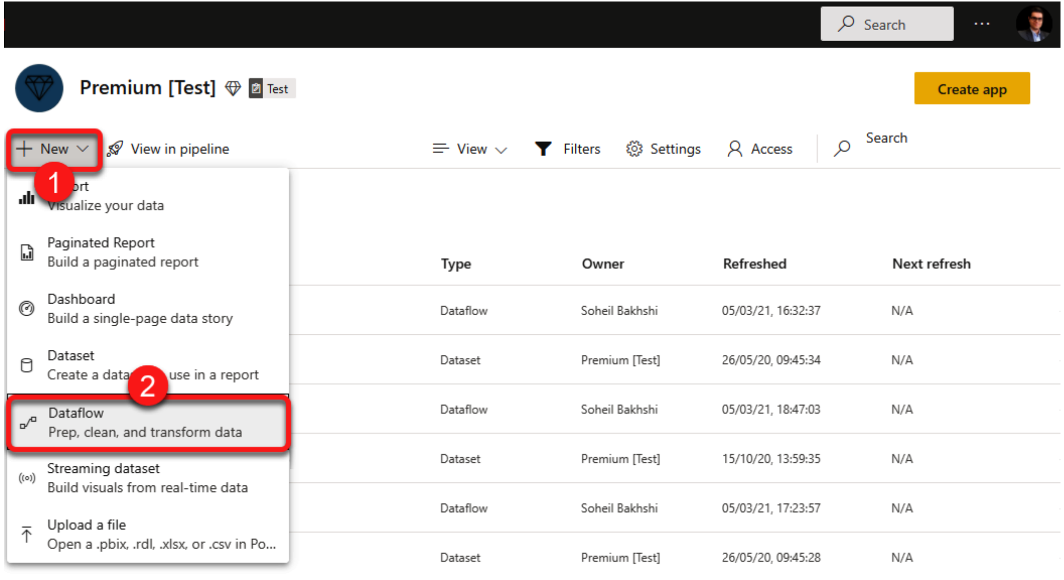


Figure 12.38 – Creating a new dataflow

3. Click the **Import model** option.
4. Select an exported dataflow definition file (**JSON**).
5. Click **Open**.

The preceding steps are highlighted in the following screenshot:

Figure 12.39 – Importing a dataflow

When we go back to the workspace, we see the imported dataflow. Just one more thing to note – we only import the dataflow definition; therefore, the data is not imported. As a result, we need to refresh the data later.

Introduction to composite models

In *Chapter 4*, *Getting Data from Various Sources*, we discussed different storage modes for a dataset. Let's quickly recall the storage modes for datasets, as follows:

- **Import**: For when we keep the data model in Power BI, and the whole data is cached in the memory. In this mode, all tables are in **Import** storage mode.
- **DirectQuery**: For when we create a data model in Power BI, but the data is NOT cached in the memory. In this mode, all tables are in **DirectQuery** storage mode.

- **Connect Live**: A specific type of **DirectQuery**, connecting to a semantic model, not a relational database (data store). When we use **Connect Live**, the data model is hosted somewhere else; therefore, we cannot make any changes in the data model. Instead, we can only get data ready from the data model.
- **Composite** (**Mixed**): For when a portion of data is cached in the memory, while the rest is not. In this mode, some tables are in **Import** storage mode, and some tables are in **DirectQuery** storage mode or **Dual** storage mode.

The latter is the main topic for this section. Previously, composite models only supported relational databases for **DirectQuery**. This means that **SQL Server Analysis Services** (**SSAS**), SSAS tabular models, **Azure Analysis Services** (**AAS**), and Power BI Datasets were automatically out of the game. With the December 2020 release of Power BI Desktop, Microsoft introduced a new generation of composite models. This new generation not only supports **DirectQuery** connections over relational databases such as SQL Server databases, but it also supports **DirectQuery** in AAS instances and Power BI datasets.

> **Note**
>
> At the time of writing this book, on-premises instances of SSAS tabular models are not yet supported.

This is a massive change in how we interact with the data, especially from an analytical perspective. With composite models, we can now connect to the individual semantic layers from a single Power BI data model. We can also import data from other data sources such as SQL Server or Excel and create an enterprise-grade self-service semantic layer using Power BI.

New terminologies

The new generation of composite models comes with new terminologies. Understanding these terms will help us in resolving more complex scenarios more efficiently, with fewer issues as a result. In the following few sections, we learn about those new terms.

Chaining

Chaining is a new terminology introduced with the new composite model. When a Power BI report or dataset is based on some other semantic model hosted in AAS or Power BI datasets, we create a chain; in other words, chaining is about the dependencies between semantic layers used in composite models. So, when we create a dataset on top of other datasets (or AAS models), the new dataset is dependent on a series of other datasets.

Chain length

When we create a chain, the chain's length refers to the number of semantic layers the current dataset is dependent on. Let's implement a scenario to better understand the terminology.

