Supercharge Power BI

Power BI is Better When You Learn To Write DAX

Matt Allington

Inside Front Cover - This page intentionally blank

Supercharge Power BI
Power BI is Better When You Learn to Write DAX

by
Matt Allington

Holy Macro! Books
PO Box 541731
Merritt Island, FL 32954

Supercharge Power BI

© 2018 Tickling Keys, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information or storage retrieval system without permission from the publisher. Every effort has been made to make this book as complete and accurate as possible, but no warranty or fitness is implied. The information is provided on an "as is" basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Author: Matt Allington

Layout: Jill Bee

Copyediting: Kitty Wilson

Cover Design: Emrul Hasan & Shannon Travise

Cover Illustration: Freepik

Indexing: Nellie Jay

Published by: Holy Macro! Books, PO Box 541731, Merritt Island FL 32954, USA

Distributed by: Independent Publishers Group, Chicago, IL

First Printing: January, 2018

ISBN: 978-1-61547-054-9 Print, 978-1-61547-237-6 PDF, 978-1-61547-360-1 ePub, 978-1-61547-131-9 Mobi

Library of Congress Control Number: 2017961952

Table of Contents

Introduction

Power BI is the latest and greatest in business intelligence (BI) software. There are so many great things about this Microsoft product that it is hard to know where to start. Perhaps one of the most important things to note about Power BI is that it is designed with business analysts and Excel users in mind. You do not need to be an IT professional to be able to use this software well.

Power BI has capabilities across four important phases of a business intelligence project:

- It has a powerful data acquisition engine that helps a user fetch and load the data needed. The underlying technology that supports data acquisition is called Power Query (accessed via the Get Data menu), and the programming language is called M.
- It has a powerful data modelling engine that allows the user to model the loaded data to make it more useful than it is in the raw state. The underlying technology that supports data modelling is called Power Pivot, and the programming language is called DAX (short for Data Analysis Expressions).
- It has a modern visualisation engine built using the latest technologies so you can build interactive reports. The Power BI visual engine has been open sourced so that anyone with the necessary skills can build new visuals to use and share within Power BI.
- Finally, it has a framework that supports multiple ways to share data with others, including a cloud-based web environment and native mobile apps. These tools make it easy to share reports and dashboards with other people who need to see and interact with the data. The tool to share Power BI reports is called the Power BI service, or PowerBI.com.

This book, *Supercharge Power BI: Power BI Is Better When You Learn to Write DAX*, teaches you the skills you need to use Power Pivot (the modelling tool bundled with Power BI) and the DAX language. Power Pivot brings everything that is good about enterprise-strength BI tools directly to you right inside Power BI Desktop—and without the negative time and cost impacts normally associated with big-scale BI projects. In addition, it is not just the time and money that matter. The fact that you can do everything yourself directly inside Power BI is very empowering. Analyses that you would never have considered viable in the past are now "can do" tasks within the current business cycle.

It is worth pointing out that you can use Power BI without learning Power Pivot. However, Power BI is definitely better when you learn to write DAX. If you don't invest time in learning DAX and Power Pivot, you will be able to take advantage of only the basic capabilities of the Power BI tool. Imagine being able to use only the `SUM()` function in Excel. You would be able to produce only very basic and simplistic spreadsheets. Similarly, with Power BI, if you don't learn the DAX language and how the Power Pivot engine works, you will be limited to simplistic capabilities that restrict the value you can get from the tool.

There is another significant benefit to learning Power Pivot and DAX for Power BI. These skills are fully transferable to Excel. Although in this book you will be learning the DAX language using the Power BI user interface, you will be able to easily move these new skills into Power Pivot for Excel should you want to do that. And who wouldn't?

Supercharge Excel

Supercharge Power BI: Power BI Is Better When You Learn to Write DAX has been written specifically to teach Power Pivot and DAX using Power BI Desktop. I have written a sister book, *Supercharge Excel: When You Learn to Write DAX for Power Pivot*. These two books cover the same basic content but with a different user interface. Because the skills you will learn in this book are fully transferable to Power Pivot for Excel and vice versa, you really need only one of these books to secure the required skills. However, if you want to learn about the differences in the UI and practice what you have learnt, then reading *Supercharge Excel* will certainly help you cement your learning across the different UIs.

Why You Need This Book

I am a full-time Power BI consultant, trainer, and BI practitioner. I have taught many Excel users how to use Power Pivot and Power BI at live training classes, and I have helped countless others online at various Power BI forums. This teaching experience has given me great insight into how Excel users learn Power BI and what resources they need to succeed. Power BI is very learnable, but it is very different to Excel; you definitely

need some structured learning if you want to be good at using this tool. I have learnt that Excel users need practice, practice, practice. The book you're reading right now, *Supercharge Power BI: Power BI Is Better When You Learn to Write DAX*, is designed to give you practice and to teach you how to write DAX. If you can't write DAX, you will never be good at Power BI or Power Pivot.

I refer above to *Excel users*, and that is quite deliberate. I have observed that Excel professionals learn DAX differently than do IT/SQL Server professionals. IT/SQL Server professionals are simply not the same as Excel business users. SQL Server professionals have a solid knowledge of database design and principles, table relationships, how to efficiently aggregate data, etc. And of course there are some Excel users who also have knowledge about those things. But I believe IT/SQL Server professionals can take a much more technical path to learning DAX than most Excel users because they have the technical grounding to build upon. Excel users need a different approach, and this book is written with them in mind. That is not to say that an IT/SQL Server professional would not get any value from this book/approach; it really depends on your learning style. But suffice it to say that if you are an Excel professional who is trying to learn DAX, this book was written with your specific needs in mind.

Incremental Learning

I am an Excel user from way back—a long way back actually. I'm not the kind of guy who can sit down and read a novel, but I love to buy Excel reference books and read them cover to cover. And I have learnt *a lot* about Excel over the years by using this approach. When I find some new concept that I love and want to try, most of the time I just remember it. But sometimes I add a sticky note to the page so I can I find it again in the future when I need it. In a way, I am incrementally learning a small number of new skills on top of the large base of skills I already have. When you incrementally learn like this, it is relatively easy to remember the detail of the new thing you just learnt.

It's a bit like when a new employee starts work at a company. Existing employees only have to learn the name of that one new person. But the new employee has to learn the name of every person in the entire company. It is relatively easy for the existing employees to remember one new name and a lot harder for the new person to start from scratch and learn all the names. Similarly, when you're an experienced Excel user reading a regular Excel book, you already know a lot and need to learn only a few things that are new—and those new bits are likely to be gold. It is easy to remember those few new things because often they strike a chord with you. Even if you don't remember the details, the next time you face a similar problem, you'll remember that you read something about it once, and you'll be able to go find your book to look it up.

Well, unfortunately for seasoned Excel users, Power BI is a completely different piece of software from Excel. It shares some things in common (such as some common formulas), but many of the really useful concepts are very different and completely new. They are not super-difficult to learn, but indeed you will need to learn from scratch, just as that new employee has to learn everyone's name. Once you get a critical mass of new Power BI knowledge in your head, you will be off and running. At that point, you will be able to incrementally learn all you want, but until then, you need to read, learn, and, most importantly, practice, practice, practice.

Passive vs. Active Learning

I think about learning as being either passive or active. An example of passive learning is lying in bed, reading your Power BI book, nodding your head to indicate that you understand what is being covered. When you learn something completely new, you simply can't take this approach. I read a lot of Power Pivot books early in my discovery, but the first time I sat in front of my computer and wanted to write some DAX, I was totally lost. What I really needed to do was change from a passive learning approach to an active approach, where I was participating in the learning process rather than being a spectator.

Passive learning on its own is more suited to incrementally adding knowledge to a solid base. Passive learning is not a good approach when you are starting something completely new from scratch. I'm not saying that passive learning is bad. It is useful to do some passive learning in addition to active learning, but you shouldn't try to learn a completely new skill from scratch using *only* passive learning.

How to Get Value from This Book

There are more than 40 "Here's How" worked-through examples and more than 70 individual practices exercises in this book. That gives you more than 110 opportunities to learn and practice. Make the most of these opportunities to develop your skills; after all, that is why you purchased this book.

If you think you can get value from this book by reading it and not doing the practice exercises, let me tell you: You can't. If you already know how to complete a task and you have done it before, then just reading is fine. However, if you don't know how to do a task or an exercise, then you should practice in front of your computer. First try to do an exercise without looking at the answers. If you can't work it out, then reread the worked-through examples (labelled "Here's How") and then try to do the exercise again. Practice, practice, practice until you have the knowledge committed to memory and you can do it without looking.

Don't Treat This Like a Library Book

When we were kids going to school, most of us were taught that you should not write in library books. And I guess that is fair enough. Other people will use a library book after you are finished, and they probably don't want to read all your scribbles. Unfortunately, the message that many of us took away was "Don't write in *any* book *ever*." I think it is a mistake to think that you can't write in your own books. You bought it, you own it, so why can't you write in it? In fact, I would go one step further and say *you should* write in the reference books you own. You bought them for a reason: to learn. If you are reading this book and want to make some notes to yourself for future reference, then you should definitely do that.

But I guess I am forgetting the eBook revolution. I know you can't write in an eBook, but I know you can highlight passages of text in a Kindle, and I assume you can do something similar in other eBooks. You can also type in your own notes and attach them to passages of text in many eBooks. There are lots of advantages of eBooks, and the one that means the most to me is the fact that I can have a new book in front of me just moments after I have decided to buy it.

Personally I find that eBooks are not a great fit as reference books. I prefer to have a tactile object so I can flip through the pages, add sticky notes, and so on. But that is just me, and we are all different. I am sure there are plenty of people in both camps. On the upside, eBooks are usually in colour, and printed books (like this one) are more often in black and white. Whichever camp you are in—eBook or physical book—I encourage you to write in this book and/or make notes to yourself using the eBook tools at your disposal. Doing so will make this book a more useful, personalised tool well into the future.

There Are No Pivot Tables in Power BI

In Microsoft Excel, the most common way to aggregate data for BI-style reporting is to use a pivot table. But there are no pivot tables in Power BI. What's worse (and confronting) is that there isn't even a spreadsheet grid for entering data on a page. Although Power BI has no pivot tables, it does allow you to use matrixes. A Power BI matrix is very similar to a pivot table and is (in my view) the best visual to use when you are learning to write DAX. Throughout this book, you will in many cases set up a matrix and then place your new measures inside that matrix so that you can visualise the results of your work. Once you have seen that the results of your measures are working as you expect, it is very easy to change the matrix into another type of visual to better display the data.

Exercise Data

It is surprisingly difficult to create your own database of meaningful data to use for data analysis practice. Think about the data that exists in a commercial retail business, for example: customer data, finance data, sales data, products, territories, etc. And it is not a simple task to create a meaningful quantity of realistic data from scratch; it is a lot of work. Microsoft has created a number of sample databases that anyone can download and use for free. I use a modified version of the Microsoft AdventureWorks database throughout this book, provided to you in Microsoft Access format. You can download a copy of it by going to http://xbi.com.au/learndax. (Note that you do not need to have Microsoft Access installed to use this database.) This is the same sample database I use in my live training classes.

AdventureWorks contains sample data for a fictitious retail bicycle company that sells bikes and accessories in multiple countries. The data consists of the customers, products, and territories for the AdventureWorks business, along with five years of transactional sales history. The examples I use in this book therefore focus on reporting and analysis that would apply to a retail business, including such things as sales results, profit margins, customer activity, and product performance.

Clearly, not everyone who wants to learn to write DAX will operate in a retail environment. However, the retail concepts covered in this book should be familiar to everyone. So it doesn't matter if your specific BI needs

are for something other than retail. The scenarios in this book are explained throughout, so you don't need to be a retail expert to complete or understand the exercises.

Getting Help Along the Way

Hopefully you will be able to complete the practice exercises in this book on your own. But sometimes you might need to ask someone a question before you can move forward. I encourage you to become a member of http://powerpivotforum.com.au and participate as someone who asks questions and also as someone who helps others when they get stuck. Answering questions for other people is a great way to cement your learning and build depth of knowledge. You will notice from the URL that this is an Aussie forum, but it is open to everyone. At this writing, only 15% of all traffic at the forum is from Australia, with the balance coming from more than 130 other countries around the world. I suggest that you sign up and get involved; your DAX will be better for it.

You can find a subforum dedicated to this book at http://xbi.com.au/scpbiforum. In the unfortunate event that there are errors in this book, you can go to this subforum for details.

How This Book Is Organised

I've organised this book to make sense to a new Power BI user. The general structure of the chapters is as follows:

- Each chapter title begins with either "DAX Topic" or "Concept." The former type covers one or more specific DAX formulas, including the syntax and usage; the latter type covers one or more principles that you need to understand in order to be competent with Power BI. I've ordered the chapters so that you can learn incrementally.
- Each "Concept" chapter starts with a description of the concept, and each "DAX Topic" chapter starts with some information about the DAX language to help you understand the topic.
- Almost every chapter provides at least one worked-through example. When you see "Here's How," you know you're reading one of those, and it's time to sit in front of your computer and follow along with me as I explain the concept. See the "Table of Here's How Sections" on page 184.
- Almost every chapter includes a number of practice exercises that help you practice what you have learnt. You will find guidelines to complete the exercises, and you can also find the answers at the end of the book. I recommend that you complete the exercises first and only then look at the answers to check that you got the correct results. This way you can cement the learning you are getting from this book.
- DAX is a lot like Excel in that there is often more than one way to do something. If you do an exercise differently than I show how to do it, as long as you get the correct/same answer, all is good.

Naming Conventions

This book uses best-practice naming conventions for Power Pivot and Power BI:

- There are no spaces in table names, like this:

```
TableName
```

- Columns in tables *always* include the table name followed by the column name in square brackets, like this:

```
TableName[ColumnName]
```

- Measures *never* include a table name, they often include spaces, and they are wrapped in square brackets, like this:

```
[MeasureName]
```

- Measure and column formulas are written with the formula name (without the square brackets) followed by the formula, like this:

```
Total Sales = SUM(Sales[ExtendedAmount])
```

1: Concept: Introduction to Data Modelling

The *data modelling engine* that is used inside Power BI is the same one used in Power Pivot for Excel. *Data modelling* is not a term that is often familiar to business users as it is normally the domain of IT BI professionals. But this is no longer the case, thanks to the introduction of Power BI and Power Pivot for Excel.

What Is Data Modelling?

Data modelling is the process of taking data from various sources; loading, structuring, and relating data logically to other data; and enhancing, embellishing, and generally preparing the data for use. The objective is to allow the data to be used without having to write a custom query every time you want to look at a different subset of data.

The data modelling process includes:

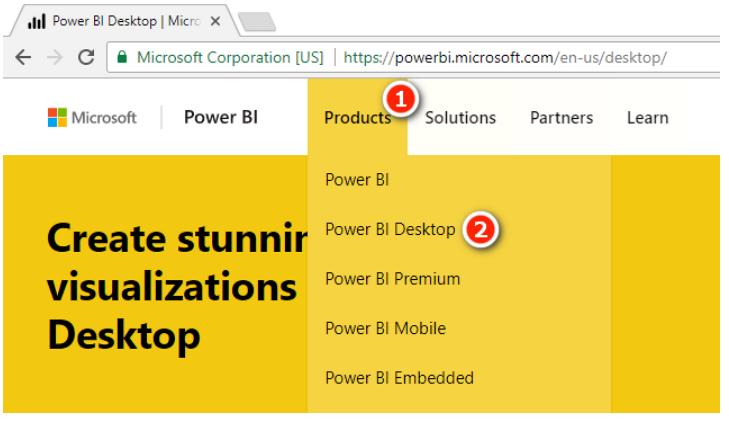
- Determining the optimal structure and shape of the source data to analyse, including whether to bring in all the data, full data, or summary data.
- Loading the data from the source into the data model (Power BI in this case).
- Defining the logical relationships between the various tables (which is similar to what you do with `VLOOKUP()` in Excel, except the data stays in the source table in Power BI).
- Defining data types (e.g., specifying whether a column of data is numeric or a column of currency values or a column of text fields).
- Creating new insights from the source data so that you can analyse concepts that don't exist natively in the source data but that can be calculated or created inside the data model. For example, if you have a table of transactional data with cost price and sell price, you can extend the data model to include calculations for margin, margin percentage, etc., even though these concepts are not explicitly in the source data. Once you have modelled these new facts in the data model, they can be reused over and over by people using your workbook.
- Giving meaningful names to your new business insights (i.e., to your measures).

When you learn the DAX language and join your tables of data in Power BI, you are actually learning data modelling. The term can be a little bit scary, but there is no reason to be concerned. By the time you have finished this book, you will be well on your way to being an accomplished data modeller using Power BI. Just use the techniques covered in this book and keep in mind that what you are actually doing is learning to be a data modeller.

Here's How: Getting Power BI Desktop

All the instructions in this book use Power BI Desktop as the data modelling tool. To download this free tool, follow these steps:

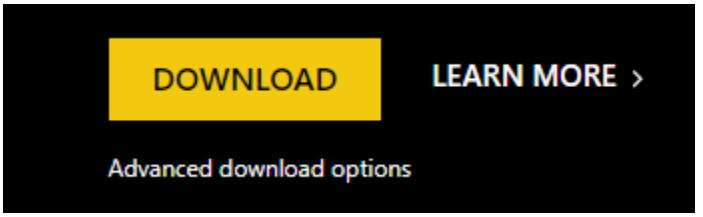
1. Navigate to http://powerbi.com and then go to the Products menu (see #1 below) and select Power BI Desktop (#2).



Note: At this writing, there are four other products shown above:

- Power BI is the link to PowerBI.com
- Power BI Premium is a capacity-based pricing model for large companies.
- Power BI Mobile refers to the free Power BI Apps for iOS and Android devices.
- Power BI Embedded allows developers to build Power BI reports directly inside their own custom software.

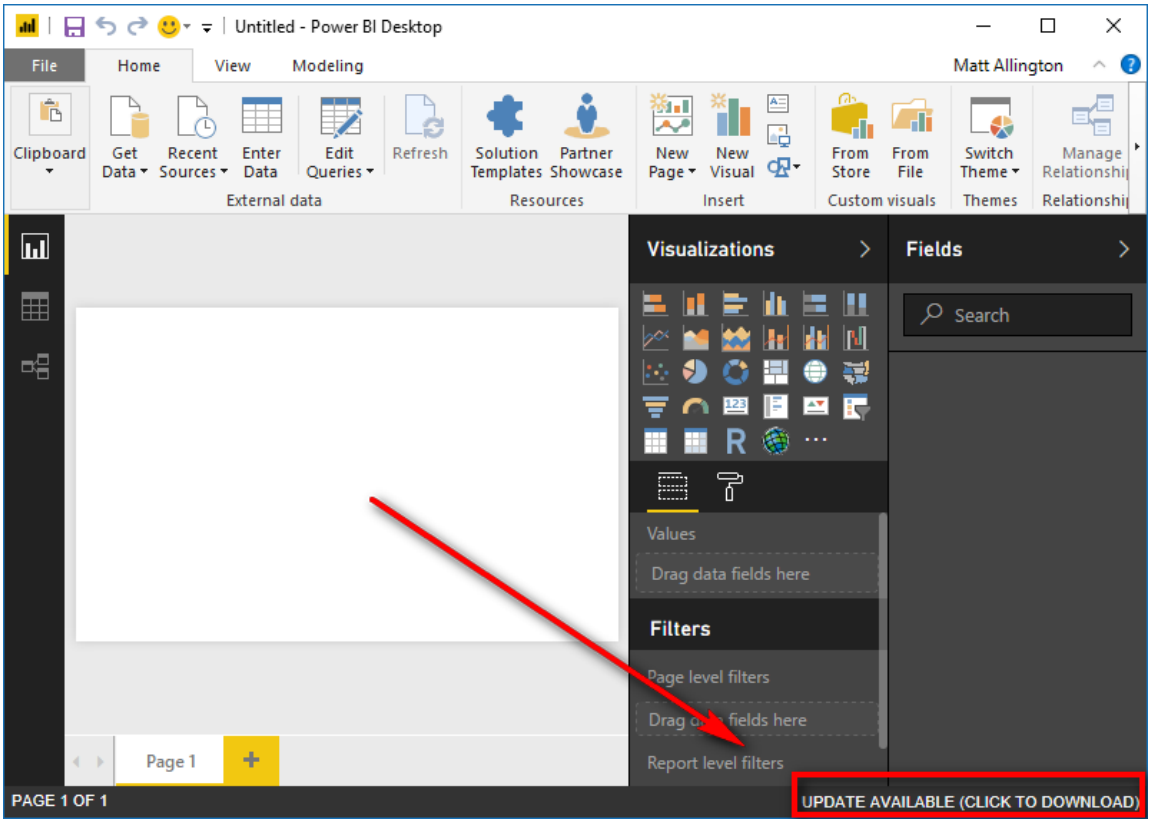
2. Once you arrive at the Power BI Desktop page, click the Download button.



Note: You need administrator rights on your PC to be able to install the software. Also note that the default download is the 64-bit version of Power BI Desktop. This is the best option for most people. Power BI Desktop 64-bit will work even if you have 32-bit Microsoft Office on your PC. If for some reason you need the 32-bit version, you can click Advanced Download Options and download it from there.

Updates to Power BI Desktop

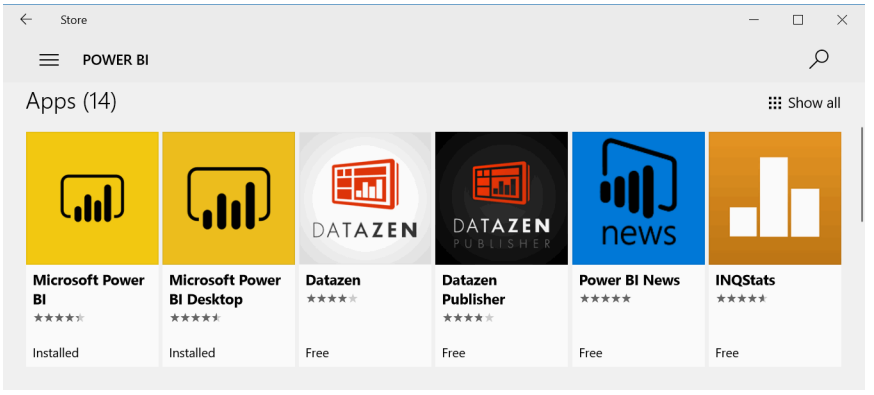
Power BI Desktop is constantly being updated, and new software updates are released every month. This is great because each month you will be able to access new and exciting features developed by the team at Microsoft. When there is a new version of Power BI Desktop available for you to install, you will see a notification in the bottom-right corner of the application, as shown below. You can simply click on this notification to download the latest version—but note that you need to close Power BI Desktop in order to complete the installation.



Note: One downside of Microsoft releasing new versions of Power BI Desktop every month is that it is inevitable that some of the screenshots in this book will look different to what you see on your screen, given that you will have a later version of the software.

Windows App Store

In October 2017, Microsoft added Power BI Desktop to the Windows App Store. To install from the App Store, click the Windows button in the taskbar on your PC and then type **store**. Open the App Store and then search for Power BI Desktop.



Note: In the image above, Microsoft Power BI refers to PowerBI.com. Microsoft Power BI Desktop (the second app listed) is the software you need to install.

The advantage of installing from the App Store is that Microsoft will update the software automatically each time there is a new version available. Note there may be an additional level of control from your IT department that determines when the updates are available.

Power BI Pro vs. Power BI Free Accounts

Power BI Desktop is a free data model authoring tool that is used to build data models and reports. After you build a report, you can share it as a file with other Power BI Desktop users (just as you can an xlsx file). Power BI Desktop can also publish your reports to PowerBI.com, where the reports can be easily shared with other Power BI users; this is the normal way to share.

You need a Power BI account in order to use PowerBI.com. There are two types of Power BI accounts: free and Pro. A free account allows you to use most of the functionality of PowerBI.com, but there are some notable exceptions. You cannot share your work with others other than public (unsecure) sharing, and you can't access the data by using Analyze in Excel or export to PowerPoint. If you want to share reports and dashboards with other users, all users who want to be part of the sharing must have Power BI Pro accounts.

2: Concept: Loading Data

The image below shows the data connector that appears when you connect to a SQL Server database. (There is a different data connector for each data source. You'll see how to get to the various data connector screens later in this chapter.) There are two modes that you can use in Power BI Desktop when loading data from a database like SQL Server: Import and DirectQuery.

Most data sources do not provide these two options and instead only allow you to use Import mode. This book focuses on Import mode, which makes a physical copy of the data from the source and loads it into Power BI Desktop. When you use Import mode, Power BI Desktop loads a complete copy of the source data into the data model as the first step in the process. Once it's loaded, you can share your pbix workbook with others, and there is no need for anyone else to have direct access to the source data. Alternatively, you can publish your reports to PowerBI.com and share the contents with others from there. When you publish a report to PowerBI.com, a complete copy of the data is loaded into the cloud also, without the need for it to access the source data.

When you load data, you have to decide which data to load, including which tables, which columns in each table, etc. I call this the *shape* of the data. The following "Here's How" shows how to load data that has been prepared for you. But you need to be aware that the process of deciding which data to load is an important part of the data modelling process, as discussed later in this chapter.