The business has a semantic model hosted at an AAS dataset that is still under development. The AAS developers have a massive backlog and future tasks to implement. The business has an urgent requirement for reporting. The business needs to define banding to the `Unit Price` column on the `Internet Sales` table, as follows:

- **Low**: When the unit price is smaller than **US Dollars** (**USD**) $100
- **Medium**: When the unit price is between $101 and $1,000
- **High**: When the unit price is between $1,001 and $3,000
- **Very high**: When the unit price is greater than $3,001

Let's look at the scenario in more detail. First of all, we have a semantic model in AAS, but the developers are too busy with their day-to-day tasks to respond to an urgent request from the business. The business can have many urgent requests every day, but our AAS developers cannot stop developing in order to answer the business requirements. As a Power BI data modeler, we can help to solve this issue very quickly.

The following steps show how to meet the preceding requirement in Power BI Desktop:

1. Select **Connect Live** when connecting to the AAS instance from Power BI Desktop, as illustrated in the following screenshot:

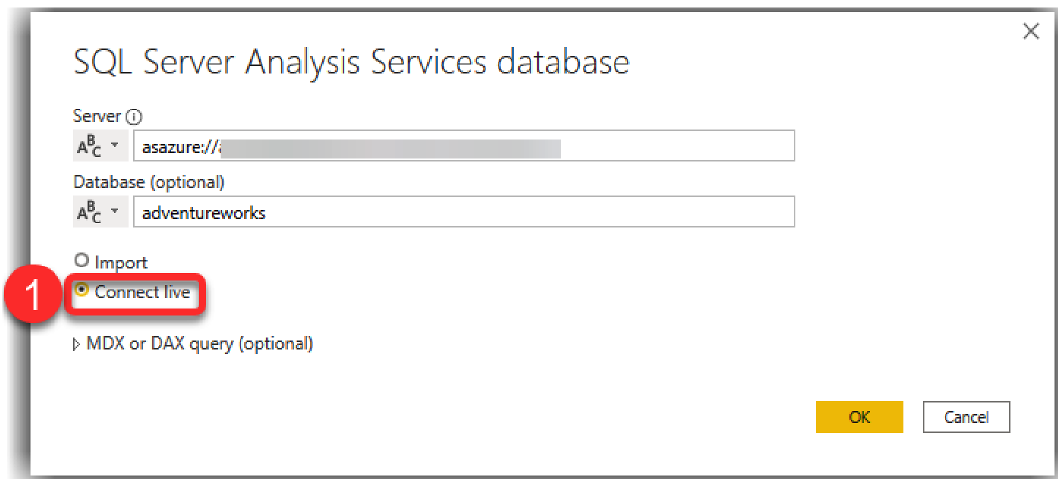


Figure 12.40 – Connecting live to AAS

2. Click the **Model** tab to see the current AAS data model, as illustrated in the following screenshot:

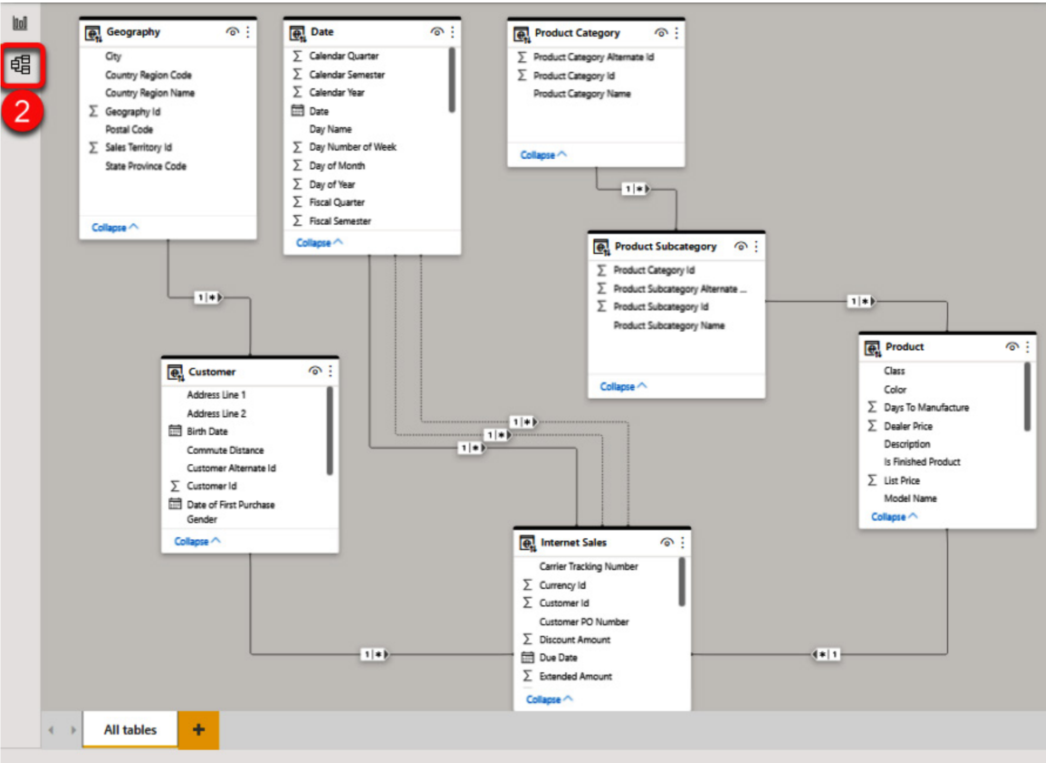


Figure 12.41 – Current data model in AAS (connection mode: Connect Live)

3. Click the **Transform data** button from the **Home** tab, as illustrated in the following screenshot. This changes the connection mode from **Connect Live** to **DirectQuery**:

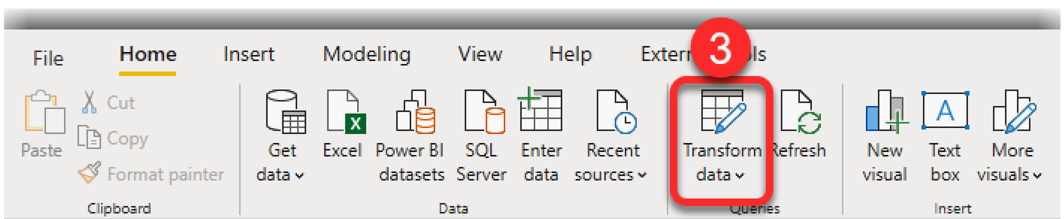


Figure 12.42 – Changing the connection mode from Connect Live to DirectQuery on top of an AAS model

4. Click **Add a local model**, as illustrated in the following screenshot:

To make changes, you need a DirectQuery connection

This .pbix has a live connection to a remote model. To make changes such as renaming columns, adding data from multiple sources, and more, you need DirectQuery. To switch to a DirectQuery connection, add a local model to the .pbix. Keep in mind, this change is permanent.

Learn more

4

Cancel Add a local model

Figure 12.43 – Confirming to add a local model

5. After clicking the **Add a local model** button, an empty **Power Query Editor** window shows up. Close the **Power Query Editor** window, as illustrated in the following screenshot:

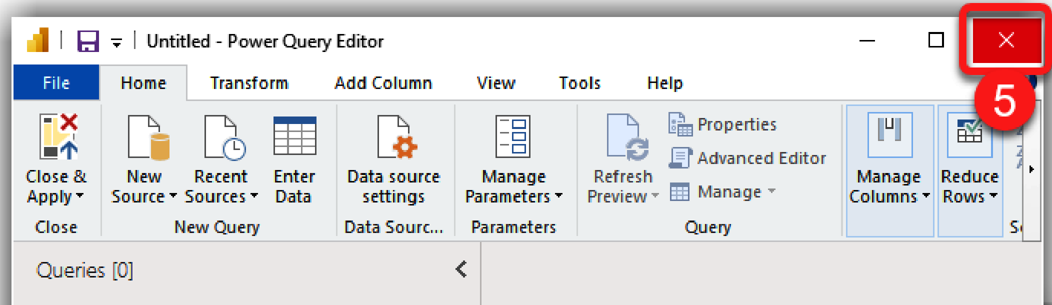


Figure 12.44 – Closing the empty Power Query Editor window

6. Go to the **Model** view again to see the changes (the tables' color changed from black to blue, and the icon for the tables has also changed), as illustrated in the following screenshot:

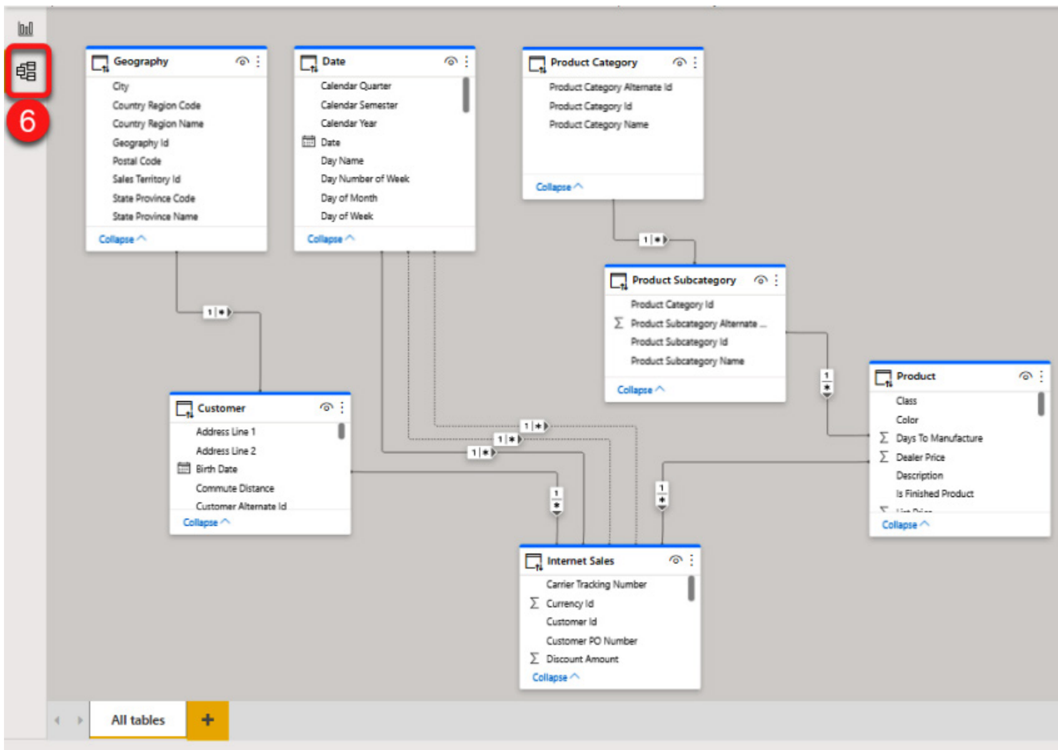


Figure 12.45 – The look and feel of the Model view after turning the connection mode to DirectQuery