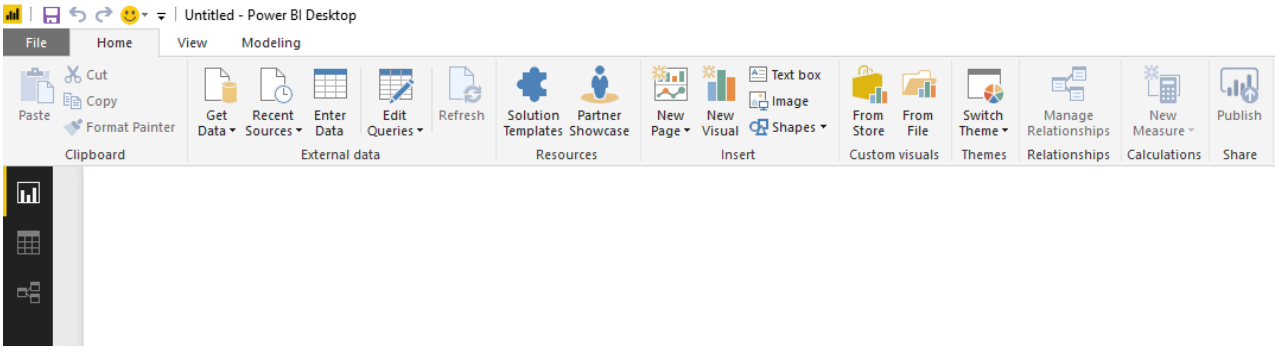
Here's How: Loading Data from a New Source

If you don't already have a copy of the custom version of the AdventureWorks database used in this book, you should download it now (from http://xbi.com.au/learndax), unzip it, and place it in a location that is easy for you to find. You are going to start by loading the following tables from the AdventureWorks Access database:

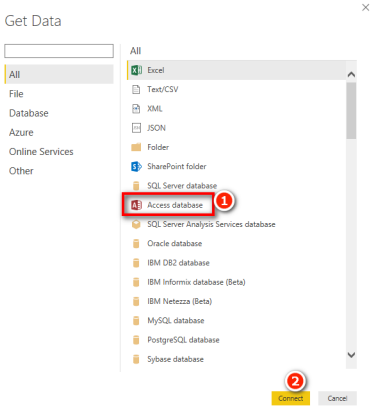
- `Sales`
- `Products`
- `Territory`
- `Calendar`
- `Customers`

The following steps show you how to load these tables and prepare them for use in Power BI:

1. Open Power BI Desktop. You should see a blank Power BI Desktop file with a menu along the top, as shown below.



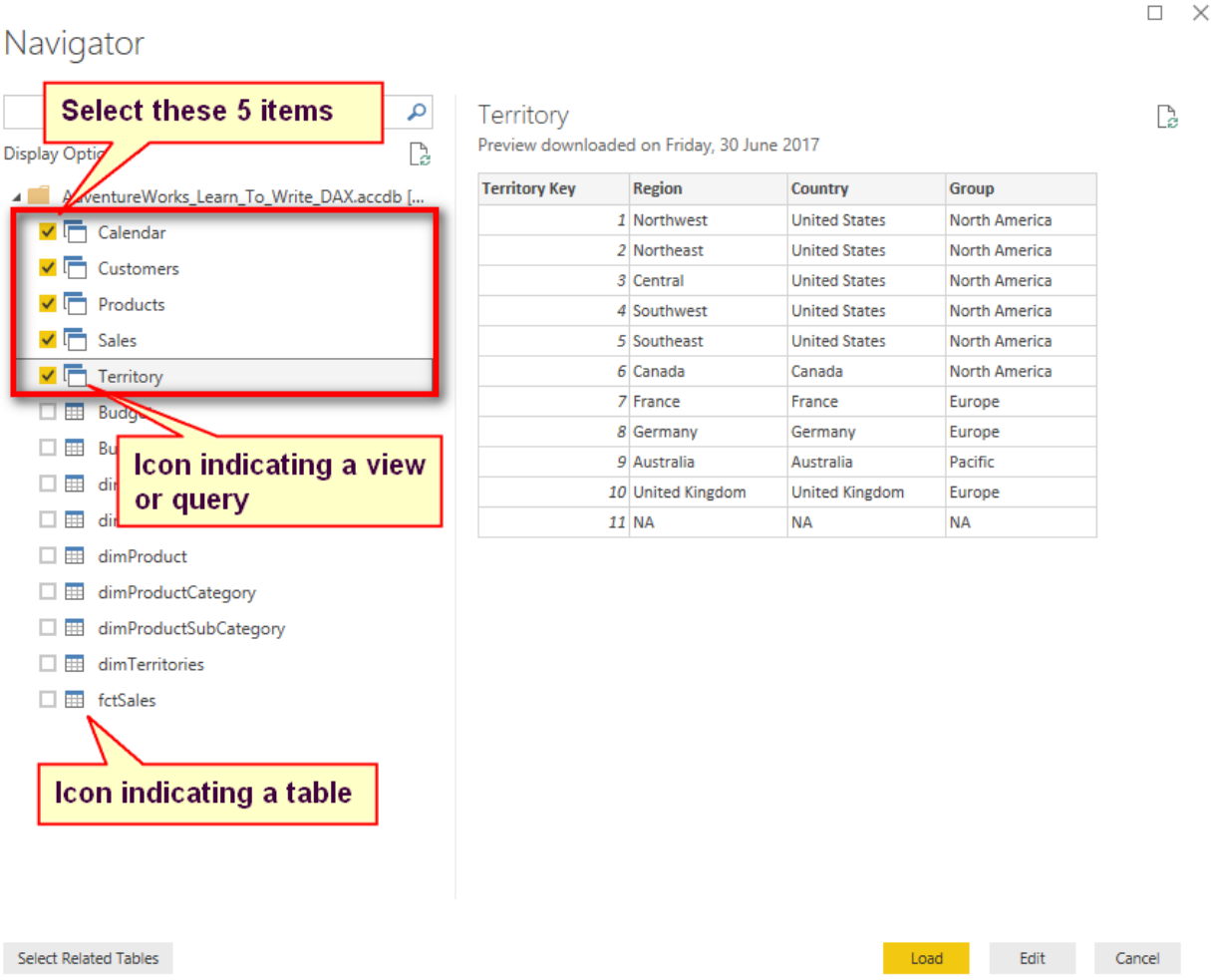
2. From the home menu in Power BI Desktop, select Get Data, All, Access Database (see #1 below) and then click Connect (see #2 below).



3. Browse to the location of the sample database you downloaded and unzipped earlier and click Open.

> **Note:** At this point, it is possible for things to go wrong, especially the first time you load data from Access. The most common cause of problems is that you have 32-bit Microsoft Office installed on your computer and 64-bit Power BI Desktop. In that case, you may see a message similar to this:
>
> ```
> The Microsoft.ACE.OLEDB.12.0 provider is not registered on the
> local machine. The 64-bit version of the Access Database Engine
> is required to connect to read this type of file.
> ```
>
> If this happens, I strongly recommend that you keep the 64-bit version of Power BI Desktop as you will need it to do any serious data processing with large data sets. You can solve the problem by installing the missing data provider. For full instructions on how to do this, read my blog article at http://xbi.com.au/3264.

4. Select the five views at the top of the list by placing a check mark in the box next to each one, as shown below. The Navigator pane shows different icons for queries/views and for tables as can be seen below.



Tip: The sample data in this book has been well prepared for learning how to use Power BI. You should not assume that *your* source database has the correct table structure for Power BI; in fact, it seldom will.

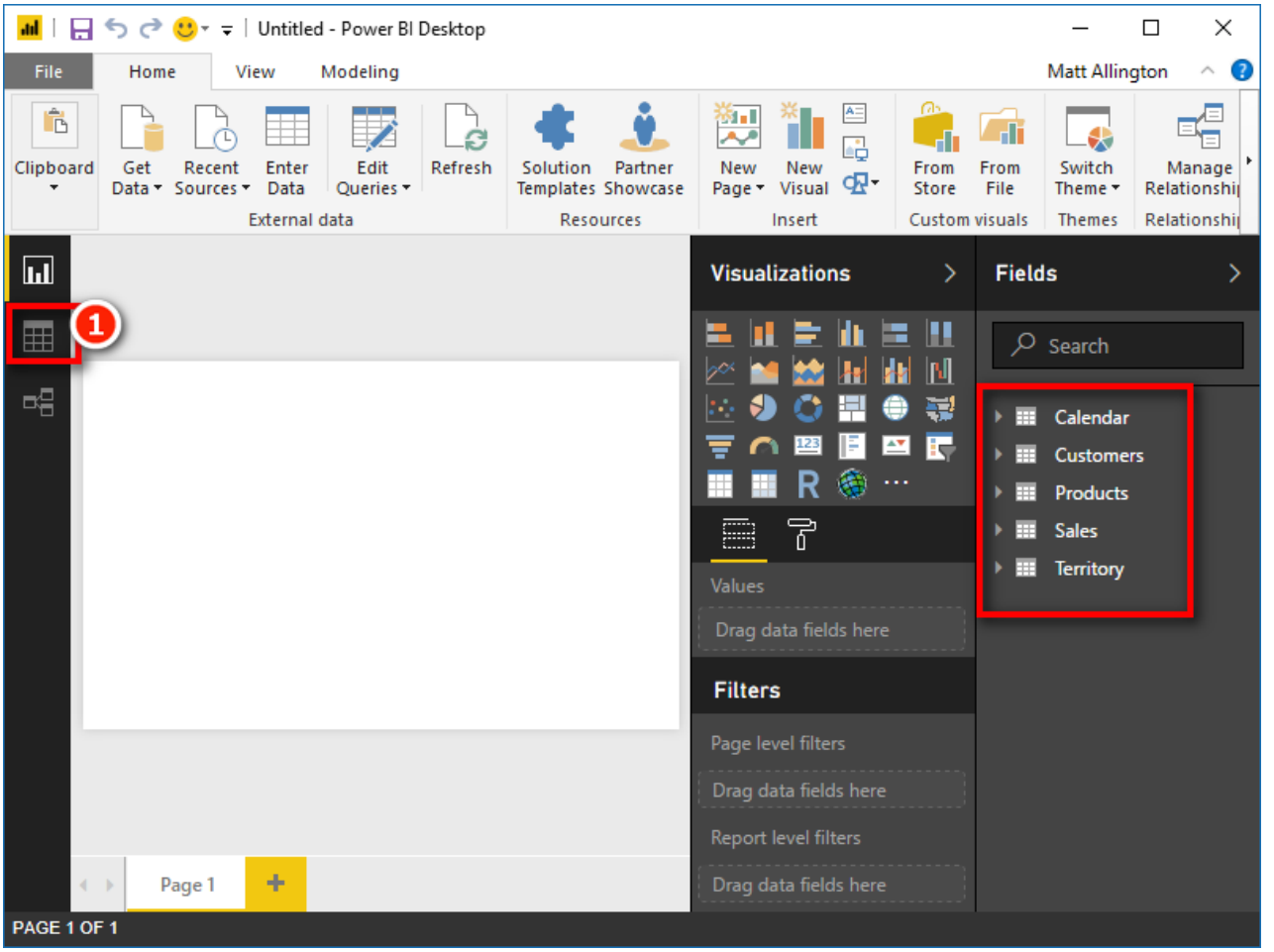
Note: If you click the Edit button now, you will launch into the Query Editor (Power Query), where you can transform the data prior to loading it into Power BI Desktop. Power Query is beyond the scope of this book, but I have a comprehensive online training course specifically designed to teach you how to use this powerful tool. You can learn more about that training course at http://xbi.com.au/powerquerytraining.

Also notice that the tables have names like dimProduct and fctSales, where dim indicates *dimension*, and fct indicates *fact*. It is very common for database tables to have prefixes like this. Business users can think of a dimension table as a lookup table and a fact table as a data (or transactions) table. The fact that there are two different types of tables—lookup tables and data tables—is a very important concept in Power BI, and you will learn a lot more about this as you work through this book.

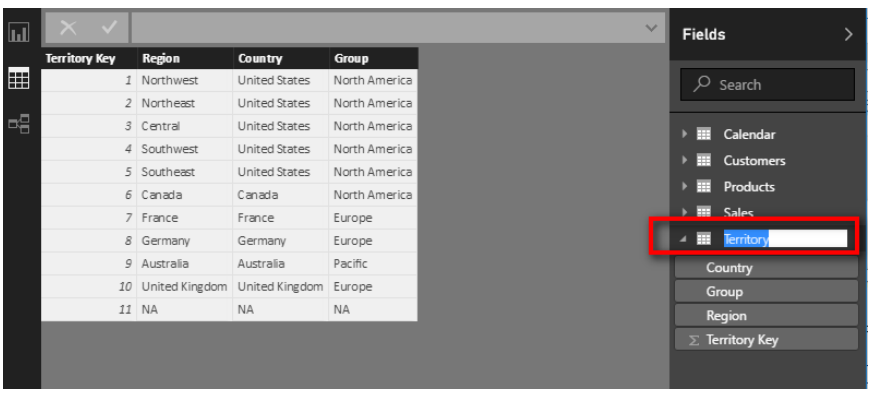
It is best practice for Power BI users to remove the dim and fct prefixes from the table names before importing these tables into Power BI. These prefixes have meaning to IT folk and help identify the type of table, but given that these table names will be visible to business users who use your Power BI reports, it is best to remove the prefixes after import by simply right-clicking a table and renaming it.

5. Click Load, and Power BI Desktop loads your data. After the Table Import Wizard is closed, you see the five tables you have just imported in Power BI on the right side, as shown below. Each of the tables is a complete copy of the data you imported from the source files (an Access database, in this

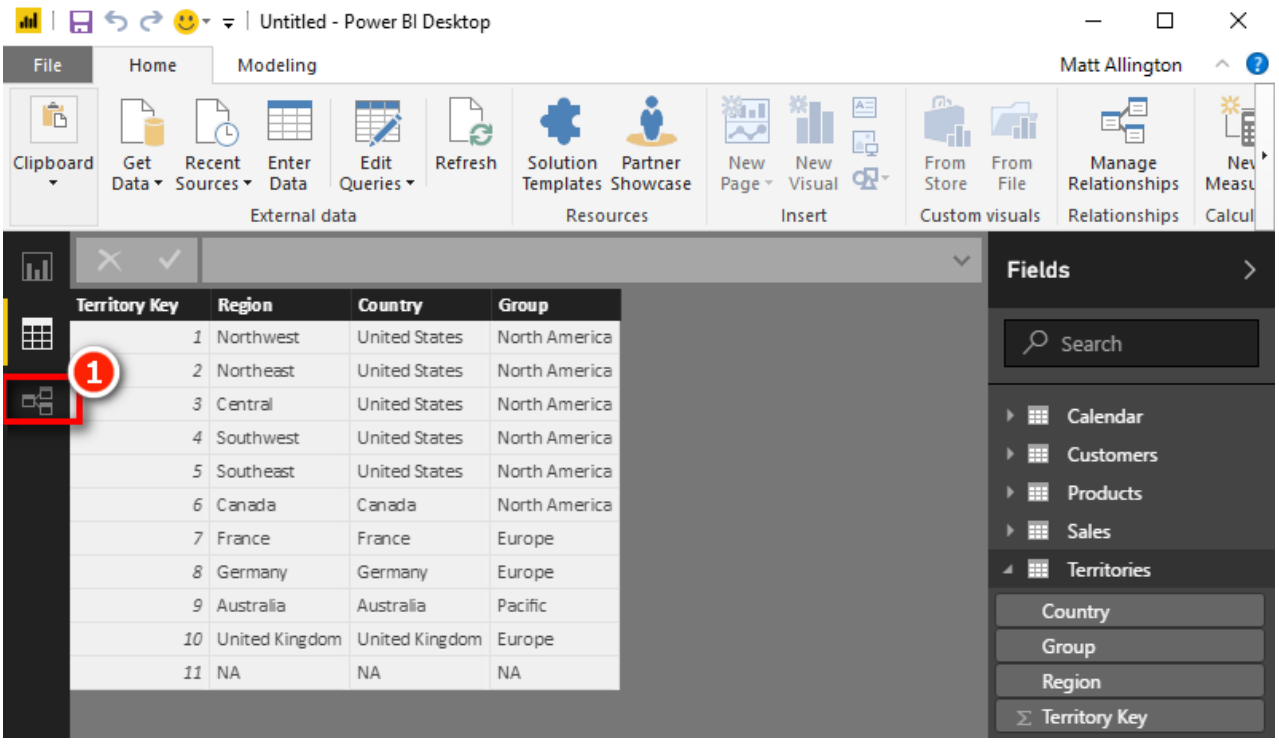
example). You don't need the source files again until you are ready to refresh the data—typically when the data changes at some time in the future. This is one of the many great things about Power BI: You can simply refresh the data when the data changes, and your workbooks are updated with the new data.



6. Now switch to Data view by selecting the Data icon (see #1 above). In this view, you can see the data in the tables.
7. In Data view, double-click the `Territory` table name on the right, as shown below, and rename it `Territories` for consistency (e.g., naming all tables with plurals—except `Calendar`, of course).

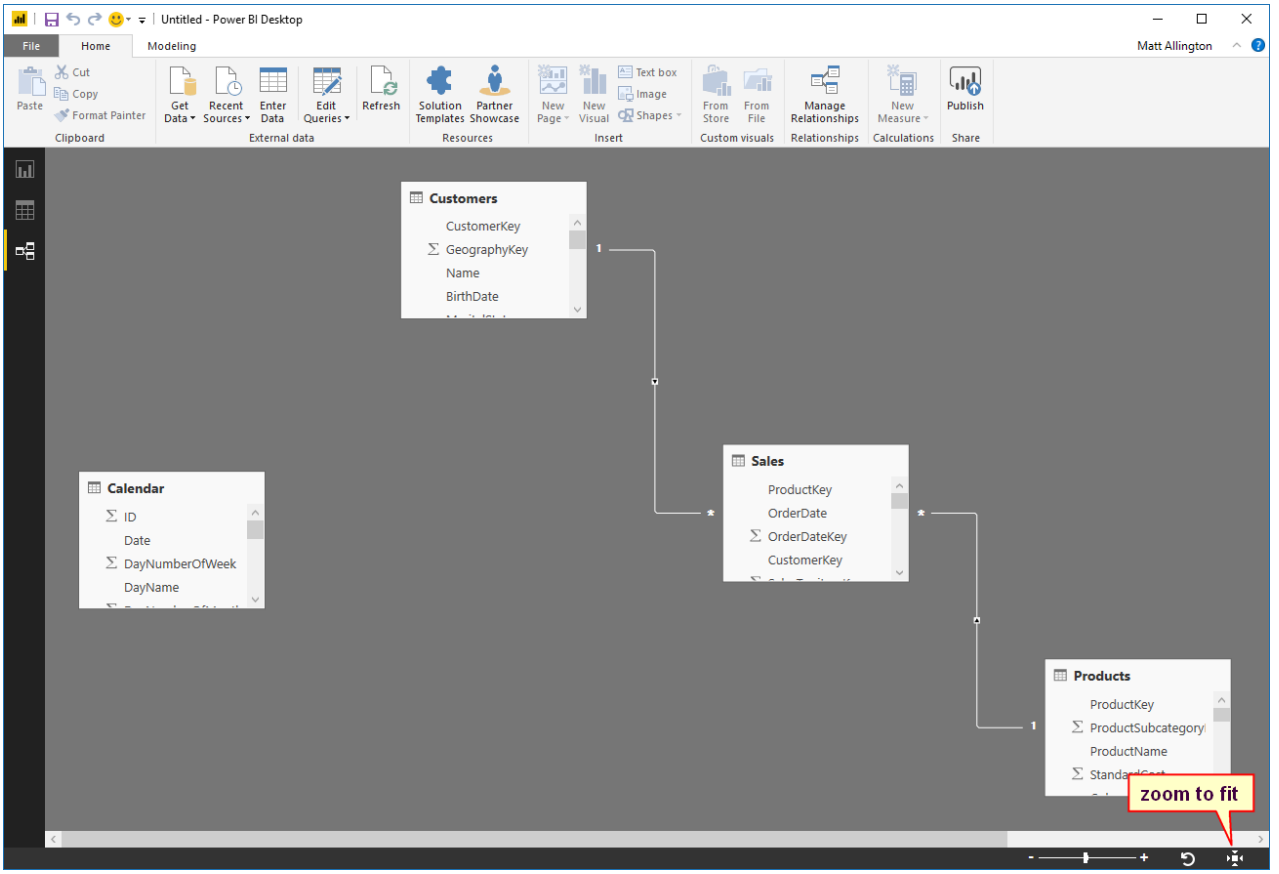


8. The next stage of the data modelling process involves creating the logical relationships between the tables. Switch to Relationships view by clicking the Relationship icon (see #1 below).



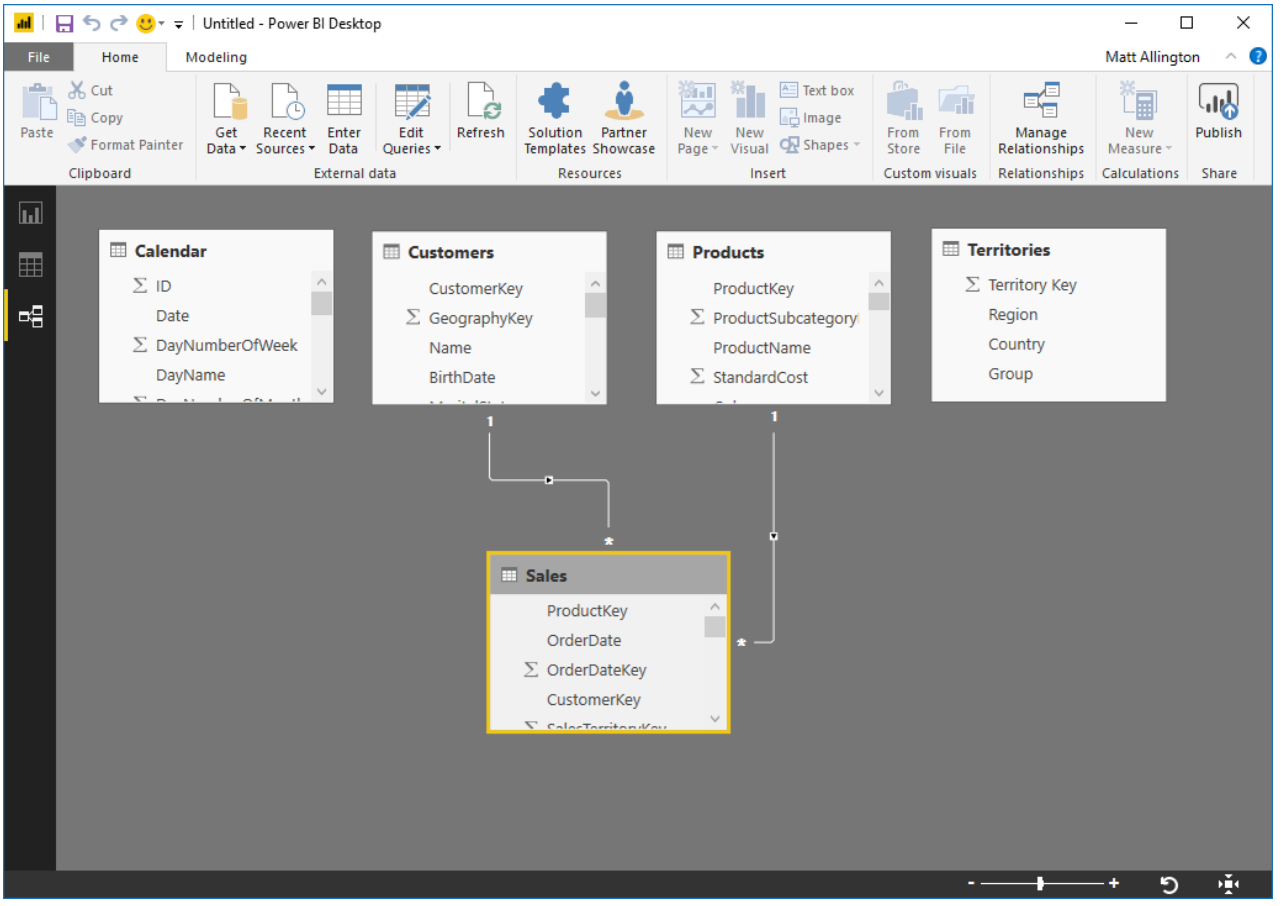
Territory Key	Region	Country	Group
1	Northwest	United States	North America
2	Northeast	United States	North America
3	Central	United States	North America
4	Southwest	United States	North America
5	Southeast	United States	North America
6	Canada	Canada	North America
7	France	France	Europe
8	Germany	Germany	Europe
9	Australia	Australia	Pacific
10	United Kingdom	United Kingdom	Europe
11	NA	NA	NA

9. If you can't see all five tables on the screen, click the Zoom to Fit button, shown below, to reveal the hidden tables.



In the image shown here, three of the tables have automatically been joined. This is Power BI guessing which relationships should be used. These automatically created relationships may or may not be correct; in this case, they are indeed correct.

10. Position your tables so that any data table or tables (there is only one in this case) are at the bottom of the screen and the lookup tables are at the top.



Once you've completed the preceding steps, you need to join the rest of the data table(s) to the lookup table(s), as described in the following "Here's How."

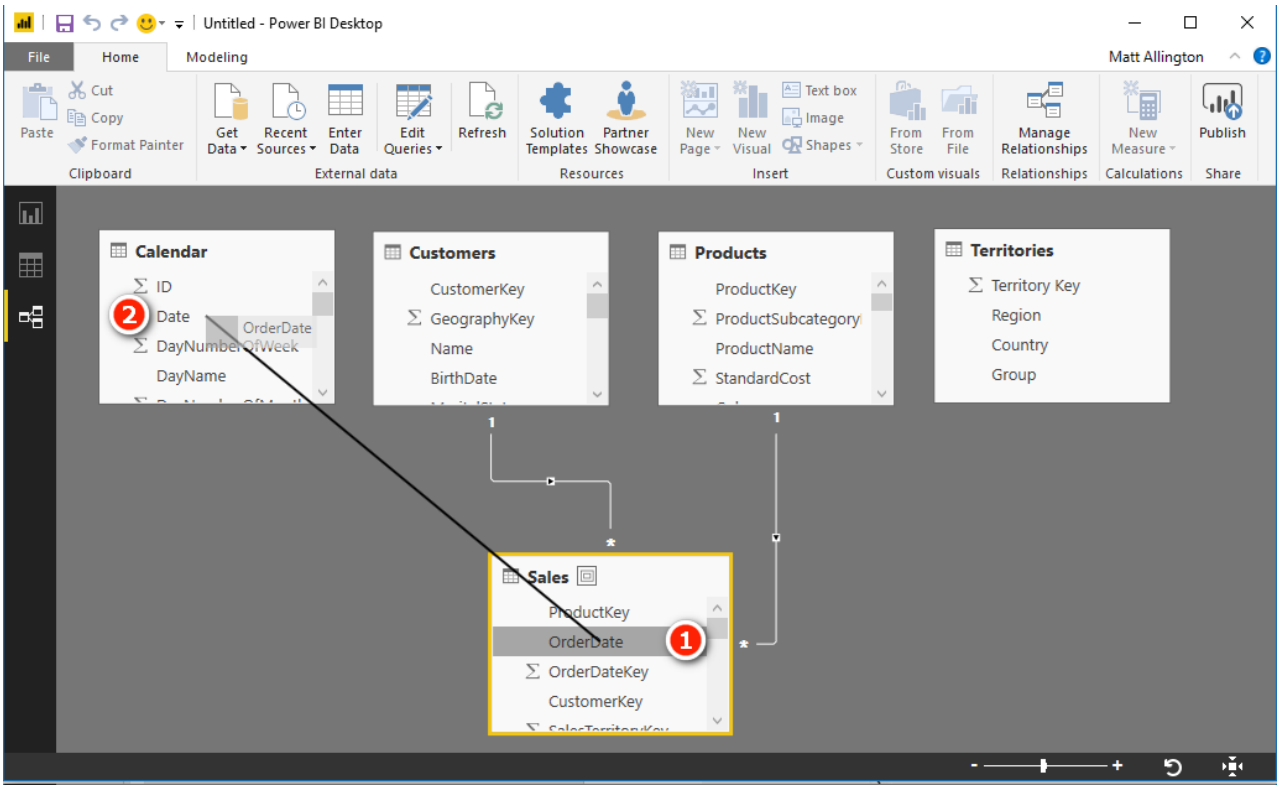
Here's How: Joining Tables in Power BI Desktop

A customer table typically has a list of all customers that a business has on file. But some of these customers may have never purchased anything from the company, some customers may have made only a single purchase, and some customers may have made many purchases. So for each entry in the `Customers` table, there is either none, one, or many rows in the `Sales` table.