7. Switch back to the **Report** view.

8. Right-click the `Internet Sales` table and then click **New column**, as illustrated in the following screenshot:

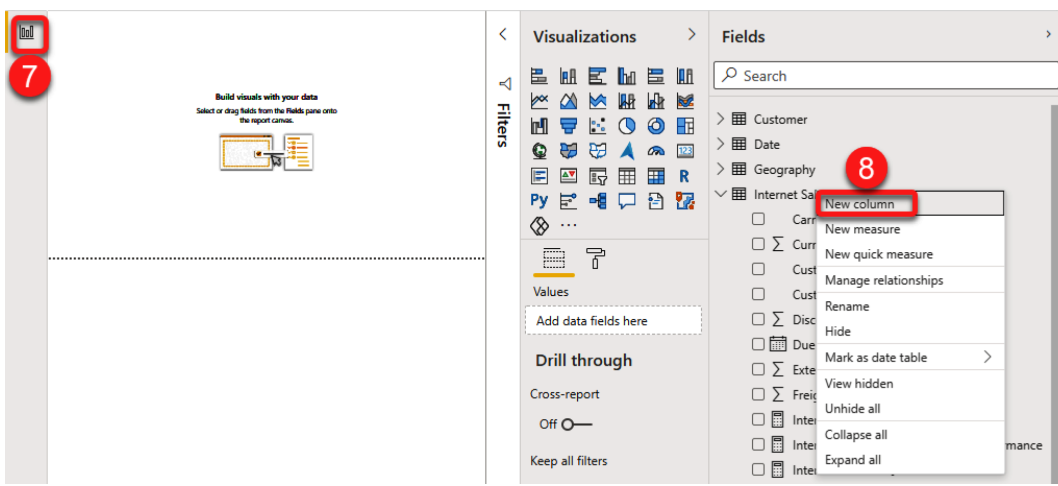


Figure 12.46 – Creating a new column when running DirectQuery on an AAS model

9. Use the following **Data Analysis Expressions** (**DAX**) expression to create a new calculated column:

```
Unit Price Range Band =
    SWITCH(
              TRUE()
            , 'Internet Sales'[Unit Price] <= 100, "Low"
            , AND('Internet Sales'[Unit Price] >= 101,
'Internet Sales'[Unit Price] <= 1000), "Medium"
            , AND('Internet Sales'[Unit Price] >= 1001,
'Internet Sales'[Unit Price] <= 3000), "High"
            , "Very High"
    )
```

Figure 12.47 – Creating a Unit Price Range Band calculated column

10. Create another calculated column using the following DAX expression to sort the `Unit Price Range Band` column:

```
UnitPriceRangeBandSort =
    SWITCH(
            TRUE()
          , 'Internet Sales'[Unit Price] <= 100, 1
          , AND('Internet Sales'[Unit Price] >= 101,
'Internet Sales'[Unit Price] <= 1000), 2
          , AND('Internet Sales'[Unit Price] >= 1001,
'Internet Sales'[Unit Price] <= 3000), 3
          , 4
    )
```

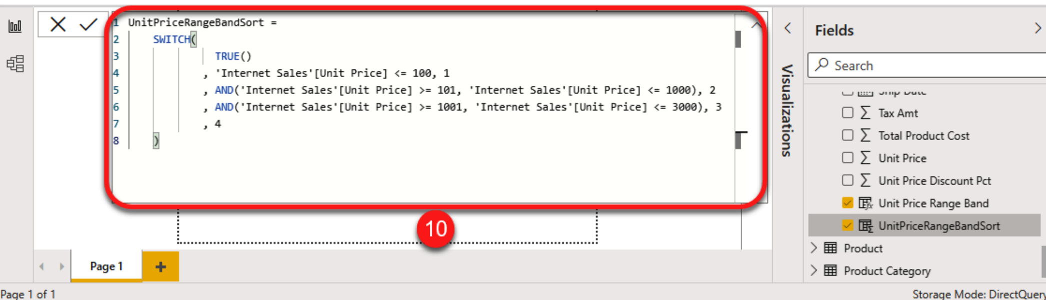


Figure 12.48 – Creating a UnitPriceRangeBandSort calculated column

11. Sort the `Unit Price Range Band` column by the `UnitPriceRangeBandSort` column, as illustrated in the following screenshot:

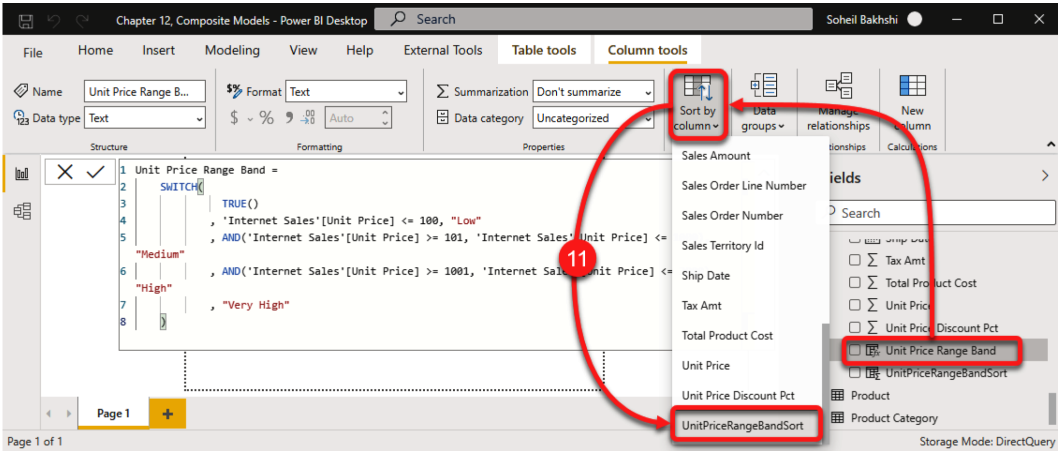


Figure 12.49 – Sorting a column by another column

As you can see, the **Data** view is not available as the dataset is now in **DirectQuery** mode. Therefore, to see the changes we make, we have to use a table visual. The following screenshot shows a table visual with the new calculated columns we built:

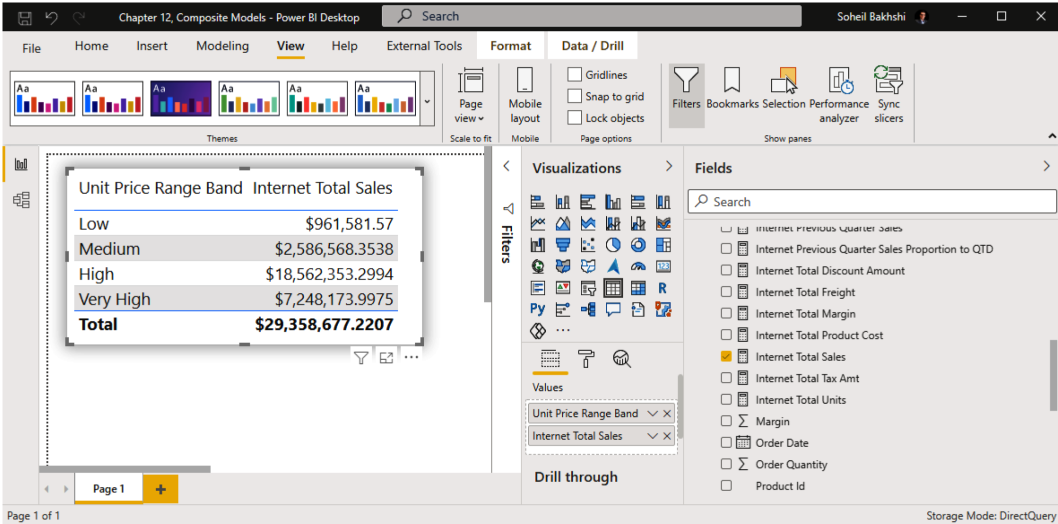


Figure 12.50 – Testing the results in a table visual when the dataset is in DirectQuery storage mode

Now that we meet the business requirements we can visualize the data as desired, but the last piece of the puzzle is to publish the report to the service. The following screenshot shows a lineage view of the report after it is published to the Power BI service:

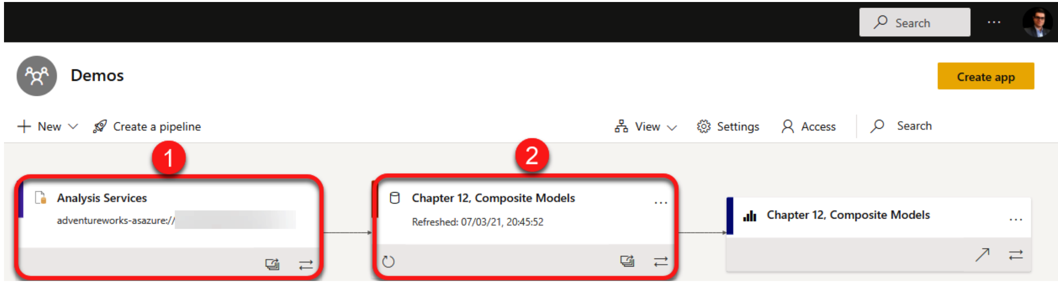


Figure 12.51 – Lineage view of a published report in the Power BI service

Now, let's revisit the new terminologies. We are **running DirectQuery on** a semantic model hosted in AAS from the `Chapter 12, Composite Models` dataset. Therefore, we created a chain. As shown in *Figure 12.51*, the chain length is **2**. If we create another dataset on top of the `Chapter 12, Composite Models` dataset, then the chain length is **3**.

> **Note**
> When writing this book, the maximum **chain length** allowed is 3; therefore, we cannot create a composite model with a chain length of 4.