The Sales table can be joined logically to the `Customers` table by using the customer key (often called customer number or ID). When these tables are joined on the customer key, there will be a one-to-many (`Customers`-to-`Sales`) relationship between these two tables.

To join a lookup table to a data table in Power BI Desktop, follow these steps:

1. Select a column from the data table (the table at the bottom of the Power BI Desktop screen, as shown below). To do this, click the `OrderDate` column in the `Sales` table and hold down the mouse button (see #1 below).
2. Drag the column up and hover over the matching key in the lookup table (in this case, the `Date` column in the `Calendar` table; see #2).
3. Release the mouse button to complete the join.

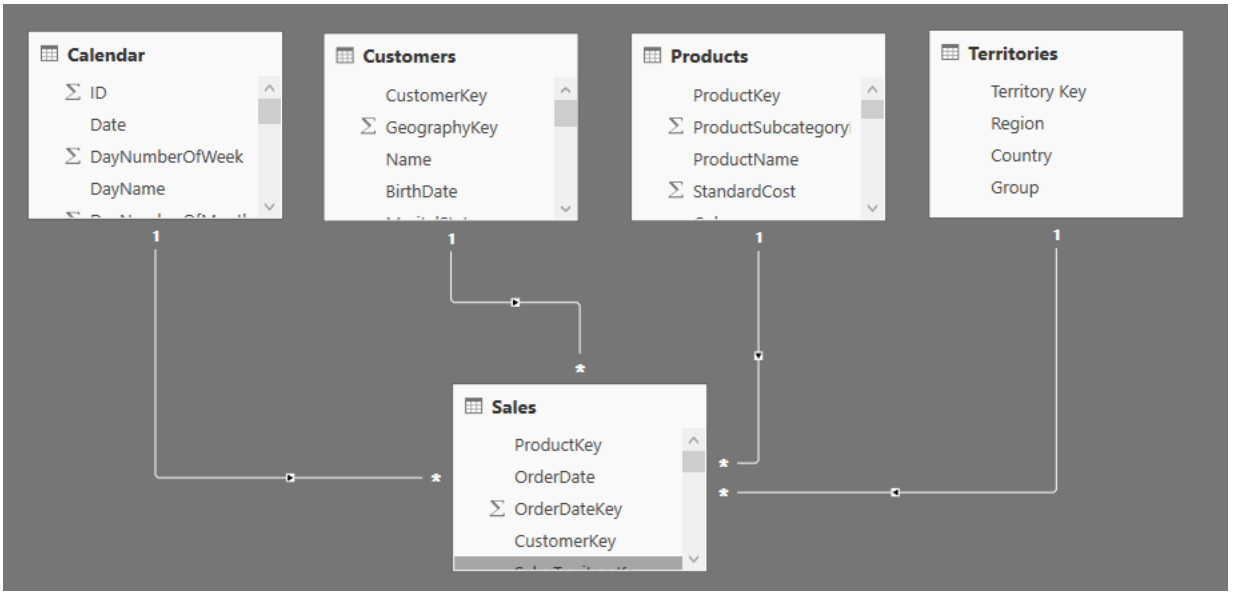


4. Complete the same process for the remaining tables. See if you can work out on your own which are the correct columns to join before you look at the answers below:

Data Table	Column	Lookup Table	Column
Sales	ProductKey	Products	ProductKey
Sales	CustomerKey	Customers	CustomerKey
Sales	SalesTerritoryKey	Territories	TerritoryKey

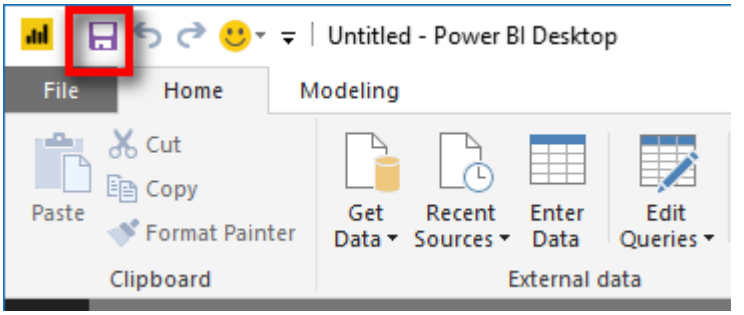
Because these relationships are one-to-many, the joins are specifically single-directional. Always drag from the data table up to the lookup table, not the other way around (as you would if you were writing a VLOOKUP in Excel).

As you can see in the image below, there is an asterisk at the end of the relationship that points to the data table, and there is a 1 at the end that points to the lookup table, and there is an arrow that points towards the sale table (more on those arrows later).



By putting the data table at the bottom, you get a visual clue that the tables at the top of the screen are lookup tables. (Get it? You have to "look up" to see the lookup tables.)

5. Save the Power BI Desktop file by clicking the Save icon, shown here, and specifying a suitable name and location.



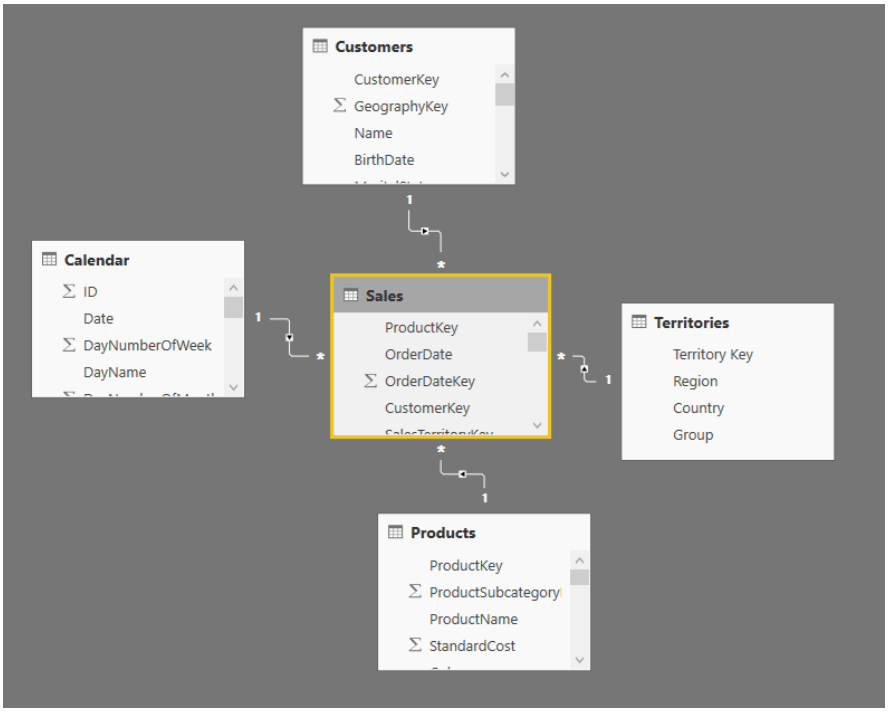
Shaping Data

It's time to pause for a minute to discuss the optimal shape of data for Power BI. When I say "shape" of data, I am talking about things like how many tables you import, how many columns are in each table, which columns are in each of the tables, etc.

Shaping data is a huge topic, and I don't have room here to discuss it fully. But I do want to give some foundational advice to get you started. One reason this advice is important is because the shape of data in transactional systems (or relational databases) is seldom the ideal shape for Power BI. When the IT department executes an enterprise BI project, one of the important first steps is to shape the data so it is optimal for reporting. This step is normally completely transparent to the end user (i.e., you), and hence the end user is shielded from the need to do this. But I am sharing this important information with you here and now because you need to understand data shaping if you want to have efficient and effective Power BI data models. Just copying what you have in your source data is unlikely to be optimal.

Choosing a Schema (Table Structure)

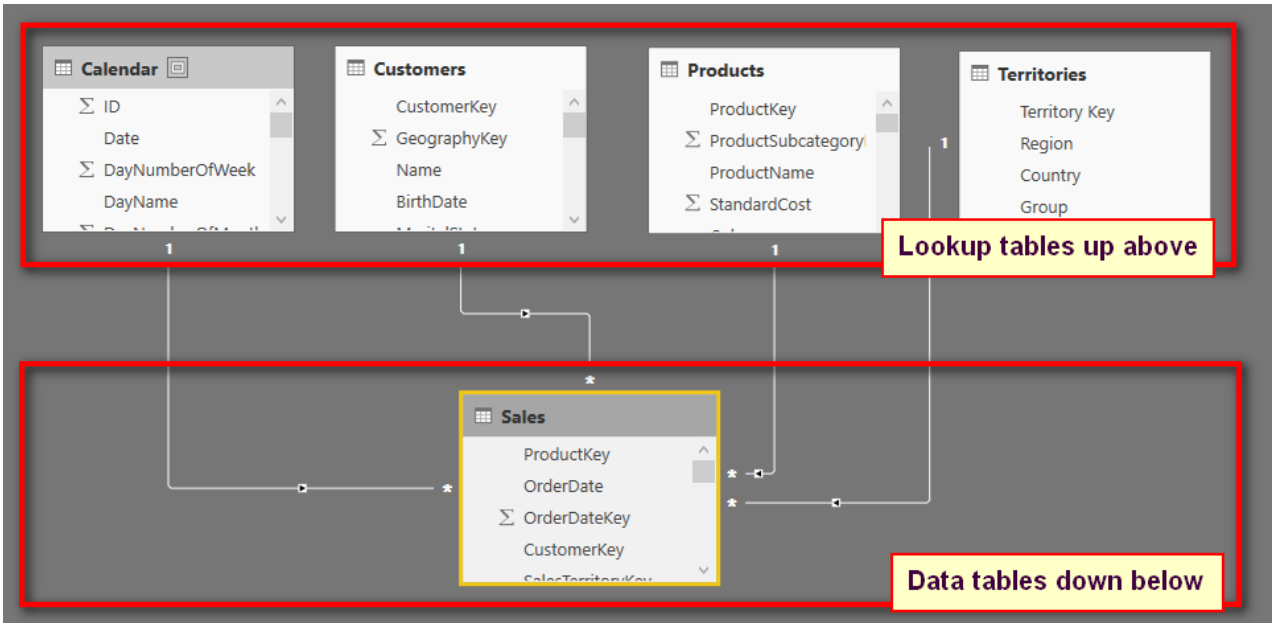
The generally accepted approach to bringing data into Power BI is to bring in the data in what's known as a *star schema*. This is a technical term that comes from the Kimball methodology (also known as *dimensional modelling*; you can look it up online) and describes the logical way data should be structured for optimal reporting performance. The objective of dimensional modelling is to allow the user to visualise the data without the need to write a new query over the database for each report. The visual layout of the tables in the following image (which includes exactly the same data you just imported) helps you see why it is called a star schema.



In this schema, data tables (only one, `Sales`, in this example) are surrounded by lookup tables (`Customers`, `Products`, `Territories`, and `Calendar` in this example), and together they visually make a star shape. You can find more comprehensive coverage of this topic in Chapter 6.

The Visual Layout of Tables in Relationships View

When it comes to visually positioning tables in Relationships view, I teach business users to position the tables such that the lookup tables are located at the top of the window and the data tables are at the bottom of the window (as shown below).



Note: There is no one correct way to shape your data, but the star schema is the recommended approach and should be used where possible. While the optimum data model shape is a star schema, other shapes work, too. For example, you can use a snowflake schema, with secondary lookup tables joined to the primary lookup tables; however, the extra relationships can come at the cost of degraded performance and possibly also confusion to users, particularly if they are building their own reports.

If you compare the last two images, you will see that they both have exactly the same logical relationship (links) between the tables: *They are both star schemas*, even though they have different visual layouts.

The visual layout in the second image, the one just above, is the one developed and recommended by Rob Collie, and I call it the "Collie layout methodology." The Collie layout methodology involves placing the lookup tables at the top of the window and the data tables at the bottom. The importance of this for business users learning Power BI will become evident later in the book. For now, just trust me and do follow the Collie layout methodology.

Understanding the Two Types of Tables: Lookup Tables and Data Tables

In the IT world, lookup tables are referred to as *dimension tables*, and data tables are called *fact tables*. For business users, though, I suggest using the terminology *lookup tables* and *data tables*.

A data table contains transactional information. In this book, the data table contains sales transactions. Lookup tables contain information about logical groups of objects, such as customers, products, time (`'Calendar'`), etc.

Before Power BI and Power Pivot, an Excel user needed to create one big flat table in Excel before creating a pivot table. Often that meant writing `VLOOKUP()` formulas to bring other data from other tables into the one allowed big flat table. It is no longer necessary to bring data from the lookup tables into the data tables using `VLOOKUP()`. Instead, you can simply load the lookup tables and join them with a relationship.

Lookup Tables

You should have one lookup table for each "object" that you need for reporting purposes. For example, in the data being used here, these objects are customers, products, territories, and time (i.e., calendar). A key feature of a lookup table is that it contains one and only one row for each individual item in the table, and it has as many columns as needed to describe the object.

So there is only one row for each unique customer in the `Customers` table. The `Customers` table has lots of columns describing each customer, such as customer number (key), customer name, customer address, etc., but there is only one row for each customer. Each row is unique, based on the customer number, and no duplicates of customer number (key) are allowed.

Data Tables

It is possible to have many data tables, but there is only one in this example: the `Sales` table. This data table contains lots of rows (60,000+ in this case) and all the transactional records of sales that occurred over several years. Importantly, the data table can be joined to each of the lookup tables. In this case the `Sales` table contains one column (technically called a *foreign key*) that matches each of the keys in each lookup table (technically called a *primary key*). Stated differently, the `Sales` data table has four foreign key columns, a date, a customer number, a product number, and a territory key. These columns allow the `Sales` data table to be logically joined to each of the lookup tables.

Ideally, data tables should have very few columns but as many rows as needed to bring in all the data records. Data tables normally have lots of rows (sometimes in the tens of millions).

The Shaping Bottom Line

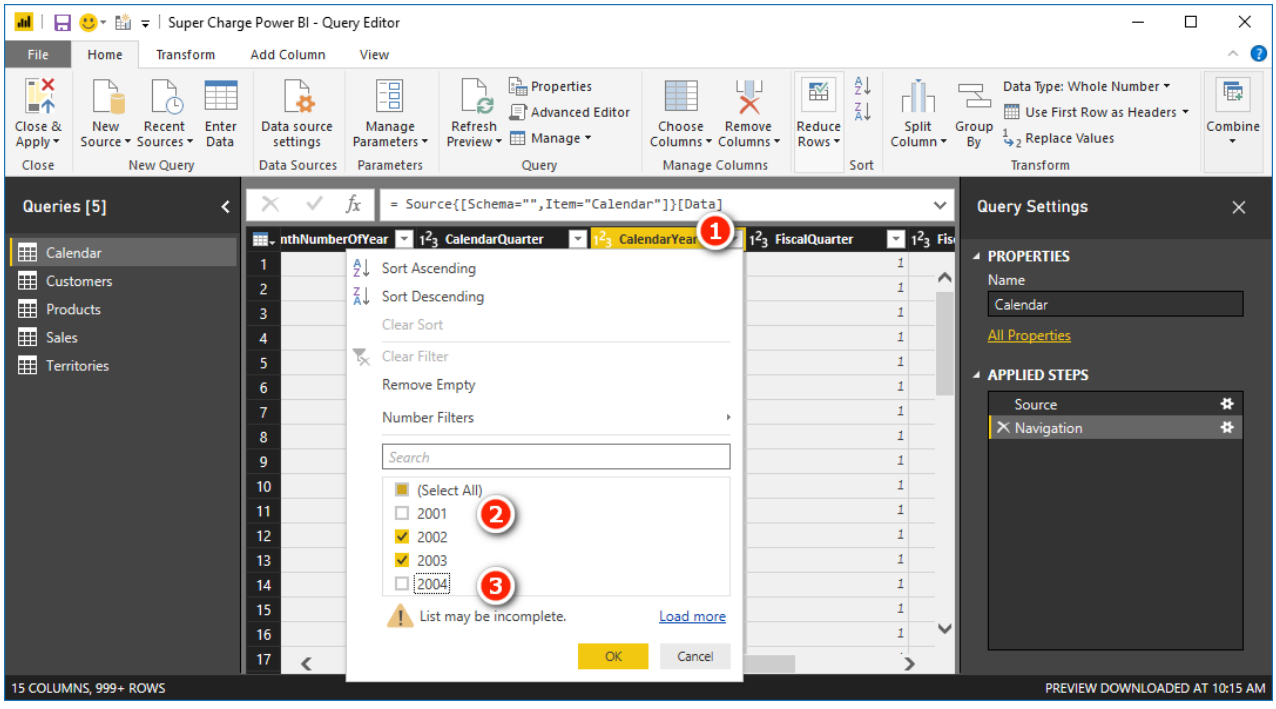
When it comes to shaping data, you need to remember the following:

- There are two types of tables: *data tables*, which contain the data you want to analyse, and *lookup tables*, which contain metadata about the objects you are going to analyse, such as the name, address, and city of each customer.
- The rule of thumb is to load one table for each object. This is both efficient for the database to process and easy for users to understand.
- The optimal way to shape your data is to use a star schema, but other schemas, such as a snowflake schema, can work, too, though they may be less efficient.
- For business users, it is best to position tables in Power BI Relationships view, using the Collie layout methodology. (You'll learn more about why you should do this in Chapter 5.)

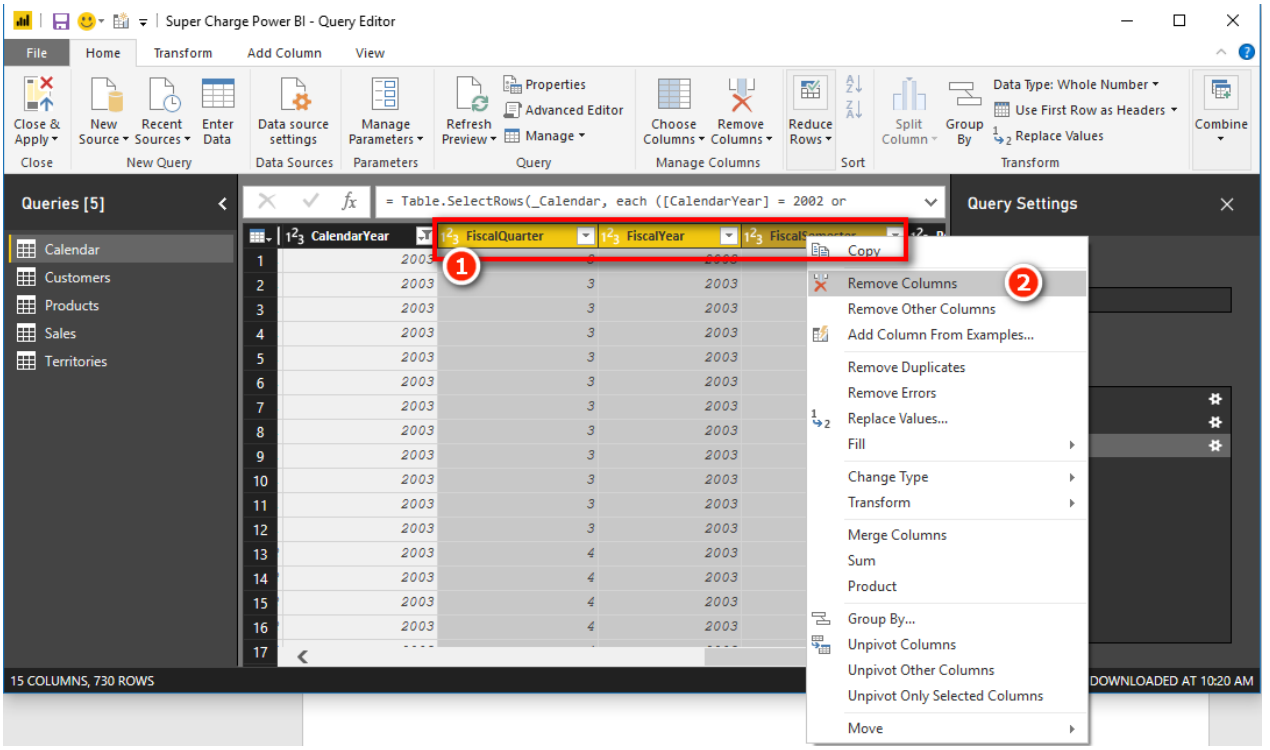
Here's How: Making Changes to a Table That Is Already Loaded

Say that you want to make changes to the `Calendar` table so that you only bring in dates for the years 2002 and 2003, and you also want to remove the fiscal date columns from the table. You can do this by using the Query Editor. The following steps walk you through how to make changes like these to a table that is already loaded:

1. In the fields list on the right (in Report view or Data view), right-click on the `Calendar` table and select Edit Query.
2. In the Query Editor that appears, navigate to the CalendarYear column and click the drop-down arrow (see #1 below).



3. Deselect the years 2001 and 2004 from the drop-down list (see #2 and #3 above) and then click OK.
4. Remove the three fiscal columns by first selecting them all using either the Shift or Ctrl keys (see #1 below) and then right-clicking on one of the selected columns and selecting Remove Columns (#2).



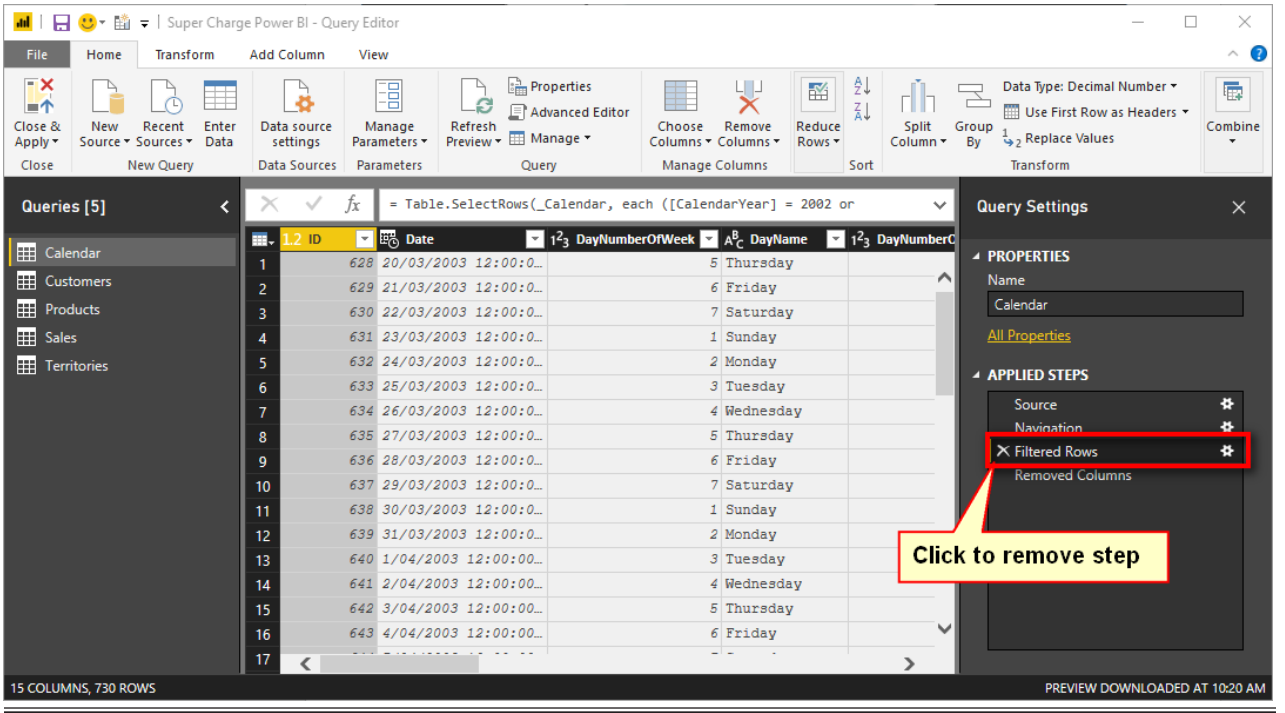
5. Click Close & Apply and then save the pbix workbook.

Note: When you are using the Query Editor as shown above, you are actually using Power Query technology (the Get and Transform menu in Excel and the Get Data menu in Power BI). Using Power Query is a big topic in its own right and is not covered in detail in this book. I recommend that you refer to my online training course, at http://xbi.com.au/powerquerytraining, to find out more.

Here's How: Deleting Steps in a Query

Now that you've seen how to make changes to a table that you have previously imported to the Power BI data model, you're ready to get some practice. Go back and clear the filters you applied on the `Calendar-Year` column because you need all the rows in the `Calendar` table for the practice exercises in this book. Clearing the filters is quite easy to do:

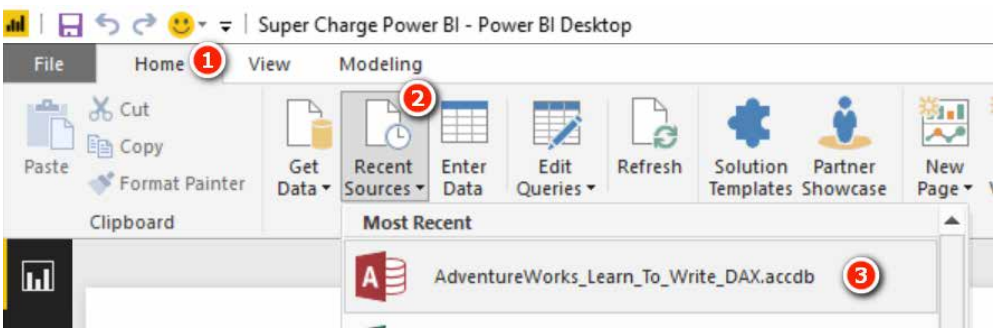
1. Edit the query for the `Calendar` table as before, by right-clicking the table and selecting Edit Query.
2. Click the X next to the step Filtered Rows, as shown in the image below, to remove the step.
3. Click Close & Apply and then save the pbix workbook.



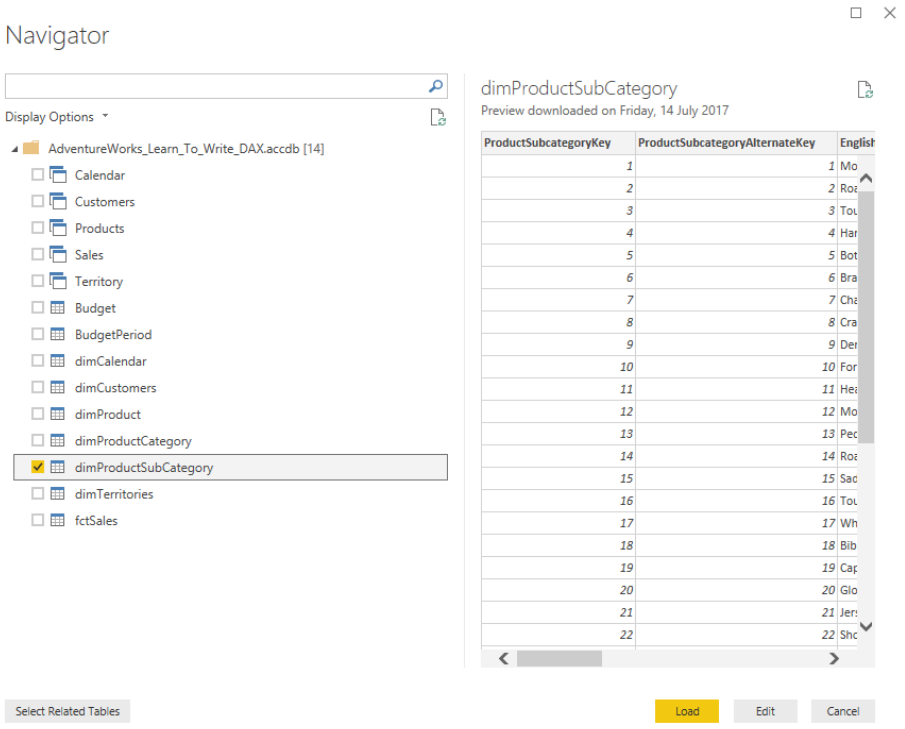
Here's How: Importing New Tables

In this exercise, you will use Get Data to bring in the `ProductSubCategory` table from the original Access database and join it to your data model. Follow these steps:

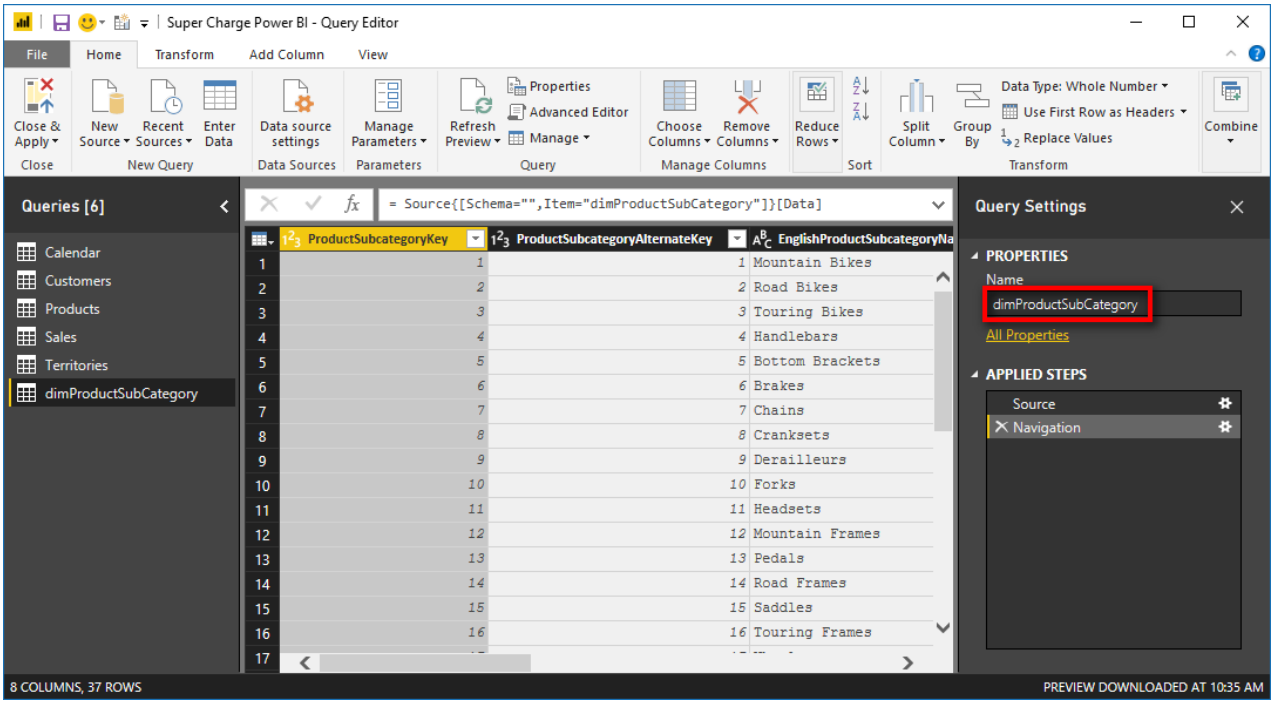
1. On the Home tab (see #1 below), click Recent Sources (#2) and then select the AdventureWorks Access database (#3).



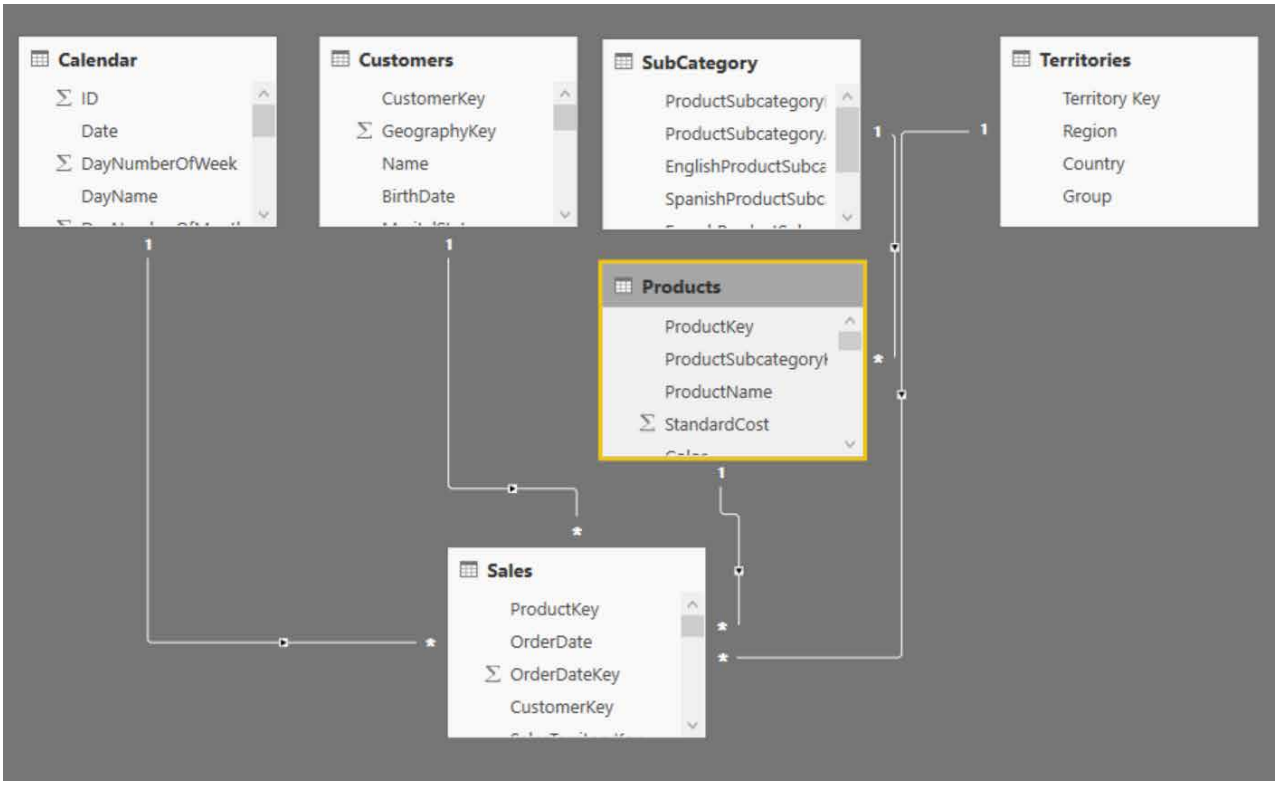
Select the `dimProductSubCategory` table, as shown below, and click Edit.



2. Remove the dimProduct prefix from the Name box on the right, as shown below, so you are left with SubCategory and then click Close & Apply. Power BI automatically connects the new SubCategory table to the existing Products table.



3. SubCategory is a lookup table of the Products table, so if you're using the Collie layout methodology, place the SubCategory table above the Products table, as shown below.



4. Save the pbix workbook.

Note: Now that there is a second lookup table (`SubCategory`) connected to another lookup table (`Products`), this is technically a snowflake schema. It will work, but it can be less efficient than a star schema. In addition, this shape can be confusing to users of the report because it does not follow the "one object, one table" rule; there are two tables that contain information about products. It is not wrong to do it this way. It is just a guideline to try to build models that follow the "one object, one table" rule where possible to keep things fast and easy to understand.

5. You won't need this SubCategory table again, so you should now delete it. Just right-click the table name in Report view or Data view and select Delete. The purpose of this exercise was simply to show you how to add new tables of data, when needed.

Here's How: Changing the File Location of an Existing Connection

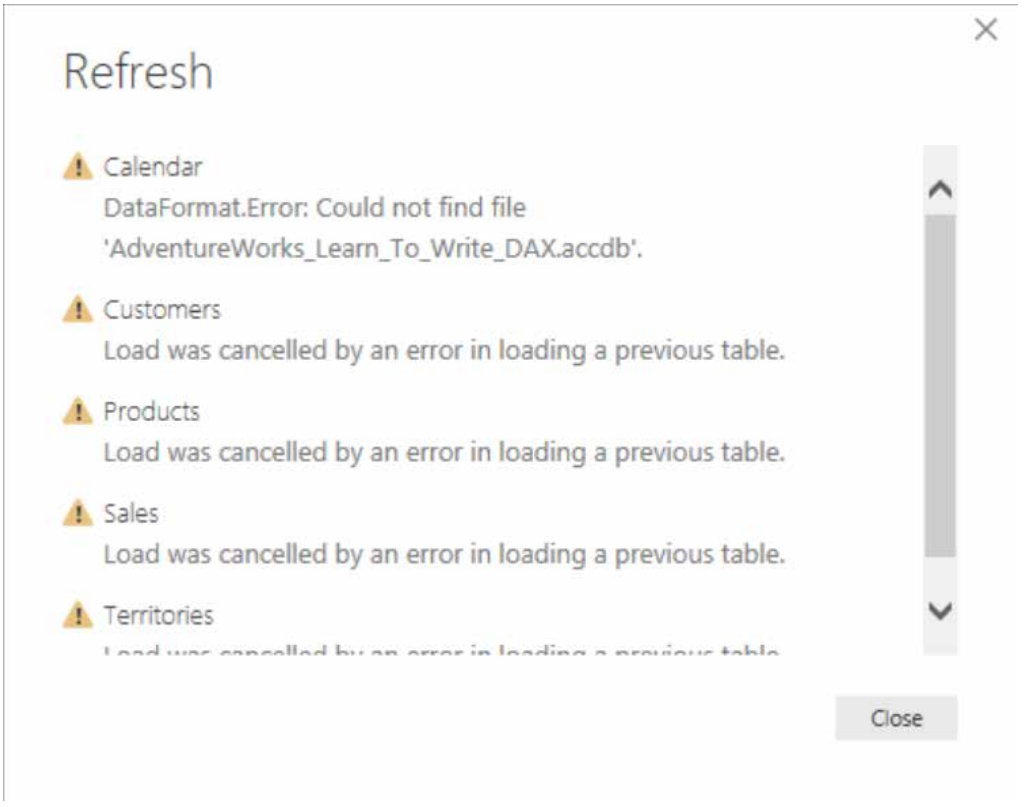
It's important to know how to move an Access database to a new location and then point the existing data connection to the new location. You need to do this, for example, if you ever send a Power BI workbook as well as the data source to another user or if you need to change your file locations on your own computer.

The data connections you create in Power BI are relative to your computer. When you send a Power BI workbook and data source to another user, that person will have to edit the data connection so that it will work on his or her own PC.

Note: You need to follow the steps in this section only if you send both a workbook and a data source to another user. But that is not normally what you do. Normally you just distribute a workbook and not the data source.

To simulate what can happen when a file location changes, the first thing you need to do in this case is to move the Access database to a new location so the existing query cannot find it. Then you can change the file location of an existing connection. Follow these steps:

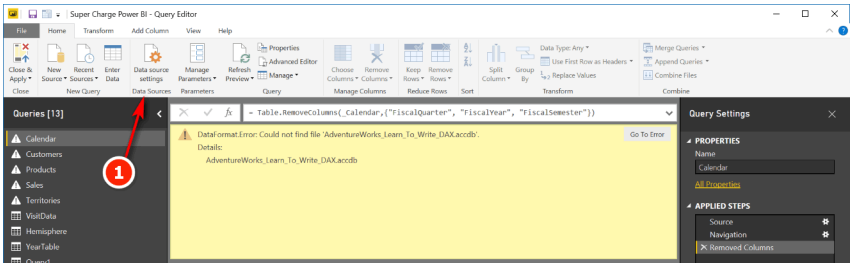
1. In Windows Explorer, create a new folder.
2. Navigate to your Access database and move it into the new folder.
3. Try to refresh your queries (by clicking Refresh on the ribbon) and note that it doesn't work, as shown below.



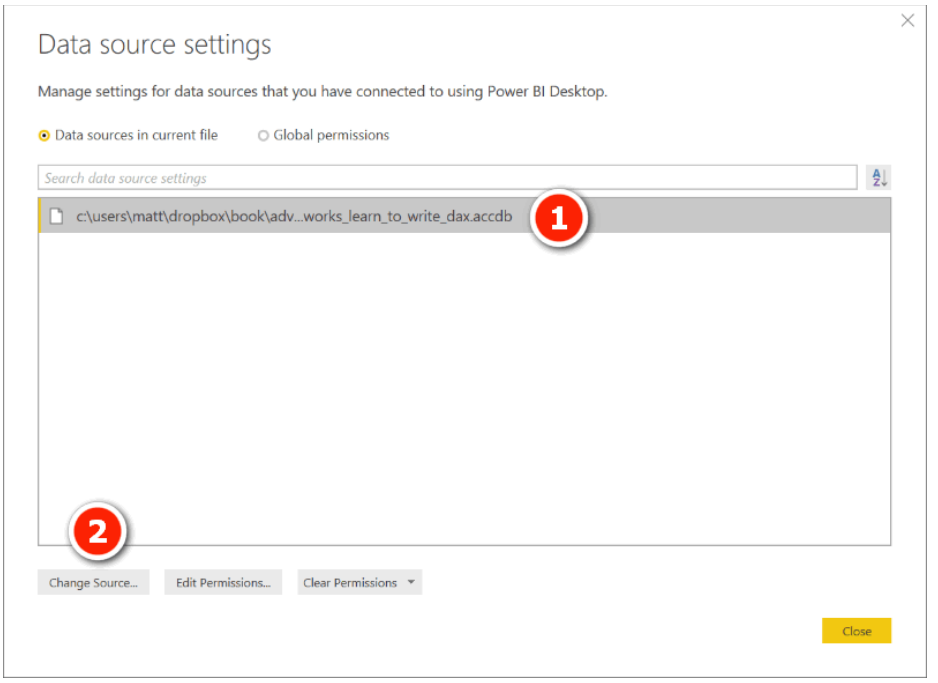
4. Click Close.
5. On the Home tab, click Edit Queries (or right-click on one of the tables in the fields list and click Edit Query there).

Note: The Query Editor keeps a cached copy of the data in the tables. When you first go into the Query Editor, the tables may seem fine. If you click Refresh Preview, Refresh All, the Query Editor will try to refresh the cache, and then you will see the resulting error messages.

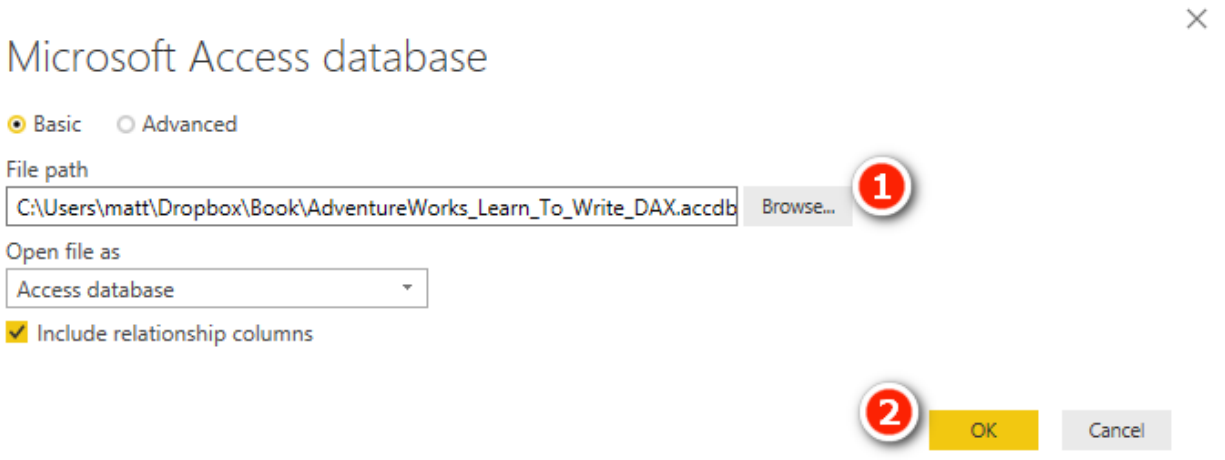
6. Click on Data Source Settings (see #1 below).



7. Select the data source that relates to the Access database (#1 below), then click Change Source (#2 below).



8. Locate the new file location using Browse (#1 below), then click OK when done.



9. Click Close & Apply and then save the pbix workbook.

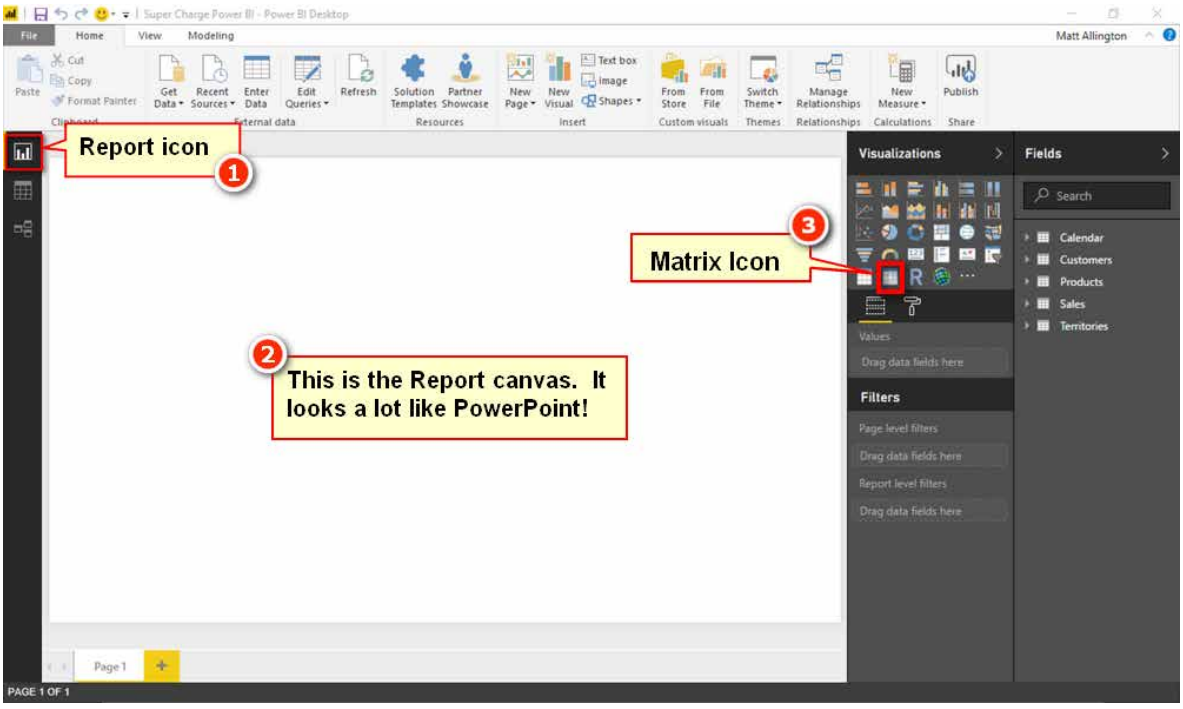
Here's How: Inserting a Matrix

The Power BI report canvas is very different to what you are used to using in Excel. The report canvas looks a lot more like a PowerPoint slide than like Excel. This can be quite confronting to Excel users who are getting started with Power BI. But the good news is that you will be comfortable with it in no time.

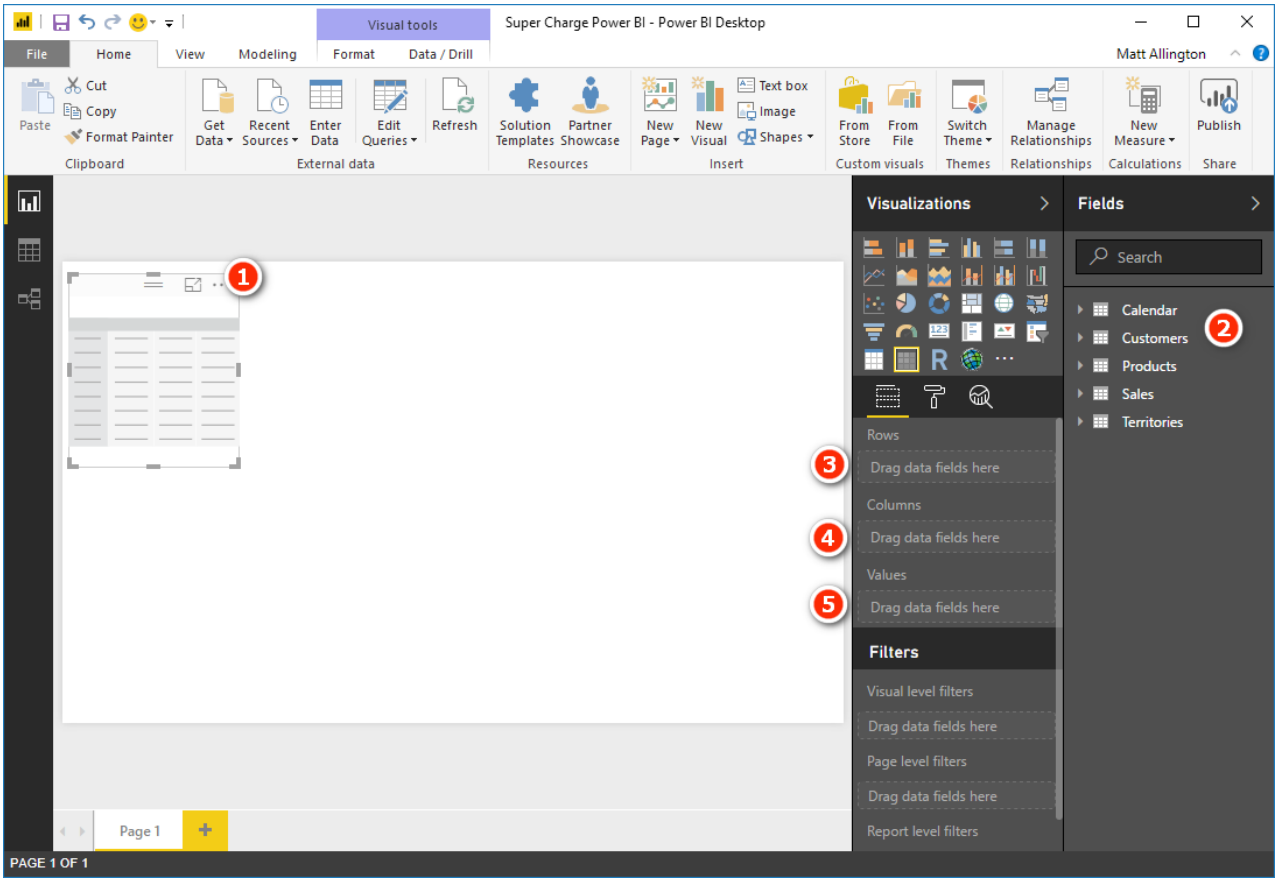
At this writing, there is no pivot table object in Power BI, but there is a matrix. A matrix is a very close substitute to a pivot table, and it is the best visualisation to use when you are starting out. There is also a table visual that you can use. The table is similar to a matrix, but it doesn't have an option to add a column. You can explore the difference yourself by switching between the two visualisation types in a report.