While the preceding scenario is a simplistic one, I hope it gives you an idea of how to use the new composite models. I am sure you can see as many benefits in using the new composite model as I do, but keep in mind that you always want to make sure you get enough benefits out of it before changing your existing data models' design.

To learn more about composite models, I encourage you to check the Microsoft documentation here:

`https://docs.microsoft.com/en-us/power-bi/connect-data/desktop-directquery-datasets-azure-analysis-services?WT.mc_id=5003466`

Summary

In this chapter, we learned about different types of SCDs. We also learned about OLS in Power BI Desktop and Tabular Editor. We now know what dataflows are and in which scenarios we can consider using them as our self-service ETL tool. Last, but not least, we learned about the new generation of composite models that support **DirectQuery** with AAS and other Power BI datasets.

This is the last chapter of this book, I hope you enjoyed reading it and that you learned some new techniques and ideas for dealing with data modeling in Power BI. For more details about the topics we discussed in this book—and future updates—keep an eye on my website, `www.biinsight.com`.

Happy data modeling!

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

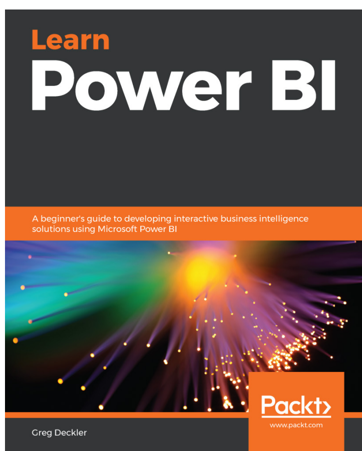
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Learn Power BI

Greg Deckler

978-1-83864-448-2

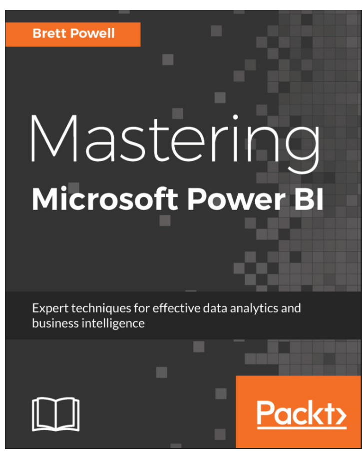
- Explore the different features of Power BI to create interactive dashboards
- Use the Query Editor to import and transform data
- Perform simple and complex DAX calculations to enhance analysis
- Discover business insights and tell a story with your data using Power BI
- Explore data and learn to manage datasets, dataflows, and data gateways
- Use workspaces to collaborate with others and publish your reports

Microsoft Power BI Quick Start Guide

Devin Knight, Mitchell Pearson, Bradley Schacht, Erin Ostrowsky

ISBN: 978-1-80056-157-1

- Connect to data sources using import and DirectQuery options
- Use Query Editor for data transformation and data cleansing processes, including writing M and R scripts and dataflows to do the same in the cloud
- Design optimized data models by designing relationships and DAX calculations
- Design effective reports with built-in and custom visuals
- Adopt Power BI Desktop and Service to implement row-level security
- Administer a Power BI cloud tenant for your organization
- Use built-in AI capabilities to enhance Power BI data transformation techniques
- Deploy your Power BI desktop files into the Power BI Report Serve



Mastering Microsoft Power BI

Brett Powell

ISBN: 978-1-78829-723-3

- Build efficient data retrieval and transformation processes with the M Query Language
- Design scalable, user-friendly DirectQuery and Import Data Models
- Develop visually rich, immersive, and interactive reports and dashboards
- Maintain version control and stage deployments across development, test, and production environments
- Manage and monitor the Power BI Service and the On-Premises Data Gateway
- Develop a fully On-Premise Solution with the Power BI Report Server
- Scale up a Power BI Solution via Power BI Premium Capacity and Migration to SQL Server Analysis Services

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

aggregation
  implementing, at Date level 428
  implementing, at Year and Month level 436-440
  implementing, for non-DirectQuery data sources 427
  Manage Aggregations feature, using 441
  using 426
aggregation, implementing at Date level
  control measures, creating in base table 432
  Internet Sales table, summarizing 428, 429
  measures, creating in summary table 431, 432
  relationships, creating 430, 431
  summary table, hiding 433-435
artificial intelligence (AI) 206
auto date/time
  reducing, by disabling model size 417-420
Azure Active Directory (Azure AD) 502
Azure Analysis Services (AAS) 92, 174-177, 566
Azure Synapse Analytics 426

B

bidirectional relationship 385
Boyce-Codd normal form 32
Bronze data sources 180
business intelligence (BI) 184