There are several ways to insert a matrix into a report. I suggest that you do it like this:

1. Open Report view by clicking the Report icon (see #1 below).



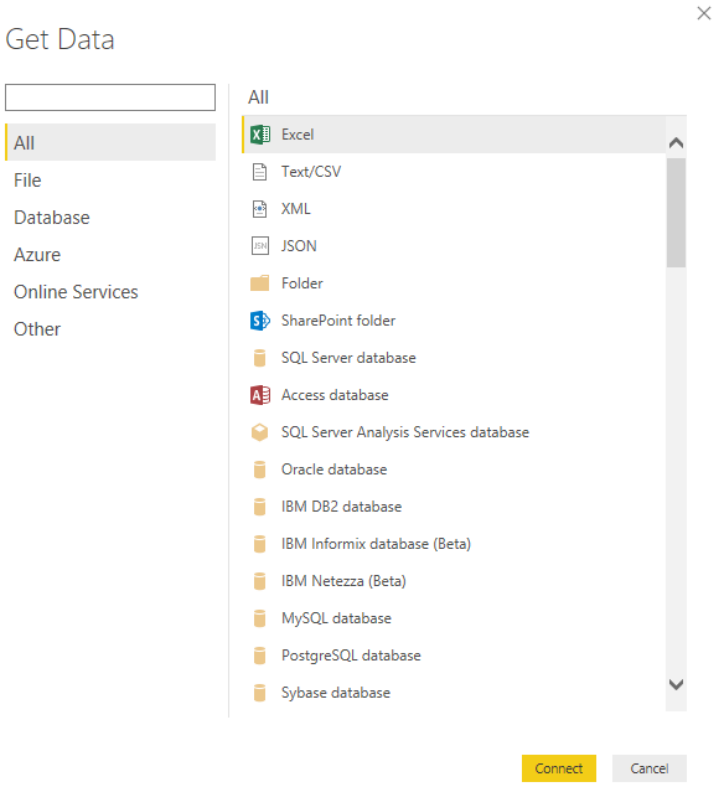
2. Click once on a blank section of the report canvas (#2 above) and then click the Matrix icon (#3).



3. You now see a new matrix shell appear on the canvas (see #1 above). Note the fields list (#2) and the Rows (#3), Columns (#4), and Values (#5) drop zones on the right-hand side. If you are experienced using pivot tables in Excel, you will recognise this as being very similar to the Excel pivot table experience.

Other Data Sources

In this book I teach you how to import data from the AdventureWorks Access database, but this is of course just one of the many data sources that you may need to access. There are many other data source connectors available in Power BI. To see a full list, simply click on Get Data, More, All. You can then see the full list of currently supported data connectors in Power BI, as shown below. Note that not all connectors are visible in this screenshot.



The general principle for importing data is the same for any data source you use: You simply select the appropriate data source and then follow the import wizard just as shown earlier in this chapter (see “Here’s How: Loading Data from a New Source”).

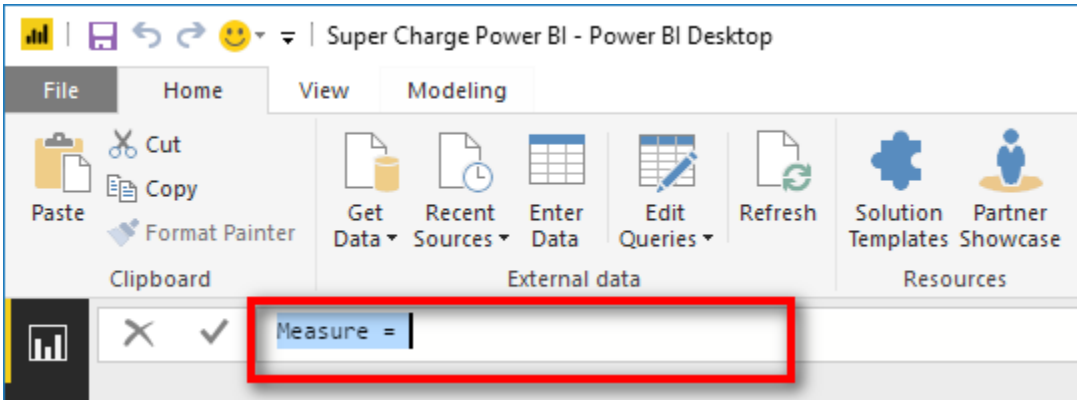
Note: The process of shaping and cleaning the data from its source prior to loading can require a significant amount of work, but that is a Power Query process and beyond the scope of this book. Refer to my online training at http://xbi.com.au/powerquerytraining for more information.

3: Concept: Measures

Measures have been around for many years in the enterprise versions of Microsoft BI tools, such as SQL Server Analysis Services. Measures have now made it into the world of business users who want to learn to create Power BI reports. There is nothing confusing or hard to learn about measures. A *measure* is simply a DAX formula that instructs Power BI to do a calculation on data. In a sense, a measure is a lot like a formula in a cell in Excel. The main difference, however, between a formula in a cell in Excel and a measure is that a measure always operates over the entire data model, not over just a few cells in a spreadsheet. You'll learn more about this later, but for now you can just think of a measure as a formula that calculates a result from the loaded data.

Techniques for Writing DAX Measures

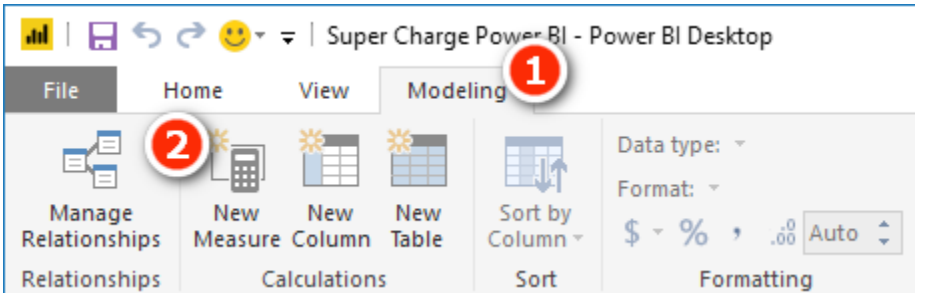
Measures in Power BI are always written in the formula bar, as shown below.



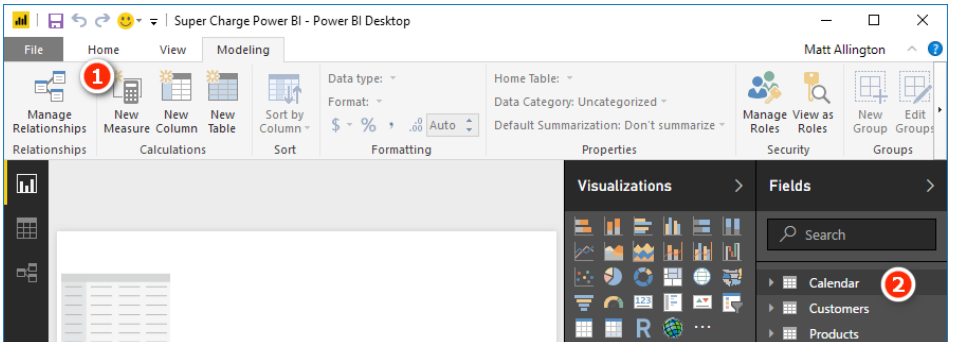
The formula bar is not visible unless you have a measure selected. When you select a measure, the formula bar appears just below the ribbon, as shown above.

There are two ways you can start the process of writing a new measure in Power BI:

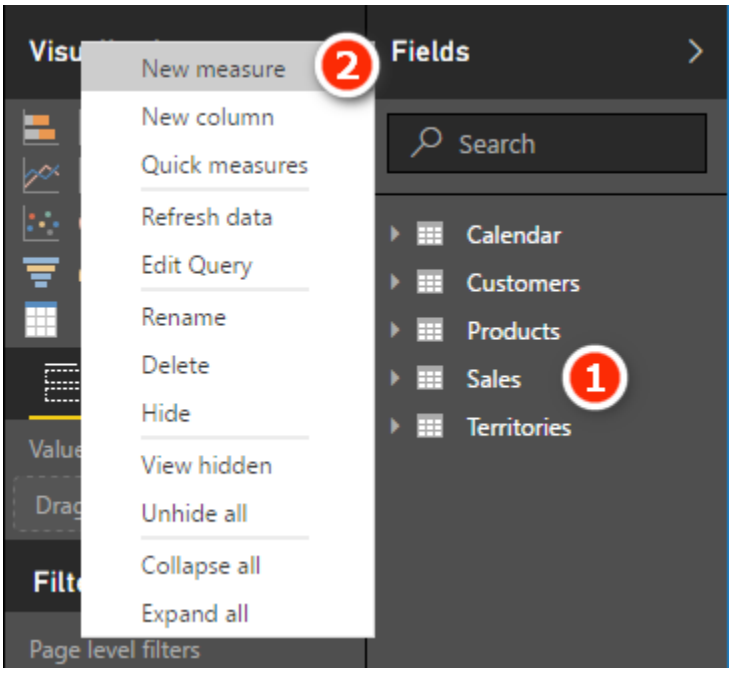
- First, you can select the Modeling tab (see #1 below) and then click the New Measure button (#2).



- I don't recommend that you use this approach to create a new measure as it has one major problem: Any time you create a new measure by clicking the New Measure button (see #1 below), the measure is automatically added to whichever table you have selected in the fields list on the right (#2). It is far too easy to place the measure in the wrong table when you use this approach. I therefore recommend that you use the approach described next.



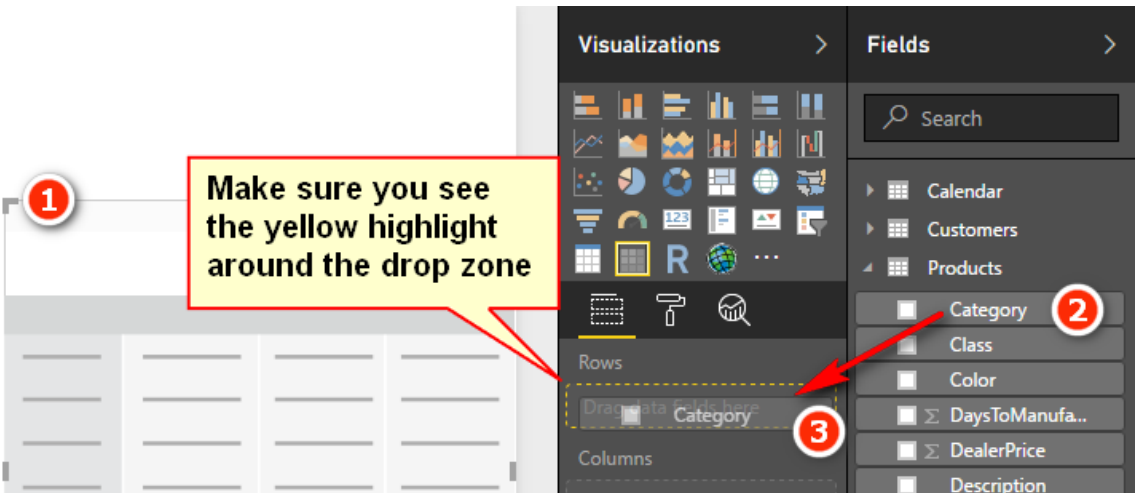
- The second way to write a new measure in Power BI is to right-click the table where you want to store the measure (see #1 below). (Best practice is to store the measure in the table where the data comes from. You'll learn more about this later.) Then select New Measure from the menu (#2) and write the measure from there.



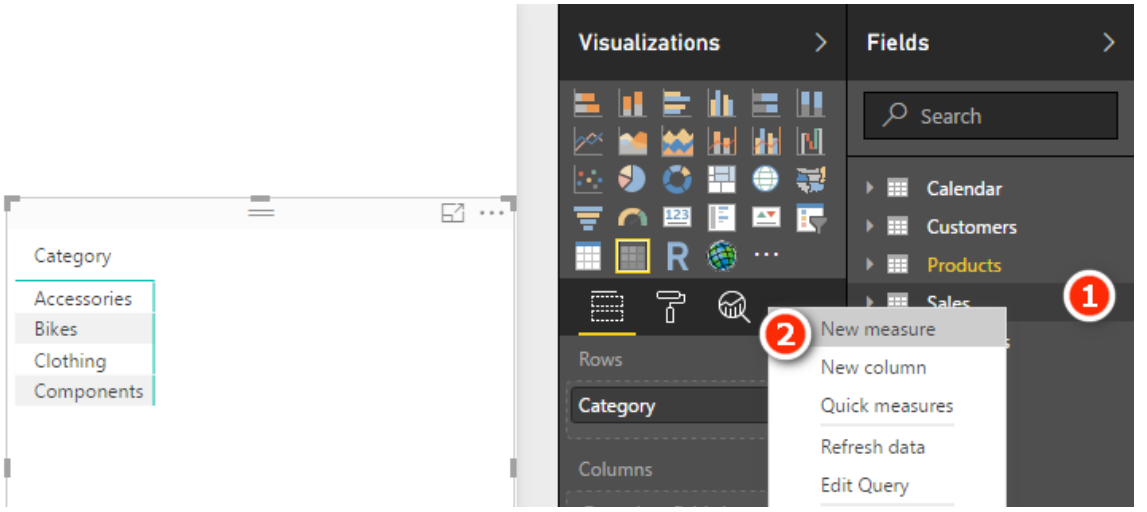
Here's How: Writing Measures

The approach to writing new measures described here is the best approach I have found to ensure that you get the best possible outcome with the least amount of rework. Follow these steps:

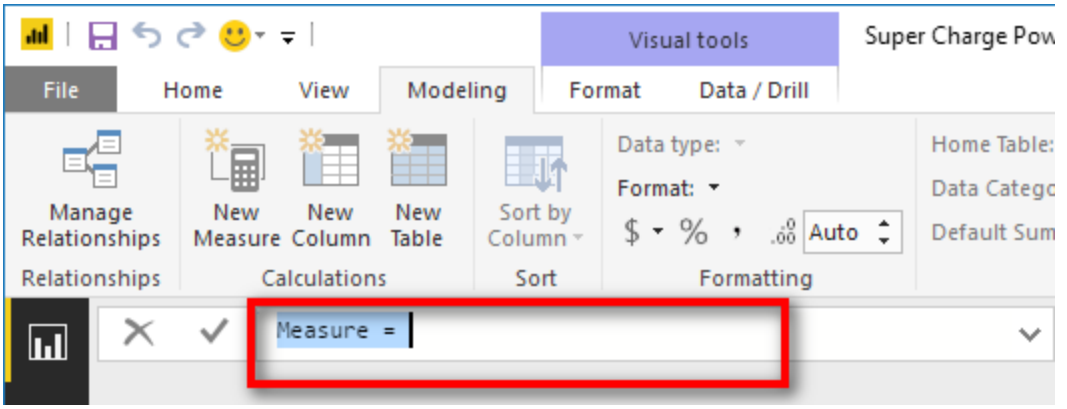
1. Create a new blank matrix (or use an existing one if you already have one set up from earlier). Make sure that you have the matrix selected on the report canvas. You can see the drag handles (see #1 below) when the matrix is selected.
2. Add some relevant data to the rows in your matrix. For the sample database used in this book, I suggest that you go to the `Products` table and place `Products[Category]` on Rows in the matrix. To do this, select the `Products[Category]` column (see #2 below) and then drag and drop the column into the Rows drop zone for the matrix (#3). Keep an eye out for the yellow dotted line around the drop zone. When you see the yellow dotted line, you can release the column, and it will be correctly placed in the matrix. You should always place measures in the table where the data comes from. In this case, you write the measure `[Total Sales]`, and the "data" you are using is in the `Sales[ExtendedAmount]` column, which is in the Sales table.



3. Right-click the Sales table (see #1 below) and select New Measure (#2).

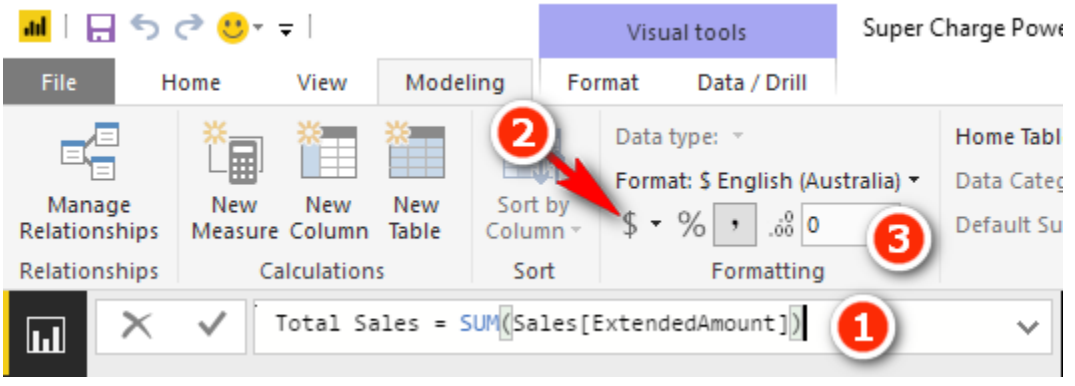


4. You now see `Measure =` appear in the formula bar at the top of the page, as shown below.

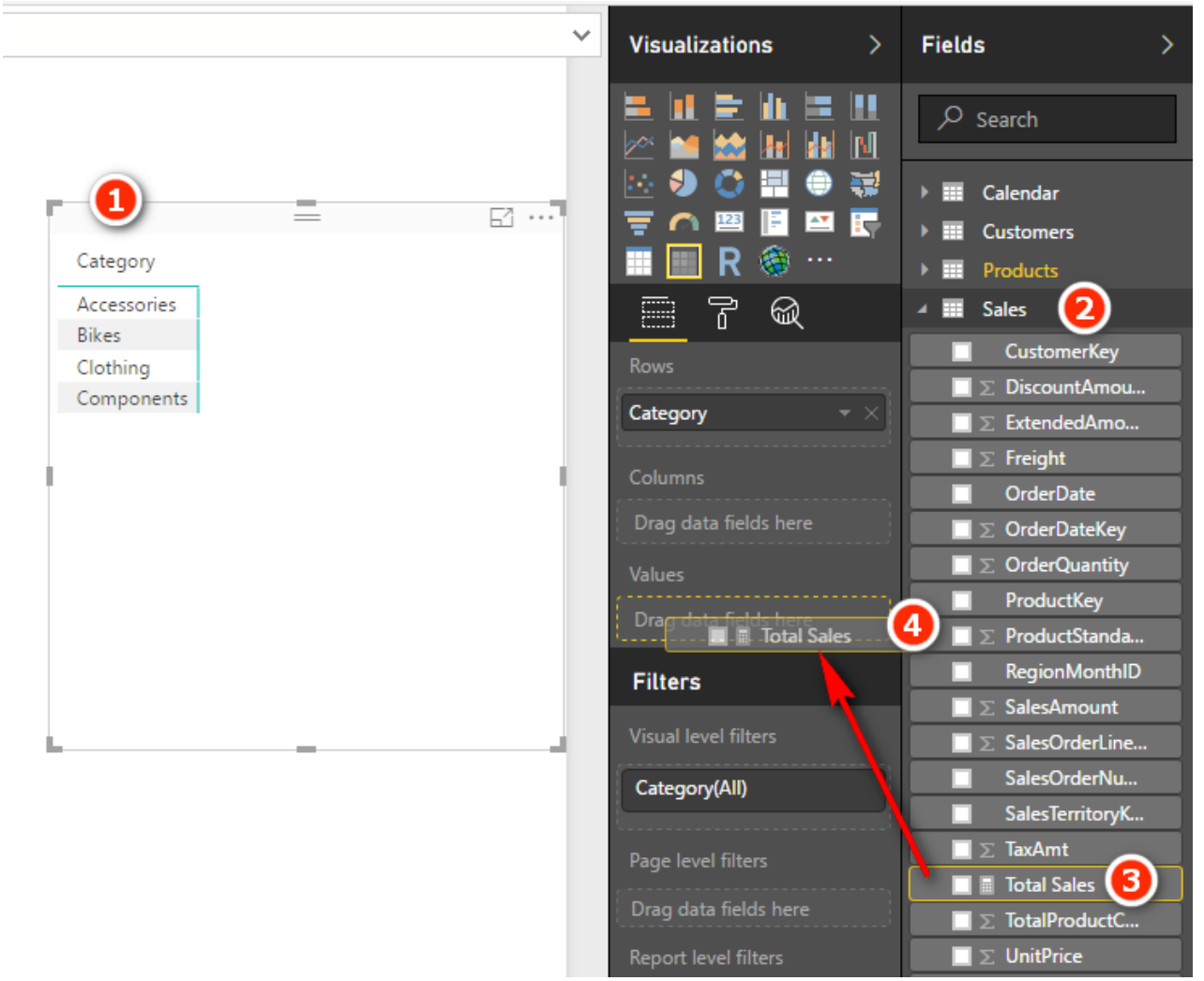


Note: The new measure has been given a default name, `Measure`, and an equals sign has been automatically added. Also note that the entire text above `Measure =` is highlighted in blue, indicating that it has been selected. The easiest way to proceed now is to simply type over the top of this text. Don't waste your time and effort trying to "save" the equals sign and trying to edit the word `Measure`. It is just not worth the effort. It is much faster to simply type over the top.

5. In the formula bar, type in the DAX formula `Total Sales = SUM(Sales[ExtendedAmount])` and press Enter. Type it directly over the highlighted text `Measure =`.
6. Immediately click back into the formula bar (see #1 below) and apply the formatting $ (#2), Currency General, and zero decimal places (#3) and then press Enter again. Get used to applying the formatting immediately so you don't forget.



7. Check to make sure you still have the matrix selected by ensuring that you can see the drag handles (see #1 below), navigate to the `Sales` table (#2), locate the new measure `[Total Sales]` (#3), and drag and drop the measure into the Values drop zone for the matrix (#4).



Tip: Following this procedure will save you time because you will not have to go back and fix things you missed. Practice doing it this way right from the start, and you will develop a good habit that will serve you well in the future.

Your matrix should now look something like the one shown on the left below. You may want to increase the font size of your matrix to make it easier to read, like the one on the right below.

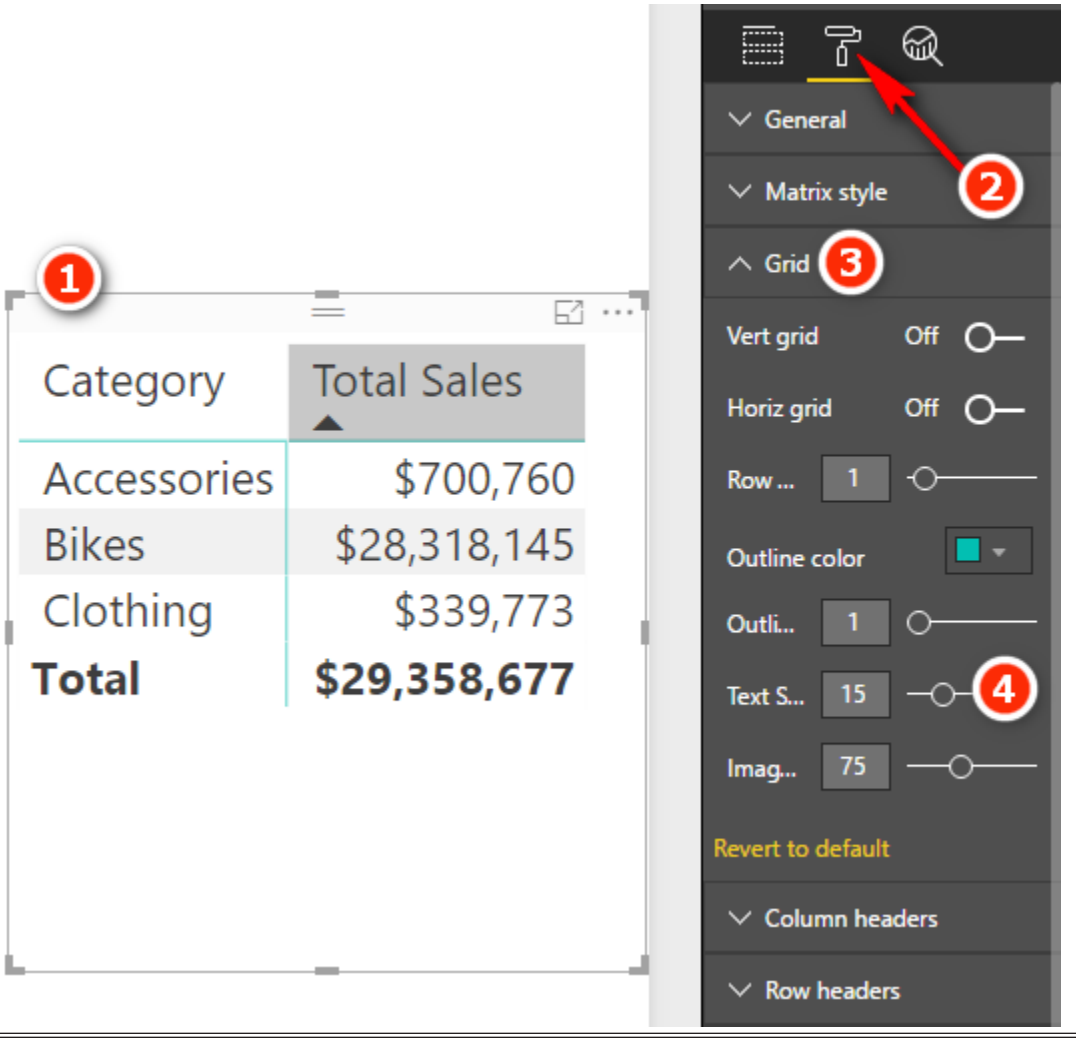
Category	Total Sales
Accessories	$700,760
Bikes	$28,318,145
Clothing	$339,773
Total	**$29,358,677**

Category	Total Sales
Accessories	$700,760
Bikes	$28,318,145
Clothing	$339,773
Total	**$29,358,677**

Here's How: Increasing Font Size

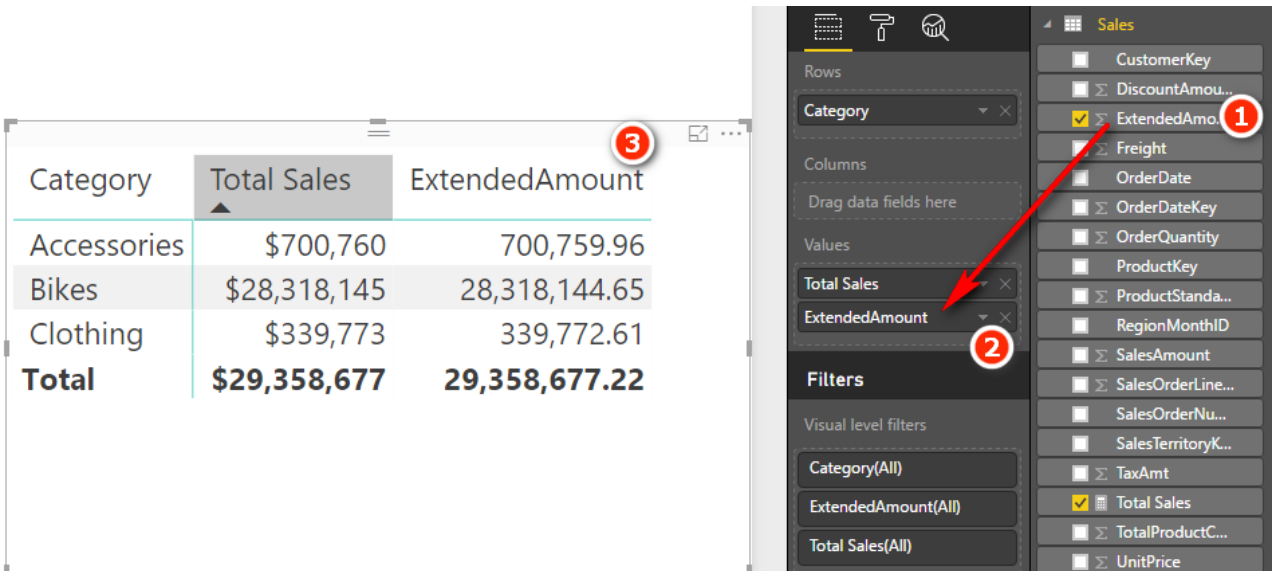
To increase the font size in a matrix, follow these steps:

1. Ensure that the matrix is still selected (see #1 below).
2. Navigate to the Format pane (#2).
3. Select Grid (#3).
4. Change the Text Size slider to an appropriate size or enter a new font size in the box (#4).

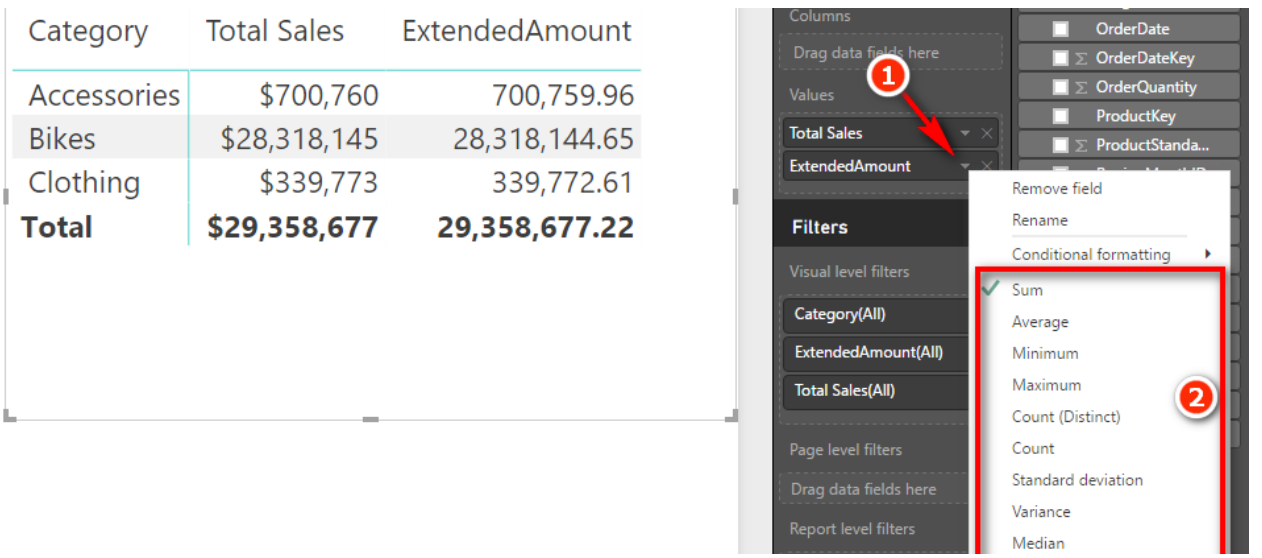


Avoiding Implicit Measures

You can add up the values in a column of numbers by dragging the column from a table (see #1 below) and dropping it in the Values drop zone for the matrix (#2), as shown below. You can then see the column in the matrix (#3). Note that you get the same answer this way as by using the measure `[Total Sales]`.



If you then click the drop-down arrow next to the column `[ExtendedAmount]` (see #1 below), you can see a range of options for changing the default aggregation behaviour (#2).



When you do this, you create what I call an *implicit measure* (although Kasper de Jonge tells me this is not the official name). This is not wrong, but personally I am not a fan of using implicit measures, and I recommend that you avoid using them—for a number of reasons:

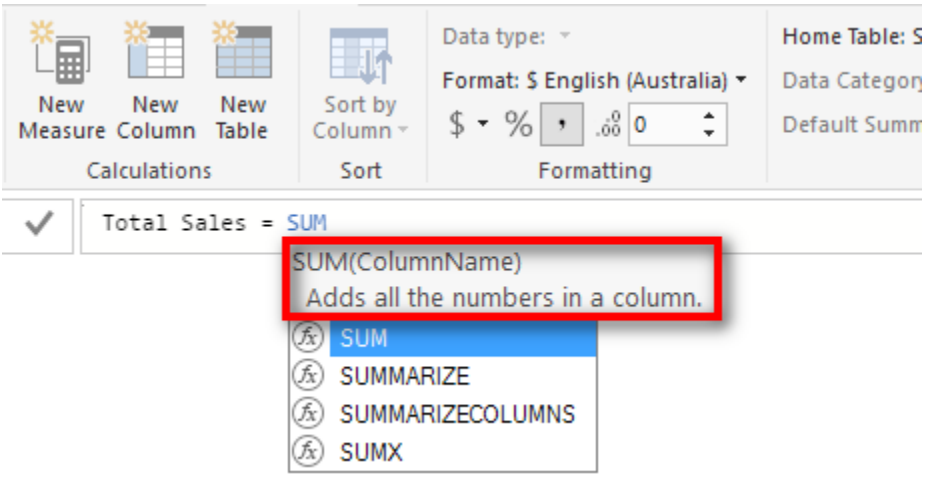
- The name of an implicit measure is not very helpful. Compare the name of this implicit measure, `[ExtendedAmount]`, with the name `[Total Sales]` that you provided when you explicitly wrote the measure yourself. You can change the name of an implicit measure, but the name changes only for the current visual (a matrix in this case). If you later add the same column again in a different visual, you have to change the name again.
- No formatting is applied when you drag to create an implicit measure. Again, you can add the formatting, but you will have to do it each time you create an implicit measure.
- You can't reference implicit measures inside other measures, so they have limited use.
- You won't learn how to write good DAX if you always use implicit measures.

So do yourself a favour and don't drag and drop your table columns. Of course, if you just want a quick look at a field for some testing, then doing this is fine. But undo the change immediately after you have taken a look. If you want to keep a measure, you should write it from scratch, using your DAX skills. It will be your skill in writing DAX that will set you apart from other users of Power BI.

Here's How: Using IntelliSense

When you type in a DAX formula in the formula bar, I recommend that you learn to leverage the IntelliSense tooltips that appear. Follow these steps to see how it works:

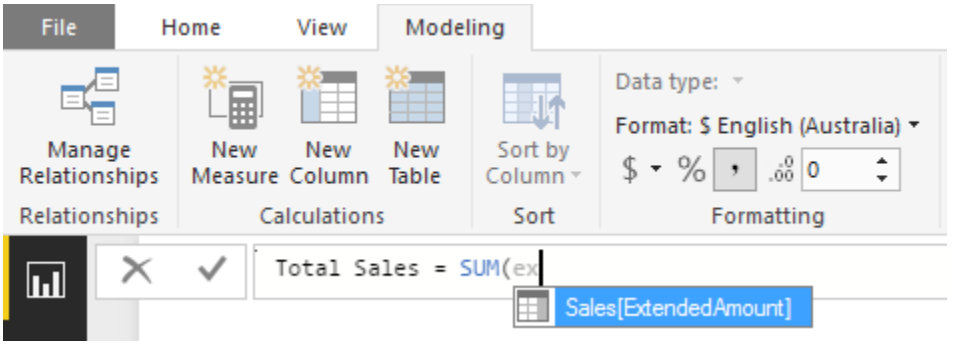
1. Type a function into the formula bar, as shown below, and you see the IntelliSense pop up to show you the syntax of the function (on the first line) and also how the function works (on the second line).



Tip: IntelliSense is your friend. Reading the information it provides will help you build your DAX knowledge and skills.

2. It is best practice in DAX to always type the table name before the column name. Power BI has a very good user interface that will help you do this. If you start typing the name of any column in your data model, as shown in the example below, where I've typed `ex`, IntelliSense prompts you with the full name *TableName*[*ColumnName*], as you can see here.

Tip: Always, always, always include the table name before the name of a column in your formulas.



3. Simply use the up and down arrow keys to highlight the column you want from the list presented by IntelliSense and then press Tab to select the column highlighted in blue. The table name is then included automatically for you.
4. Finally, type) (a closing parenthesis) and press Enter.

Tip: Try to use the keyboard and not the mouse to select from the tooltips, particularly if the list is short. This method may be slower for you to start with, but it will be faster in the long run if you learn to do it this way.

Here's How: Editing Measures

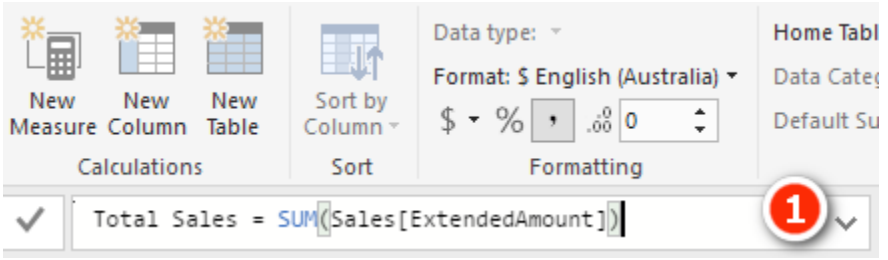
It is easy to go back and edit (or simply review) measures after you have written them. Follow these steps:

1. Find the measure you want to edit from the fields list on the right-hand side and click the measure once to select it. The formula bar reappears at the top of the page.
2. Click in the formula bar and edit the measure as required.

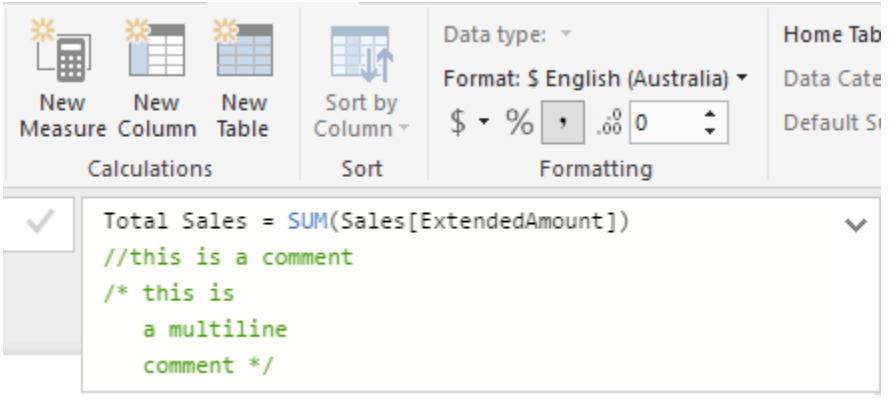
Here's How: Adding Comments to Measures

Power BI allows you to add notes and comments inside the measures you write. Follow these steps:

1. Select a measure from the fields list. It should then appear in the formula bar at the top of the page.
2. Expand the size of the formula bar by clicking the down arrow if needed (see #1 below).



3. Start a new line in the measure by pressing Shift+Enter.
4. Add comments as shown below. Use a double slash (//) at the start of a single-line comment and use the /* */ pattern to create a multi-line comment, as shown below.



When Something Goes Wrong as You Write DAX

At some point, you will start the process of creating a new measure, and something will go wrong. For one reason or another, you will need to stop what you are doing and go and do something else. In cases where you are partway through writing a formula but it is not finished, you can use the comments feature so that you don't have to scrap your measure.

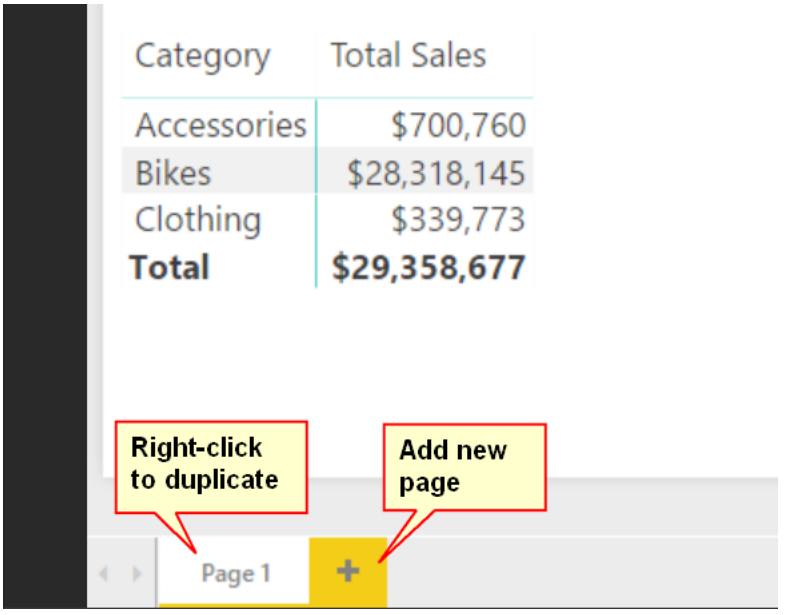
Consider the following complex partially written measure (which is not from AdventureWorks).

```
Complex Measure =
 /* SUMX(Players,
      COUNTROWS(
          FILTER(
              Table1,
              Table1[Playing_Handicap] =
              MAX(Handicaps[Possible Handicaps])
              && Table1[PLayerID] = EARLIER(Players[PlayerID]
    */
```

The formula shown in this example is not important; this example simply shows you what to do when you are not finished writing a measure in order to avoid leaving it half written, going off and doing something else, and forgetting to finish it. The easiest thing to do is to wrap the measure inside the multi-line comment indicators `/* */`. This makes the entire measure a comment that can be stored in your table without throwing an error.

Here's How: Creating New Pages in Power BI

Power BI has a tab section at the bottom of Report view where you can see the various pages you have created. It is easy to create new pages by clicking the yellow plus symbol or by right-clicking any existing page to create a duplicate (shown below).



An alternative to adding a new page is to duplicate an existing page. Duplicating is a great approach especially in this case because it means you get a new page that already has one or more visuals (e.g., a matrix) that you can use for the next exercise. You can also rename the pages to something more descriptive to help find the exercises again later. You should duplicate instead of add new pages if you want to add similar visuals to a new page.

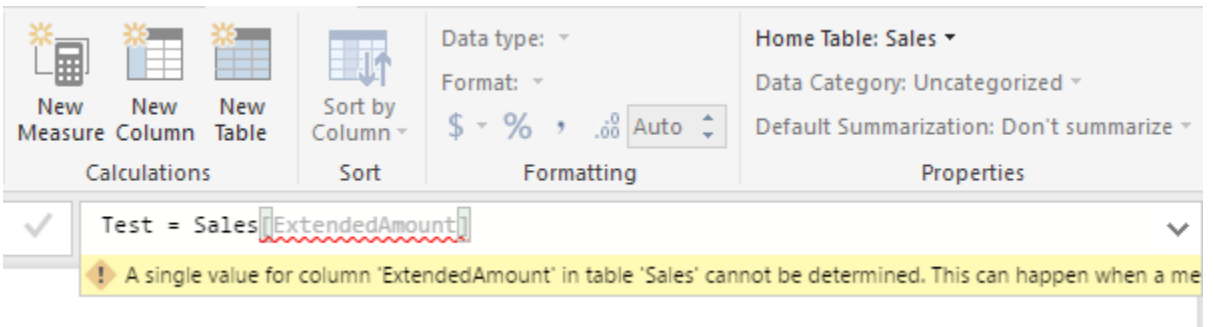
4: DAX Topic: SUM(), COUNT(), COUNTROWS(), MIN(), MAX(), COUNTBLANK(), and DIVIDE()

This chapter starts out with some basic DAX formulas to get you started. Most of the DAX functions in this chapter accept a column as the only parameter, like this: `=FORMULA(`*ColumnName*`)`. The exceptions are `=COUNTROWS(`*Table*`)`, which takes a table (not a column) as the parameter, and `DIVIDE()`, which I cover later in the chapter.

All the functions in this chapter (except `DIVIDE()`) are *aggregation functions*, or *aggregators*. That is, they take inputs from a column or table and somehow aggregate the contents (differently for each formula).

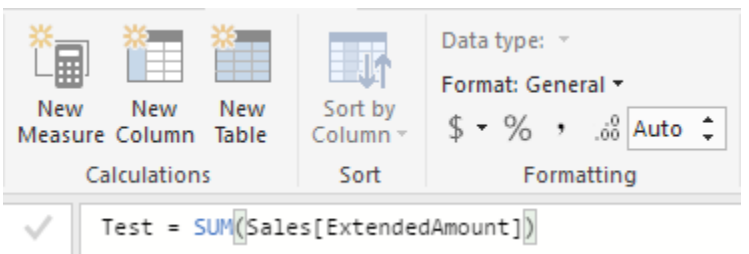
Think about the column `Sales[ExtendedAmount]`, which has more than 60,000 rows of data. You can't simply put the entire column into a single cell in a matrix because Power BI can't "fit" a column of 60,000 numbers into a single cell in the matrix.

The following example shows a DAX formula that uses a "naked" column, without any aggregation function. This does not work when you're writing a measure, as indicated by the error message.



You have to tell Power BI how to aggregate the data from this column so that it returns just a single value to each cell in the matrix. All the aggregators in this chapter effectively convert a column of values into a single value.

The correct way to write this measure is shown below.



Did you notice that this example uses the table name and the column name in the formula? Remember that this is best practice.

> **Note:** Always refer to the table name *and* the column name when writing DAX. Never refer to a column without specifying the table name first. Power BI will do this for you automatically, but you can delete the table name manually (accidentally or deliberately)—though you should not do so! You will understand why I say this shortly.

Now IntelliSense Can Be Your Enemy, Too

It is worth mentioning at this point that sometimes IntelliSense can be your enemy. IntelliSense prompts you with a list of the available functions, tables, columns, and measures only if you are writing the formula correctly. This is very useful when you are doing it correctly because you get a list of the valid syntax. But the downside is that it can be confusing if you can't see the list of possible functions, tables, columns, and measures that you are looking for and you don't know why. If at any time when you are writing a formula

you can't see the functions, measures, columns, or tables you are looking for, you should stop and check your syntax. If the syntax is wrong, IntelliSense stops prompting you. Over time, you will learn to trust IntelliSense, and you will learn to stop and check when it is not working as you expect it to.

Reusing Measures

One important capability in DAX is that you can reuse measures when writing other measures. Say that you create a new measure called `[Total Sales]`. Once this measure exists in the Power BI data model, it can be referenced and reused inside other measures. For example, after creating the measure `[Total Sales]`, you could use the following formula to create a new measure for 10% tax on the sale of goods:

```
Total Tax = [Total Sales] * 0.1
```

Note that the new measure `[Total Tax]` is a calculation based on the original measure `[Total Sales]` multiplied by 0.1.

It is good practice to reuse measures inside other measures.

> **Note:** I did not add the table name in front of the measure name above. That is, I wrote `[Total Sales]` and not `Sales[Total Sales]`. Although you should always add the table name in front of a column (for example, Sales[ExtendedAmount]), it is best practice to omit the table name before a measure. The reason for doing it this way is that a reader can look at Sales[ExtendedAmount] and [Total Sales] and immediately tell that the first is a column and the second is a measure simply by the existence (or not) of the table name.

Writing DAX

It's time to start to write some DAX of your own to get some practice. When I say *write*, I mean sit in front of your PC, open your workbook with the data from Chapter 1 loaded, and really write some DAX. Especially if you have never written formulas using these functions, you should physically do it now, as you read this section. Imagining yourself doing it in your mind is not enough.

If you haven't already done so, go ahead and load the test data by following the steps in Chapter 1. Once it is loaded and prepared, you are ready to create the new measures in the following practice exercises. The first measure you will write is the same one from "Here's How: Using IntelliSense" in Chapter 3.

Practice Exercises

Periodically throughout the rest of this book, you will find practice exercises that are designed to help you learn. You should complete each exercise as you get to it. The answers to all these practice exercises are provided in Appendix A.

Practice Exercises: SUM()

Try to write DAX formulas for the following measures without looking back at how it was done. If you can't do it, refer to Chapter 3 and then give it another go. Remember that you are here to practice! You can find the solutions to these practice exercises"Appendix A: Answers to Practice Exercises" on page 178.

Write DAX formulas for the following columns, using `SUM()`.

1. [Total Sales]

You should have already written this measure earlier in this book. If not, write a new measure that is the total of the sales in the `ExtendedAmount` column from the `Sales` table.

2. [Total Cost]

Create a measure that is the sum of one of the cost columns in the `Sales` table. This measure uses exactly the same structure as the measure above, but it adds the cost of the product instead of the sales amount. You can use any of the product cost columns in the `Sales` table; all the cost columns are the same in this sample database.

3. [Total Margin $]

Create a new measure for the total margin, which is total sales minus total cost. Make sure you reuse the two measures you created above in this new measure.

4. [Total Margin %]

Create a new measure that now expresses the total margin from above as a percentage of total sales. Once again, reuse the measures you created above. I don't cover the `DIVIDE()` function until later in this chapter, but you can try to work out how to use it by using the IntelliSense if you like.

5. [Total Sales Tax Paid]

Create another measure for total sales tax paid. Look for a tax column in the `Sales` table and add up the total for that column.

6. [Total Sales Including Tax]

The total sales amount above excludes tax, so you need to add two measures together to get this total.

7. [Total Order Quantity]

This is similar to the other measures, but this time you add up the quantities purchased. Look for the correct column in the `Sales` table.

How Did It Go?

As you worked through the practice exercises, did you do the following?

- Did you create a matrix first and put `Products[Category]` on Rows in your matrix? (Or did you put something else on Rows, as appropriate for these measures?) This is best practice because it enables you to you get feedback immediately after you write your measure; you can see the results.
- Did you right-click the `Sales` table and select New Measure to start the process? Doing so guarantees that the measure gets placed in the correct table, so you don't lose it. Remember that you should always put a measure in the table where the data is stored, so these practice measures belong in the Sales table.
- Did you reference all columns in your measures in the format *TableName*`[`*ColumnName*`]` (i.e., always reference the table name)? Remember that you should never reference a column in DAX without first specifying the table name; always use the table name and the column name. Power BI makes this easy for you most of the time.
- Did you immediately apply formatting to your measure after you wrote it?
- Did you use the keyboard and look at the IntelliSense when you typed the measures? Try not to use the mouse. It may be faster for you now, but relying on the mouse will prevent you from getting faster with the keyboard in the future. Learn to use the keyboard and follow the process covered in "Here's How: Using IntelliSense."

Remember that the answers to all the exercises in this book "Appendix A: Answers to Practice Exercises" on page 178. Try to avoid peeking at the appendix when you should be thinking and typing. If you do the thinking now, you will *learn how to do it*, and that will pay you back in spades in the future.

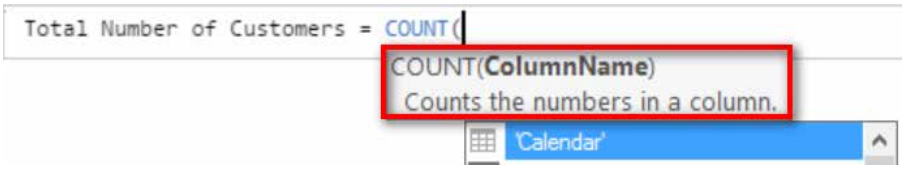
Okay, it's time to move on with a new DAX function.

The COUNT() Function

As you write the formula shown below using `COUNT()`, take the time to look again at how IntelliSense can help you write DAX.

Remember that whenever you type a new formula, you can pause, and IntelliSense shows the syntax for and a description of the function. The description includes some very useful information. For example, in the figure below, the tooltip says that this function "counts the numbers in a column." This gives you three very useful pieces of information. You've already worked out the first one: It *counts*. In addition, this tooltip

tells you that the function counts *numbers* and also that the numbers need to be in a *column*. This should be enough information about the `COUNT()` function for you to write some measures using it.



Practice Exercises: COUNT()

Now it is time to write some DAX formulas using the `COUNT()` function. Find the solutions to these practice exercises in "Appendix A: Answers to Practice Exercises" on page 178.

> **Note:** Don't forget to set up a matrix before you work the following exercises. A good approach is to give the page in your last exercise a name, such as SUM, and then duplicate the page for this next exercise, giving it the name COUNT. This way, you can easily look back at your work later for a refresher. Whenever you set up a new matrix for a new exercise, make sure you have something meaningful on Rows, such as Products[Category]. Look at the image in "How Did It Go?" after these practice exercises if you are not sure how to set up the matrix.

8. [Total Number of Products]

Use the `Products` lookup table when writing this measure. Just count how many product numbers there are. Product numbers and product keys are the same thing in this example.

9. [Total Number of Customers]

Use the `Customers` lookup table. Again, just count the customer numbers. Customer numbers and customer keys are the same thing in this example.