C

calculated columns
  avoiding 403-405
calculated table
  creating 34-36
  creating, in Power BI Desktop 41
calculation groups
  DAX functions 490, 491
  format string issue, fixing 488, 490
  implementing, to handle time intelligence 481-487
  requisites 479, 480
  terminologies 480
  testing 487, 488
  using 479
carriage return (CR) 153
chief executive officer (CEO) 515
column cardinality 120

columns
adding, from examples 209-211
binning 336-340
calculated columns 335
custom column, adding 206- 209
duplicating 211-213
grouping 336-340
merging 204, 20
properties 341-346
splitting, by delimiter 201-204
comma-separated values (CSV) file 142, 149-155
Common Data Service (CDS) 92
common data sources
data, obtaining 142
composite keys
handling 355-360
composite models
about 565, 566
chaining 566
chain length 567-574
terminologies 566
computed entity 547
configuration tables
dynamic conditional formatting, with measures 396-402
segmentation 393, 394
using 393
connection modes
Connect Live mode 184
data import mode 182
DirectQuery mode 183
working with 181, 182
Connect Live mode
about 184
application 184
limitations 184
Custom Connectors software development kit (SDK) 92
Customer Relationship Management (CRM) 177
custom functions
about 132-137
recursive functions 138
custom types 101

D

data, obtaining
from AAS 174
from common data sources 142
from CSV file 149-155
from Excel 156-164
from folder 142-149
from OData feed 177-179
from Power BI dataflows 169, 170
from Power BI datasets 164-169
from SQL Server 171, 172
from SSAS 174
from text file 150-155
from text file (TXT) 149
from TSV file 149-155
Data Analysis Expressions (DAX)
about 92, 494, 571
Date and Time dimensions, creating 276
date dimension, generating with 76-78
time dimension, creating 84-87
data cleansing 229
dataflows
about 545
computed entities, creating 560-562
creating 548-551
entities, creating 552-556
exporting 562, 563
importing 564, 565

linked entities, creating from other dataflows 557-560
terminologies 547
using, scenarios 546
data import mode
about 182
application 182
limitations 182
data modeling
about 56
in Power BI 12
in Power BI Desktop 320
data model layer, Power BI
about 6
Data view 7, 8
Model view 9
data preparation best practices
size of queries, optimizing 312
data preparation considerations
about 288
case sensitivity, appreciating 292
portion of data, loading 288-292
query folding 292
dataset storage mode
about 187, 188
Composite mode/Mixed mode 501
DirectQuery mode 501
Import mode 501
types 187
data source certification
about 180
Bronze data sources 180
Gold data sources 181
Platinum data sources 181
Silver data sources 181
data type conversion
about 192-200, 302
best practices 302-312
Dataverse 92
data visualization layer, Power BI
about 9
Report view 10
date dimension
dealing with 235-238
generating, with DAX 76-78
valid dates, detecting 57-66
Date table
marking, as date table 79-84
DateTime
dealing with 235-238
DateTimeZone
dealing with 236-238
DAX Studio
using 42, 43
Degenerate Dimensions 258
delimiter
used, for splitting column 201-204
denormalization 19-25
Development (Dev) 127
dimensional modeling 16
dimensions and facts
defining 248
identifying 242, 243
linkage, between existing tables 244, 245
lowest required grain of Date and Time, searching 246, 247
number of tables, in data source 244
potential dimensions, determining 249
potential facts, determining 249-252
Dimensions tables
creating 252
currency 263
customer 263, 264

Date dimension 269-273
geography 253-257
product dimension 259-262
Sales Demographic 265-268
sales order 257-259
Time dimension 273-276
dimension tables 17
DirectQuery mode
about 183
application 183
limitations 183
DirectQuery or Dual storage modes 293
dynamic RLS
implementing 511
implementing, scenarios 511-528

E

Enterprise Resource Planning (ERP) 143
entities 547
Excel 156-164
Excel workbooks 14
extract, transform, and load (ETL) 531

F

Factless Fact tables 377
fact tables
about 17
creating 277- 285
fields
about 332, 547
columns 335
custom formatting 334
data types 332, 333
hierarchies 346, 347
measures 347
Fixed Dimensions 532
foreign key 354

G

gigabyte (GB) 182
globally unique identifier (GUID) 93
Gold data sources 181
Group By
working with 218-220

H

High Availability (HA) 171

I

inactive relationships
dealing with 388
multiple direct relationships,
between two tables 390-392
reachability, via multiple
filter paths 388-390
incremental refresh
about 455-457
configuring, in Power BI
Desktop 457-463
testing 463-466
iterative data modeling approach
about 29
business logic, demonstrating
in data visualization 31
data modeling 30
data preparation, based on
business logic 30
information, obtaining from business 30
logic, testing 31
professional data modelers concept 32

J

JavaScript Object Notation
    (JSON) 521, 562
joins
  FullOuter 227
  Inner 227
  LeftAnti 227
  LeftOuter 227
  RightAnti 227
  RightOuter 227

L

licensing considerations, Power BI
  about 25
  calculation groups 27, 28
  Dataflows 28
  incremental data load 27
  individual dataset size, determining 26
  shared datasets 28
line feed (LF) 153
linked entity 547

M

machine learning (ML) 206
Manage Aggregations feature
  about 426, 427
  used, for managing aggregations in
    Power BI Desktop 443-449
  used, for testing aggregation 449-455
  using 441-443
measures
  about 347
  explicit measures 351
  implicit measures 348-350
  textual measures 351, 352
megabytes (MB) 184
model organization
  about 406
  folders, using 413
  insignificant model objects, hiding 406
  measure tables, creating 409-413
Multidimensional Expressions
    (MDX) 175