How Did It Go?

Did you end up with the following matrix?

Category	Total Number of Customers	Total Number of Products
Accessories	18,484	35
Bikes	18,484	125
Clothing	18,484	48
Components	18,484	189
Total	**18,484**	**397**

If not, check your answers against those in"Appendix A: Answers to Practice Exercises" on page 178.

> **Note:** The matrix above is a bit confusing because [Total Number of Customers] doesn't seem to be correct. It is returning the same value for every row in the matrix, and this is not something you are used to seeing. But if you think about it, it actually does make sense. You are not counting how many customers purchased these product categories; you are counting the number of customers in the customer master table, and the number of customers doesn't change based on the product categories; the customers are either in the master table or not. (You'll learn more about this in Chapter 5.)

Did you get any errors that you weren't expecting? Did you use the correct column(s) in your measures? Remember from the tooltip above that the `COUNT()` function counts numbers. It doesn't count text fields, so if you try to count the names or descriptions, you get an error.

The COUNTROWS() Function

Let's move on to a new function, `COUNTROWS()`. I prefer to use `COUNTROWS()` instead of `COUNT()`. It just seems more natural to me. These functions are not exactly the same, even though they can be used inter-

changeably at times. If you use `COUNT()` with *TableName*`[`*ColumnName*`]` and the column is missing a number in one of the rows (for some reason), then that row won't get counted. `COUNTROWS()` counts every row in the table, regardless of whether all the columns have a value in every row. So be careful and make sure you select the best formula for the task at hand.

Practice Exercises: COUNTROWS()

For these exercises, rewrite the two measures from Practice Exercises 8 and 9 using `COUNTROWS()` instead of `COUNT()`. Find the solutions to these practice exercises in"Appendix A: Answers to Practice Exercises" on page 178.

10. [Total Number of Products COUNTROWS Version]

Count the number of products in the `Products` table, using the `COUNTROWS()` function.

11. [Total Number of Customers COUNTROWS Version]

Count the number of customers in the `Customers` table, using the `COUNTROWS()` function.

How Did It Go?

Not surprisingly, for Practice Exercises 10 and 11, you should get the same answer you got with `COUNT()`, as shown below.

Category	Total Number of Customers COUNTROWS Version	Total Number of Products COUNTROWS Version
Accessories	18,484	35
Bikes	18,484	125
Clothing	18,484	48
Components	18,484	189
Total	**18,484**	**397**

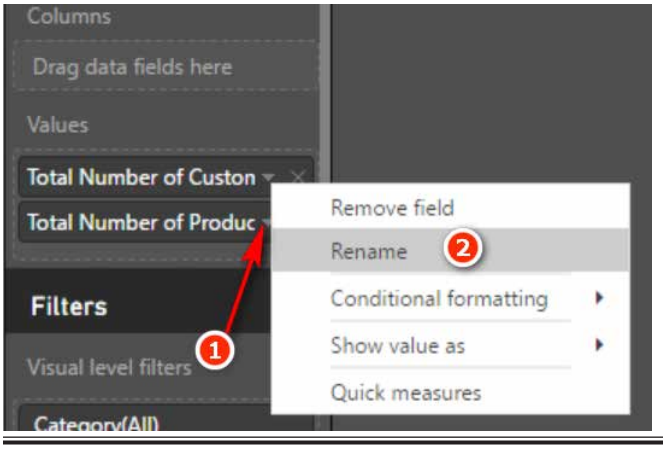
A Word on Naming Measures

You may have noticed that I sometimes use very long and descriptive names for measures. I encourage you to make measure names as long as they need to be to make it clear what the measures actually are. You will be grateful you did down the track, when you are trying to work out the fine difference between two similar-sounding measures.

Here's How: Changing Display Names in Visuals

It is possible to change the display name of a measure or column once it is in a visual. Here are the steps:

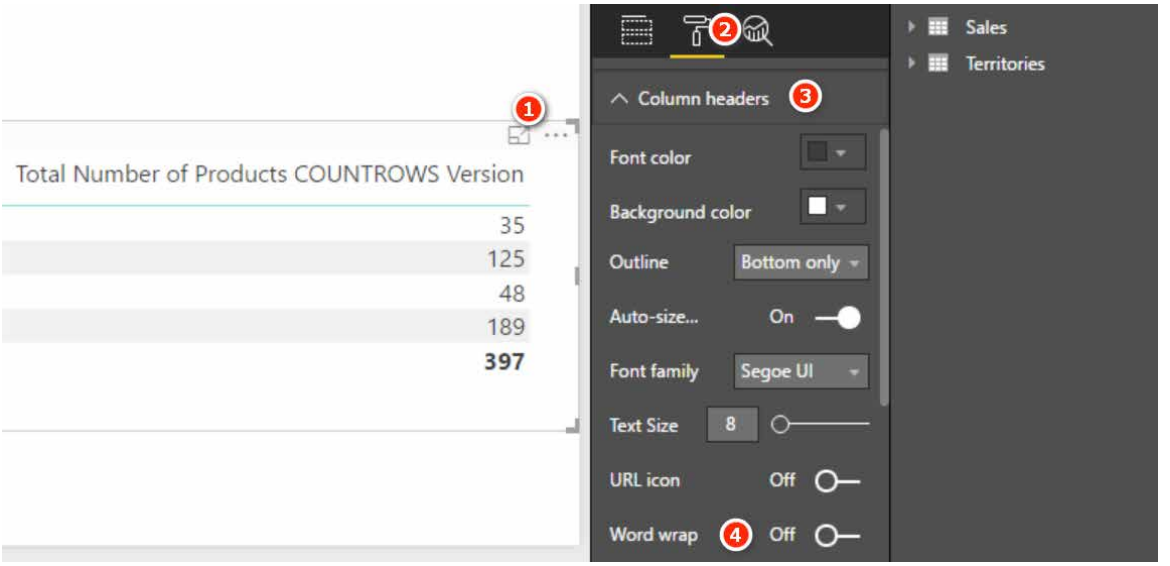
1. Select the visual.
2. Find the measure or column in the Fields List Values section and click its down arrow (see #1 below).
3. Click Rename (#2). This renaming applies only to this single visual and does not change the actual name of the measure or column.



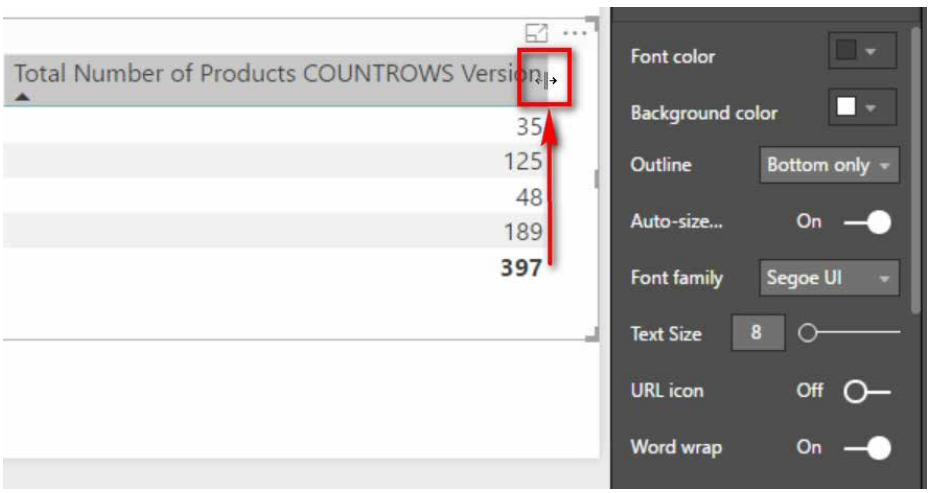
Here's How: Word Wrapping in a Visual

Changing the measure name is useful when you want a shorter name to appear in your visual. Sometimes the name of a measure makes sense only in certain situations (i.e., in some visuals and not in other visuals). For such situations, if you want to keep using a longer descriptive name but want to make it fit in a matrix, you can turn on word wrap. To do so, follow these steps:

1. Make sure you have selected the matrix (see #1 below).
2. Navigate to the Format pane (#2).
3. Select Column Headers (#3).
4. Set Word Wrap to On by using the toggle (#4).



5. Wrap the columns by hovering your mouse to the right side of the column header in the matrix and then clicking and dragging the mouse to the left, as shown below.



6. The following image shows the result of applying word wrap to the matrix from earlier in this chapter.

Category	Total Number of Customers COUNTROWS Version	Total Number of Products COUNTROWS Version
Accessories	18,484	35
Clothing	18,484	48
Bikes	18,484	125
Components	18,484	189

The DISTINCTCOUNT() Function

DISTINCTCOUNT() counts each value in a column once and only once. If a value appears more than once in a column, it is still counted only once. Consider the Customers table. In this case, the customer key is unique, and by definition each customer key appears only once in the table. (Keep in mind that customer key = customer number.) So in this case, using DISTINCTCOUNT() with the customer key in the Customers table gives you the same answer as using COUNTROWS() with the Customers table. But if you were to use DISTINCTCOUNT() with the customer key in the Sales table, you would actually be counting the total number of customers that had ever purchased something—which is not the same thing.

Practice Exercises: DISTINCTCOUNT()

To practice using DISTINCTCOUNT(), create a new matrix and put Customers[Occupation] on Rows in the matrix and [Total Sales] on Values. Then write the following measures using DISTINCTCOUNT(). Find the solutions to these practice exercises in "Appendix A: Answers to Practice Exercises" on page 178.

12. [Total Customers in Database DISTINCTCOUNT Version]

You need to count a column of unique values in the Customers table. Go ahead and write the measure now. When you are done, add the [Total Number of Customers] measure you created earlier to the matrix as well. You should end up with a matrix like the one below.

Occupation	Total Sales	Total Customers in Database DISTINCTCOUNT Version	Total Number of Customers
Clerical	$4,684,787	2,928	2,928
Management	$5,467,862	3,075	3,075
Manual	$2,857,971	2,384	2,384
Professional	$9,907,977	5,520	5,520
Skilled Manual	$6,440,081	4,577	4,577
Total	**$29,358,677**	**18,484**	**18,484**

How Did It Go?

Did you get the same answer as above in the new measure? Did you remember to format the measure to something practical (e.g., a whole number with thousands separators)?

13. [Count of Occupation]

Create a new matrix and put Customers[YearlyIncome] on Rows. Then create the measure [Count of Occupation].

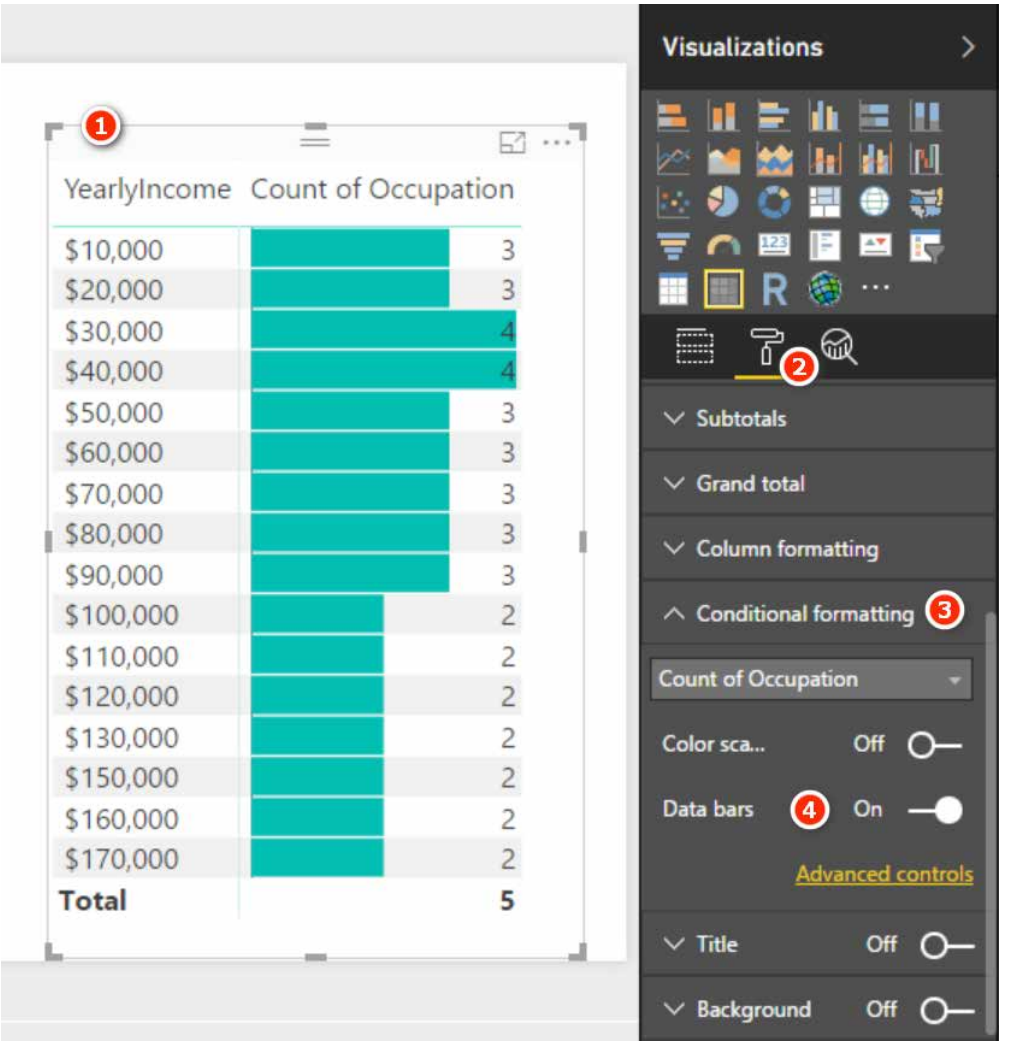
Use DISTINCTCOUNT() to count the values in the Occupation column in the Customers table. You end up with a matrix like the one shown here. The way to read this matrix is that there are customers in three different occupations that have incomes of 10,000, there are customers across four occupations that have incomes of 30,000, etc.

YearlyIncome	Count of Occupation
$10,000	3
$20,000	3
$30,000	4
$40,000	4
$50,000	3
$60,000	3
$70,000	3
$80,000	3
$90,000	3
$100,000	2
$110,000	2
$120,000	2
$130,000	2
$150,000	2
$160,000	2
$170,000	2
Total	**5**

Here's How: Applying Conditional Formatting

It is much easier to read a matrix if you apply some of the formatting features that come with Power BI. For example, compare the matrix above left with the conditionally formatted version above right. I am sure you agree that it is much easier to gather insights from the version on the right. Follow these steps to apply this type of conditional formatting:

1. Make sure you have the matrix selected (see #1 below).
2. Go to the Format pane (#2), select Conditional Formatting (#3), and turn on the formatting effect you want to use, such as Data Bars (#4).



As you can see, using well-placed conditional formatting is a great way to make your matrixes easier to read and helps the insights jump out.

Practice Exercises: DISTINCTCOUNT(), Cont.

The following exercises give you more practice using `DISTINCTCOUNT()`. Find the solutions to these practice exercises in "Appendix A: Answers to Practice Exercises" on page 178.

14. [Count of Country]

Create a new matrix and put `Territories[Group]` on Rows. Write a new measure called `[Count of Country]`, using `DISTINCTCOUNT()` over the `Country` column in the `Territories` table. This matrix, as you can see below, shows you how many countries exist in each sales group.

Group	Count of Country
Europe	3
NA	1
North America	2
Pacific	1
Total	**7**

15. [Total Customers That Have Purchased]

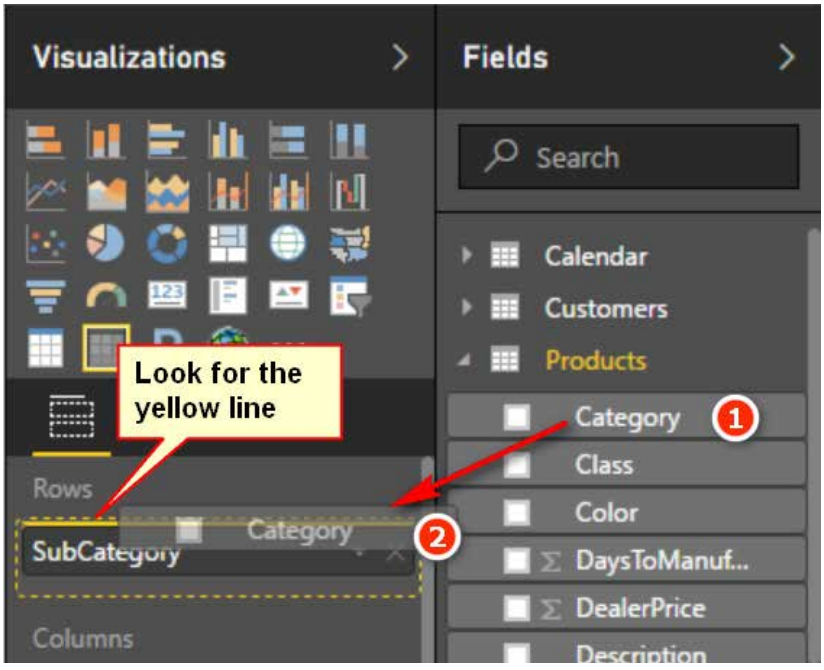
Create a new matrix and put `Products[SubCategory]` on Rows. Then, using `DISTINCTCOUNT()` on data from the `Sales` table, create the new measure `[Total Customers That Have Purchased]`. If you haven't already done so, apply some conditional formatting to the matrix and then sort the column from largest to smallest (by clicking on the heading). You can see below that Tires and Tubes has the largest number of customers who have purchased at least once.

SubCategory	Total Customers that have Purchased
Tires and Tubes	8,490
Road Bikes	6,397
Helmets	5,960
Bottles and Cages	4,548
Mountain Bikes	4,089
Jerseys	3,192
Touring Bikes	2,143
Caps	2,132
Fenders	2,110
Gloves	1,376
Shorts	1,019
Cleaners	875
Hydration Packs	719
Socks	559
Vests	557
Bike Racks	325
Bike Stands	243
Total	**18,484**

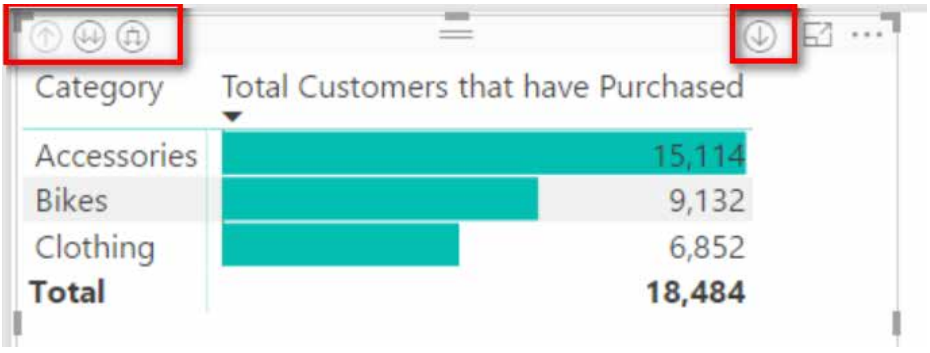
Here's How: Drilling Through Rows in a Matrix

One of the features I really love about Power BI is the ability to nest columns and then drill through. Let's look at how it works, using the matrix from above:

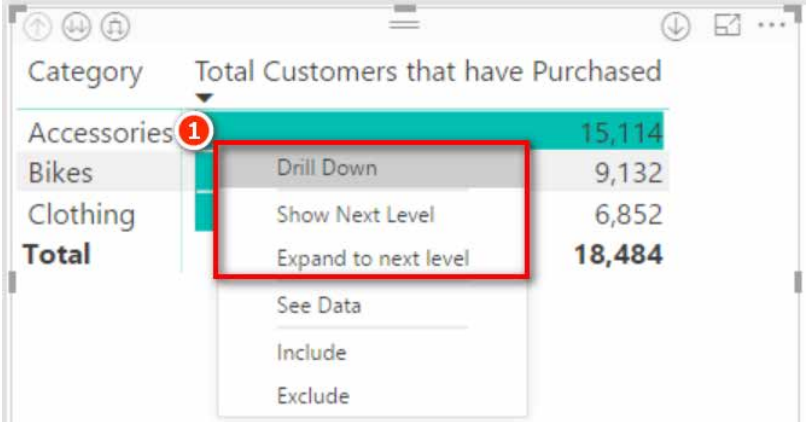
1. Make sure you have the matrix selected and then locate the `Products[Category]` column in the fields list (see #1 below).
2. Drag the column to the Rows drop zone (#2). Take care to check for the dotted yellow line, as indicated below. This line tells you where the column will be dropped (above or below the existing column).
3. Drop the `Products[Category]` column above the `Products[SubCategory]` column, as indicated by the arrow below.



4. Once you do this, the matrix changes in a few subtle ways, as shown below.



5. Note there are three new icons at the top left of the matrix, and there is one new icon at the top right. Also, if you right-click on a product category row (see #1 below), the submenu now has some new menu items.



6. All of these menus provide drill-through capabilities for the columns you have added to the Rows drop zone in your matrix. Don't get confused here; you add the columns from the tables to Rows in your matrix (by stacking one on top of another), and then you can drill through. Spend a few minutes trying out the various drill-through behaviours for each of these menus. You can drill down and drill up through the matrix.
7. Now put `'Calendar'[CalendarYear]` on Columns in your matrix and notice how the matrix changes.
8. Go back to the conditional formatting settings for the matrix, turn off the data bars, and turn on colour scales. You should end up with a matrix like the one below. (I have used the Expand to Next Level drill-through feature in this matrix to get the nested layout shown below. This is very similar to how a pivot table looks.)

Category	2001	2002	2003	2004	Total
Accessories			6,792	9,435	15,114
Tires and Tubes			3,766	5,147	8,490
Helmets			2,541	3,617	5,960
Bottles and Cages			1,903	2,744	4,548
Fenders			879	1,236	2,110
Cleaners			376	509	875
Hydration Packs			300	425	719
Bike Racks			136	191	325
Bike Stands			117	129	243
Bikes	1,013	2,677	4,875	5,451	9,132
Road Bikes	840	2,062	2,558	2,369	6,397
Mountain Bikes	173	615	1,961	2,094	4,089
Touring Bikes			824	1,332	2,143
Clothing			2,867	4,196	6,852
Jerseys			1,316	1,922	3,192
Caps			874	1,280	2,132
Gloves			567	829	1,376
Shorts			435	584	1,019
Socks			246	317	559
Vests			205	354	557
Total	**1,013**	**2,677**	**9,309**	**11,377**	**18,484**

Tip: When you write these measures, remember to select the `Sales` table as the location to store them. Remember that best practice says to put a measure in the table where the data comes from. The easiest way to ensure that a measure is placed in the correct table is to start the process by first right-clicking the correct table and then selecting New Measure.

Practice Exercises: MAX(), MIN(), and AVERAGE()

`MAX()`, `MIN()`, and `AVERAGE()` are aggregators. They take multiple values in a column as an input and return a single value to the matrix. In these next practice exercises, you will create new measures using these aggregators. Find the solutions to these practice exercises in "Appendix A: Answers to Practice Exercises" on page 178.

You should use the columns of data in the `Sales` table for these exercises. There are some additional pricing columns in the `Products` table, but those prices are only theoretical prices, or "list prices." In this sample data, the actual price information related to a transaction is stored in the `Sales` table.

16. [Maximum Tax Paid on a Product]

Remember to use a suitable column from the `Sales` table and use the `MAX()` function.

17. [Minimum Price Paid for a Product]

Again, use a suitable column from the `Sales` table but this time use the `MIN()` function.

18. [Average Price Paid for a Product]

Again, use a suitable column from the `Sales` table but this time use the `AVERAGE()` function.

You should end up with a matrix like the one shown below.

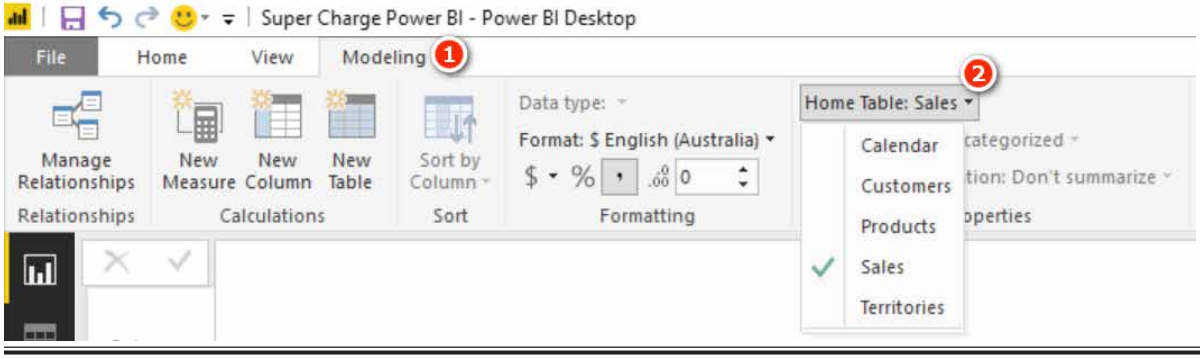
Category	Maximum Tax Paid on a Product	Minimum Price Paid for a Product	Average Price Paid for a Product
Accessories	$12.72	$2.29	$19.42
Clothing	$5.60	$8.99	$37.33
Bikes	$286.26	$539.99	$1,862.42
Total	**$286.26**	**$2.29**	**$486.09**

Note: Notice that when you add these measures straight into a matrix, you get positive immediate feedback about whether your measures look correct. This is only a sense check, and you should of course confirm that your formulas are correct as part of the process.

Here's How: Moving an Existing Measure to a Different Home Table

If you have been following my advice, when you create a new measure, you first right-click on the table where you want the measure to be placed, and then you select the new measure. However, even if you do it this way, it is possible at some stage that you will end up with a measure being in the wrong table. This is fairly easy to fix. Follow these steps to move a measure to a different table:

1. Locate the measure and select it. You can use the Search box at the top of the fields list, if necessary, to find the measure.
2. Navigate to the Modeling tab (see #1 below)
3. Select the Home Table drop-down list (#2) and select the correct table.



Practice Exercises: COUNTBLANK()

In the following exercises, you'll use the `COUNTBLANK()` function to create a measure to check the completeness of the master data.