N

naming conventions
  for Power BI developers and
    data modelers 314, 315
normal entity 547
Notepad++
  download link 153
numbers
  extracting, from text 233-235

O

object-level security (OLS)
  about 494, 537
  implementing 537-540
  members, assigning to roles in
    Power BI service 542, 543
  roles, assigning in Power BI
    service 543, 544
  roles, validating 540-542
OData feed 177-179
on-premises data gateway 558
Open Data Protocol (OData) 14, 177, 288
optimized queries, techniques
  query load, disabling 314
  summarization (Group by) 313
  unnecessary columns and rows,
    removing 312, 313

P

Parent-Child hierarchies
  about 466-468
  depth, identifying 468, 469
  levels, creating 470-475
period-over-period calculations
  performing 66-76
Platform as a Service (PaaS) 174
Platinum data sources 181
Portable Document Format (PDF) file 142
Power BI
  data flow process 11, 12
  data modeling 12
  efficient data model, building 13-15
  licensing considerations 25
  Power Query (M) formula language 92
Power BI dataflows 169, 170
Power BI datasets 164-169
Power BI Desktop
  calculated tables, creating 41
  data modeling 320
  incremental refresh,
    configuring 457-463
  Microsoft Contoso Sales sample,
    download link 5
  URL 4
Power BI layers
  about 4
  data model layer 6
  data preparation layer (Power Query) 6
  data visualization layer 9
Power BI Report Server, RLS roles
  members, assigning to 499-501
Power BI service, RLS roles
  members, assigning to 498
Power Query
  about 92
  expressions 93, 94
  query formula step, creating 93
  types 93, 100
  values 93, 94
  variables 93
Power Query Diagnostics tool 299
Power Query Editor
  about 101-103
  Advanced Editor 113
  Data View pane 109, 110
  Queries pane 103
  Query Settings pane 106
  Status bar 113
Power Query features, for data modelers
  about 115
  column distribution 120-123
  column profile 123
  column quality 116, 118, 119
Power Query Online 28, 545
Power Query, types
  custom types 101
  primitive types 100, 101
Power Query, values
  primitive values 94
  structured values 95-99
Premium Per User (PPU) 182
primary key 354
primitive types 100, 101
primitive values 94
Production (Prod) 127
Project Web App (PWA) 177

Q

queries
  about 547
  appending 221-224
  duplicating 228, 229
  merging 224-227
  referencing 228, 229
Queries pane, Power Query Editor
  about 106
  constant values 104
  custom functions 103
  groups 104-106
  query parameters 104
  Query Properties 107-112
  tables 103
Query Editor
  queries, organizing 300-302
query folding
  about 292, 293
  and data sources 294
  best practices 295-299
  impact, on data refresh 292
  indications 294, 295
query parameters 124-131

R

recursive functions 138
relationship cardinalities
  about 360
  many-to-many relationships 361-363
  one-to-many relationships 361
  one-to-one relationships 361
relationships
  bidirectional relationships 366-369
  cardinalities 360
  filter propagation behavior 364, 365
  primary keys/foreign keys 354
  using 353, 354
report level measures 174
RLS implementation
  approaches 502
  flow 501, 502
roleplaying dimensions
  implementing 475-478
roles
  validating 496
row-level security (RLS)
  about 494, 495
  roles 495
  rules 496
  validating roles 496, 497
rows
  filtering 214-218

S

SAP Business Warehouse (SAP BW) 181
schema modeling
  versus transactional modeling 16
segmentation 393
semantic model 13
server-side/client-side data processing 293
Silver data sources 181
slowly changing dimensions (SCDs)
  dealing with 530, 531
  SCD type 1 (SCD 1) 532
  SCD type 2 (SCD 2) 532-536
  SCD type zero (SCD 0) 532
Smart Date Key 237
snowflaking 18
SQL Server
  about 171, 172
  data warehouse 15

SQL Server Analysis Services (SSAS)
  about 4, 174, 566
  multidimensional model 174, 175
  tabular model 174, 175
SQL Server Analysis Services Tabular Models (SSAS Tabular) 92
SQL Server Management Studio (SSMS) 449
star schema modeling 16
static RLS
  implementing 502-511
storage modes
  types 185
  working with 185, 186
structured values
  function value 99
  list value 95
  record value 96
  table value 97-99
subject-matter experts (SMEs) 180
Surrogate Key 532

T

tables
  about 320
  calculated tables 326-331
  featured tables 325
  properties 321-324
tab-separated values (TSV) file 149-155
Tabular Editor 350
temporal mechanism 533
temporal tables 533
text
  numbers, extracting from 233-235
text file (TXT) 142-155
third normal form 32
time dimension
  creating, with DAX 84-87
time intelligence 56
transactional modeling
  versus star schema modeling 16
Transact-SQL (T-SQL) script 171

U

Uniform Resource Locator (URL) 178
Universal Time Coordinate (UTC) 238
User Acceptance Testing (UAT) 127
User Principal Name (UPN) 511

values
  replacing 229-232
virtual tables
  about 34
  relationships 44-56
  results, displaying virtually 41
  using, in measure 36-40

xVelocity engine 13