Create a new matrix and put `Customers[Occupation]` on Rows. Then start to write the new measure. In these exercises, you need to create measures to find out two things:

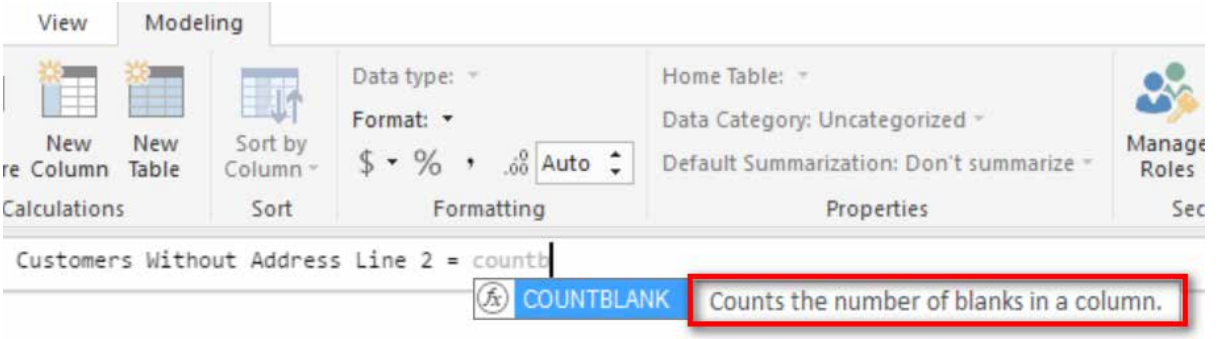
- How many customers are missing Address Line 2 from the master data?
- How many products in the Products table do not have a weight value stored in the master data?

Find the solutions to these practice exercises in "Appendix A: Answers to Practice Exercises" on page 178.

19. [Customers Without Address Line 2]

The `AddressLine2` column is in the `Customers` table. As you write the measure `[Customers Without Address Line 2]`, be sure you do the following:

4. Select the table where you want to store the measure and be sure to add it there.
5. Give the measure a suitable name.
6. Start typing the measure. Pause after you have started to type the formula and read the IntelliSense to see what the function does (if you don't already know). As shown below, it does exactly what you want it to do: It counts how many blanks are in this column.



7. Complete the formula, apply the formatting, check the formula, and then save.

20. [Products Without Weight Values]

The column you need to use is in the `Products` table. You should end up with a matrix like the one shown below.

Occupation	Customers Without Address Line 2	Products Without Weight Values
Clerical	2,878	122
Management	3,007	122
Manual	2,350	122
Professional	5,440	122
Skilled Manual	4,497	122
Total	**18,172**	**122**

Note that the first measure, `[Customers Without Address Line 2]`, is being filtered by the matrix (i.e., `Customers[Occupation]` on Rows), and the values in the matrix change with each row. But the second measure, `[Products Without Weight Values]`, is not filtered; the values don't change for each row in the matrix. You have seen this earlier in this book. The technical term for filtering behaviour in Power BI is *filter context*. Chapter 5 provides a detailed explanation of what filter context is, and that will help you understand what is happening here and why.

The DIVIDE() Function

`DIVIDE()` is a simple yet powerful function that is also known as "safe divide." `DIVIDE()` protects you against divide-by-zero errors in your visuals. A matrix, by design, hides any rows or columns that have no data. If you get an error in a measure inside a matrix, it is possible that you will see lots of rows that you would otherwise not see, and you will possibly also see some error messages. The `DIVIDE()` function is specifically designed to solve this problem. If you use `DIVIDE()` instead of the slash operator (`/`) for division, DAX returns a blank where you would otherwise get a divide-by-zero error. Given that a matrix will filter out blank rows by default, a blank row is a much better option than an error.

The syntax is DIVIDE (*numerator*, *denominator*, *optional-alternate-result*). If you don't specify the alternate result, a blank value is returned when there is a divide-by-zero error.

Practice Exercises: DIVIDE()

Create a new matrix and put `Products[Category]` on Rows. Then add `[Total Sales]` and `[Total Margin $]` to the matrix so you have some data to look at. This helps set the context for the new measures you will write next.

Write the following measures using `DIVIDE()`. Find the solutions to these practice exercises in "Appendix A: Answers to Practice Exercises" on page 178.

21. [Margin %]

Write a measure that calculates the percentage margin on sales (`Total Margin $` divided by `Total Sales`). Reuse measures that you have already written.

> **Note:** This is a duplicate of a measure, called [Total Margin %], that you wrote at the start of this chapter. This time, however, you write the formula by using the DIVIDE() function and give it the name [Margin %]. The result will be the same, of course.

22. [Markup %]

Find `Total Margin $` divided by `Total Cost`.

23. [Tax %]

Divide the total tax by the total sales amount.

How Did It Go?

Did you format the last three measures as percentages, as shown below?

Category	Total Sales	Total Margin	Margin %	Markup %	Tax %
Accessories	$700,760	$438,675	62.6%	167.4%	8.0%
Bikes	$28,318,145	$11,505,797	40.6%	68.4%	8.0%
Clothing	$339,773	$136,413	40.1%	67.1%	8.0%
Total	**$29,358,677**	**$12,080,884**	**41.1%**	**69.9%**	**8.0%**

5: Concept: Filter Propagation

In Chapter 4 we looked at the `COUNT()` function and saw some strange behaviour with the `[Total Number of Customers]` measure. You need to understand the process of filter propagation before you can truly understand what is happening there.

Consider the following matrix.

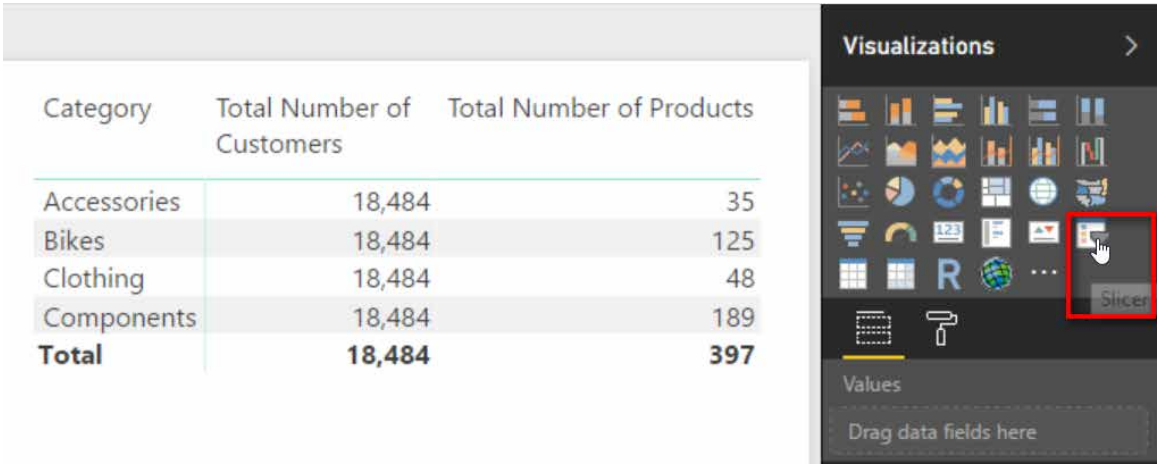
Category	Total Number of Customers	Total Number of Products
Accessories	18,484	35
Bikes	18,484	125
Clothing	18,484	48
Components	18,484	189
Total	**18,484**	**397**

The result [Total Number `of Products]` in this matrix is displaying a different value for each product category (i.e., each row in the matrix has a different number of products), but the value for `[Total Number of Customers]` is the same for each product category in the matrix. The technical reason this happens is because the row labels in the matrix (see #1 above) are "filtering" the products in the `Products` table in the data model *before this measure is evaluated.* But these same rows (product categories) are *not filtering the* `Customers` *table at all*.

A matrix "filters" data and then displays subtotals for each row in the matrix; that's what it's designed to do. The filtering in a matrix is called the *initial filter context—initial* because it is possible to change the filter context later by using the `CALCULATE()` function. (For more information, see Chapter 9.) So the initial filter context is the standard filtering coming from a matrix (or any other visual) before any possible modifications are applied from DAX formulas using `CALCULATE()`.

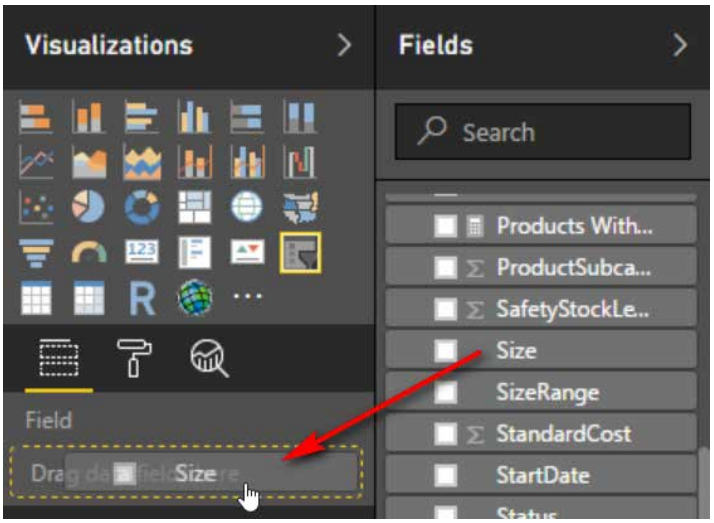
Cross-Filtering Visuals

So far in this book we have used only a single visual on a report page. Now is a good time to add a second visual to the report canvas. Do you remember the process? After clicking on a blank section on the canvas, click on the slicer visual (shown below) to add a new slicer to your report.



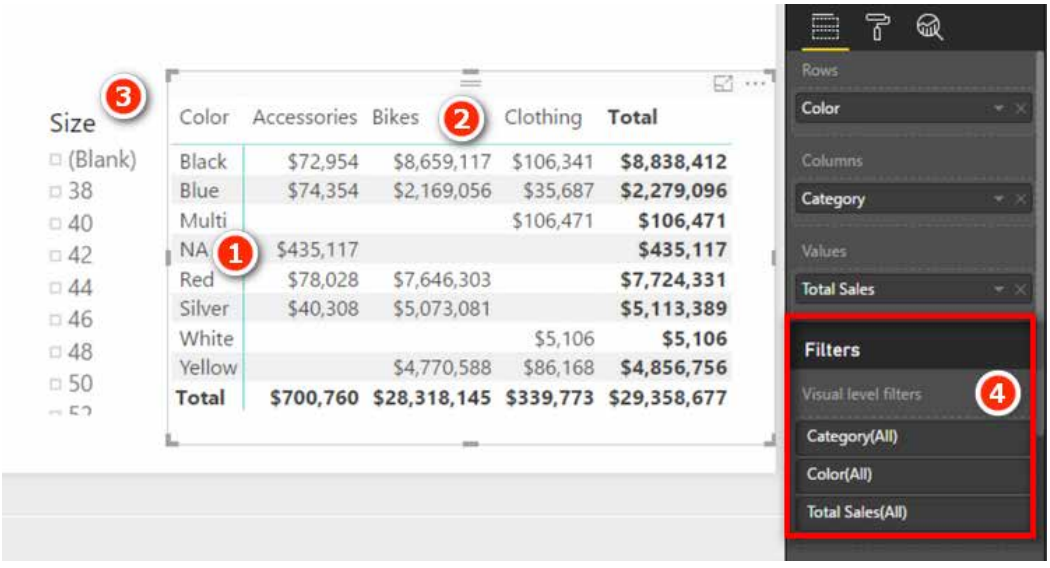
> **Tip:** Before you try to add another visual to the report canvas, you should first click on a blank area of the canvas to ensure that you don't accidently have an existing visual selected. If you try to add a new visual while you have an existing visual selected, the existing visual will be changed instead of a new additional visual being added to the canvas. If you make a mistake, you can always click Undo, but it's best to get into the habit of clicking on a blank area of the canvas before adding a new visual.

After adding the slicer, locate and drag the `Products[Size]` column onto the slicer, as shown below.



The initial filter context in Power BI comes from different areas of the matrix as well as other areas in the report:

- Rows (see #1 below)
- Columns (#2)
- Slicers (#3)
- Filters (#4)



In fact, with Power BI, instead of using the slicer (#3), you can use any other visual on the canvas to filter any of the other visuals. In addition, from the Filters section (#4) you can choose a visual-level filter, a page-level filter, or even a report-wide filter. There is a lot to look for when checking initial filter context in Power BI.

Reading the Initial Filter Context

The following matrix, which first appeared in Chapter 4, shows `[Total Number of Customers]` and `[Total Number of Products]`.

Category	Total Number of Customers	Total Number of Products
Accessories	18,484	35
Bikes	18,484	125
Clothing	18,484	48
Components	18,484	189
Total	**18,484**	**397**

In Chapter 4 this matrix was a bit confusing because it had the same value for `[Total Number of Customers]` on every row in the matrix. Once you learn to read the initial filter context in a visual, you will be able to make more sense of what is going on here.

Let's step through the process of reading the initial filter context from this matrix. Before we do that, though, you should add the `[Total Sales]` measure to the matrix so it looks as shown below.

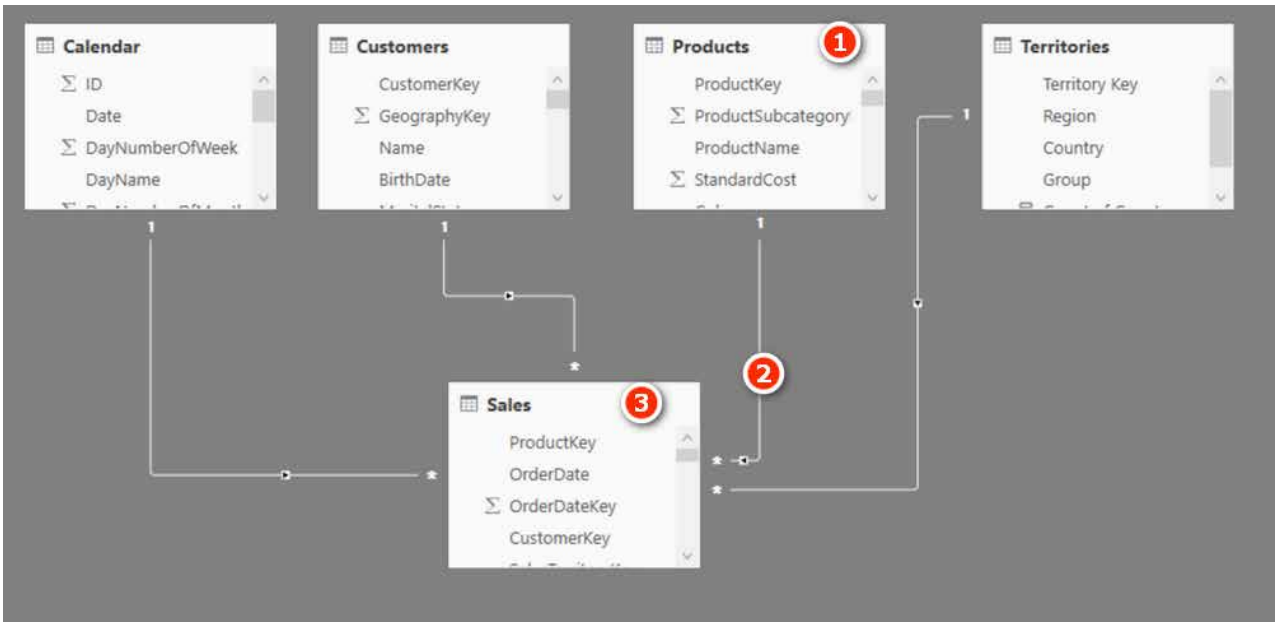
Category	Total Number of Customers	Total Number of Products	Total Sales
Accessories	18,484	35	$700,760
Bikes	18,484	125	$28,318,145
Clothing	18,484	48	$339,773
Components	18,484	189	
Total	**18,484**	**397**	**$29,358,677**

Then point to the cell that's highlighted in red above and say this out loud (really): *"The initial filter context for this cell is* `Products[Category] = Accessories`*."* Then point to the cell underneath the red-highlighted cell; this cell has an initial filter context of `Products[Category] = Bikes`. You can figure out the rest based on this pattern. It is important that you learn to "read" the initial filter context from your visuals because it will help you understand how each value in a visual is calculated. And it is important to refer to the *full table name and column name* because that forces you to look, check, and confirm exactly which tables and columns you are using in your visuals.

Understanding the Flow of the Initial Filter Context

Once you know what the initial filter context is, you can mentally apply the following steps to your data model and track how the filters flow through relationships (though technically the filters propagate from one table to another):

1. The initial filter context coming from the visual is applied to the underlying table(s) in the data model. In this example, there is just one table involved, the `Products` table (see #1 below), where `Products[Category] = "Accessories"`. The `Products` table is filtered so that only rows in the table that are equal to `Accessories` remain; all other rows are filtered so that they are not in play. (Note that the initial filter context can impact more than one table, but in this example, it is just the one table.)
2. The filter applied to the Products table automatically propagates through the relationship(s) between the tables, flowing *downhill* to the connected table(s) (see #2 below). The filters automatically flow from the "one" side of the relationship to the "many" side of the relationship, in the direction of the arrow; or you can think of the filters as flowing from the lookup table to the data table. Whatever terms you use, it's always downhill. This is one of the reasons it is good for beginners to lay out the tables using the Collie layout methodology—with the lookup tables above and the data tables below. This mental cue helps you instantly visualise how automatic filter propagation works. (See "Shaping Data" in Chapter 2.)
3. The connected table, the Sales table, is then also filtered (see #3 below). (Remember that there can be more than one connected table.) Only the products that are of the type Products[Category] = "Accessories" remain in play in the Sales table, and all the other products are filtered away. This is temporary—just for this calculation of *this one single cell in the visual*.



After the automatic filter propagation has been completed, then and only then does the measure get evaluated. In this case, the measure is Total Sales = SUM(Sales[ExtendedAmount]). It returns the value $700,760 to the single matrix cell we started to look at in this example. This process is repeated for every single cell in the matrix, including any subtotal and grand total cells.

> **Note:** A subtotal and grand total are not the additions of the rows above. That's not how it works. Every cell goes through the same "filter, then evaluate" process described above, even if it is a subtotal or grand total row.

Understanding Filter Propagation

Let's look at another cell in the matrix. You always evaluate each cell on its own, without regard for any other cell in the visual, even if the cell is a subtotal or grand total cell. All cells are evaluated using the same process, without regard for any other cell in the visual (a matrix, in this example).

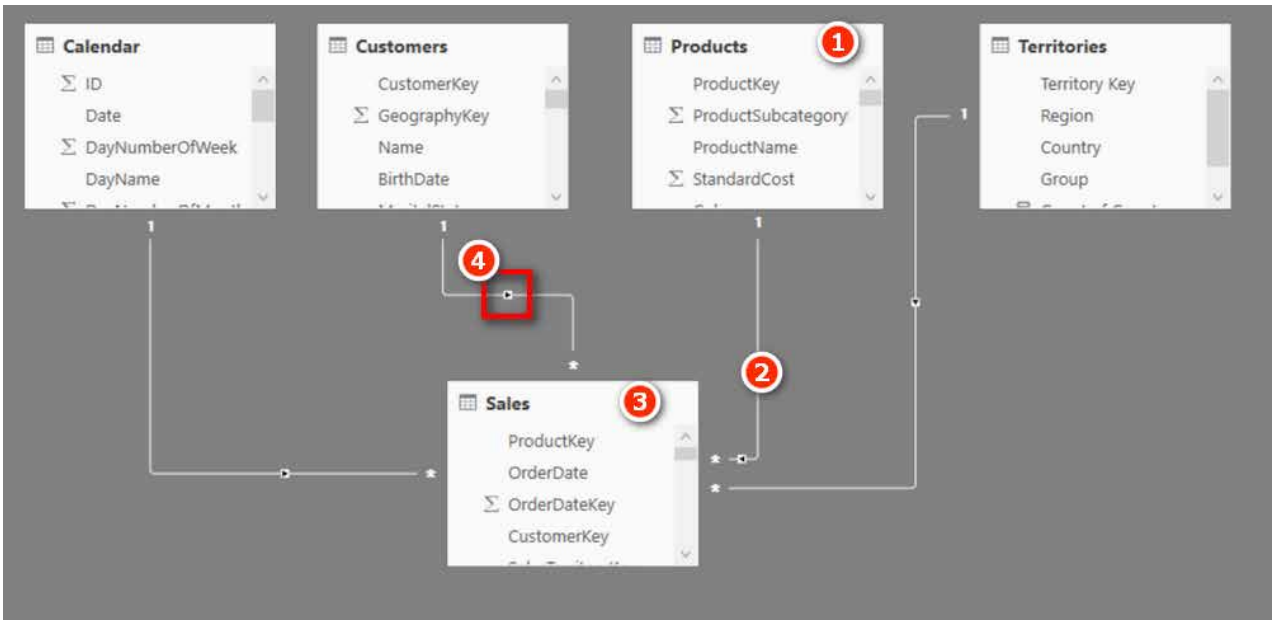
Look at the matrix below and read the initial filter context for the highlighted cell out loud: *"The initial filter context for this cell is Products[Category] = Clothing."*

Category	Total Number of Customers	Total Number of Products	Total Sales
Accessories	18,484	35	$700,760
Bikes	18,484	125	$28,318,145
Clothing	18,484	48	$339,773
Components	18,484	189	
Total	**18,484**	**397**	**$29,358,677**

The initial filter context filters the tables in the data model as follows:

1. The initial filter context is applied to the table(s). In this example, Products[Category] = "Clothing". The Products table (see #1 below) is then filtered so that only rows in the table that are equal to Clothing remain.
2. This filter automatically propagates through the relationships that exist between the tables, flowing *downhill only* to the connected table(s) (see #2 below).

3. The connected table (Sales in this example) is then also filtered so that the same products in the Products table will remain in the Sales table (i.e., only clothing products will be unfiltered in the Sales table) (see #3 below).
4. The filter applied to the Sales table *does not* automatically flow back uphill to the Customers table (or to the other two tables, for that matter) (see #4 below). Filters *only* automatically propagate through the relationships *downhill* from the "one" side of the relationship to the "many" side. The arrow (see #4 below) indicates that the filters do not flow from the Sales table to the Customers table.



So the net result is that the `Customers` table is completely *unfiltered* by the initial filter context. Because the `Customers` table is unfiltered, the total 18,484 is returned to the matrix in this cell (and the same is true for every other cell for this measure in the current matrix).

Even if this doesn't seem right to you yet, realise that it is working as designed. Understanding gives you power, so stick with it until you are clear about how it works. Read this section a few times if you need to. You will learn to love the way it is designed and will learn to make it work for you.

Tip: You simply *must* understand how filter propagation works in Power BI, or you will never be really good at writing DAX. I suggest that you read this chapter multiple times, if necessary, to make sure you are clear about it.

6: Concept: Lookup Tables and Data Tables

All the sample and exercise data in this book so far has been prepared for you; there has been nothing for you to do except follow the instructions. But the simplicity of following my instructions shields you from a deep and important topic: *how* you should structure the tables of data that you load. This chapter covers the various types of tables you can load into Power BI.

This topic is easy to skim over and dismiss as trivial, but in my experience, it is one of the easiest things to get wrong, particularly if you don't know why it is important or how to do it properly. If you get the table structure wrong, then everything else becomes orders of magnitude harder.

> **Tip:** Don't brush off this chapter as unimportant. You need to know this stuff if you want to proceed with speed and confidence, using your own data.

Data Tables vs. Lookup Tables

Two main types of tables are loaded into Power BI: *data tables* (also called *fact tables*, or *transaction tables*) and *lookup tables* (also called *dimension tables*, *reference tables*, or *master data tables*). These two types of tables have some very important differences, as described in the following sections.

Data Tables

Although data tables don't have to be the largest tables loaded into Power BI, they typically are. The `Sales` table used in this book is a transactional table that contains details of individual transactions that occurred in AdventureWorks retail outlets around the world. Every row in this table represents a line item on a register receipt for an individual shopping transaction. Data tables can consist of millions (or even billions) of rows of data. Some examples of data tables include `Sales`, `Budget`, `Exchange Rates`, `General Ledger`, `Exam Results`, and `Stock Count`.

There is no limitation on how often similar transactions can occur and be stored in a data table. Consider a burger chain selling burgers and fries. There could be literally hundreds of transactions each day that are all but identical because the same type of burger can be sold many times on any given day.

Lookup Tables

Lookup tables tend to be smaller than data tables (with fewer rows) and often can be wider (with more columns). Some examples of lookup tables include `Customers`, `Products`, `Calendar`, and `Chart of Accounts`.

Lookup tables have a special feature that makes them different to data tables: A lookup table must have a uniquely identifying code of some type to uniquely differentiate each row in the table. This unique code is often called a key (or primary key, in the database world). Let's consider the `Products` table used in this book. AdventureWorks sells lots of different products—397 to be precise. Each of these products has a unique product code, a three-digit number that is unique for that product. For example, `ProductKey` 212 is a Sports 100 Helmet, Red. No other product in the `Product` table has the same code. If you think about it, this is the way it has to be; there would be chaos if a business used the same product code for different products. The same is true for customers and store ID numbers. In fact, the same is true for the `Calendar` table, given that the date field is a unique ID for each day in the calendar.

Flattened Tables

A good way to help you understand the importance of table structure and the different approaches you can take to loading data is to talk about single large flattened tables. In the early days of Excel pivot tables (before Power Pivot for Excel), you could only create a pivot table on top of a single table of data. If you wanted to do some analysis over a table full of sales data, you could use a pivot table to aggregate the data.

In the image below, the pivot table (see #2 below) has been built on the `Sales` table (#1), and the pivot can easily add up the total sales for each of the products, as identified by the `ProductKey`.

Row Labels	Total Sales
592	$25,425
593	$22,035
594	$28,250
595	$27,120
596	$25,920
597	$26,460
598	$31,319
599	$30,239
600	$22,140
604	$194,396
605	$196,016
606	$208,436
Grand Total	**$837,755**

ProductKey	OrderDate	CustomerKey	SalesTerritoryKey	ExtendedAmount
592	3/06/2004	13035	9	564.99
592	3/06/2004	16684	9	564.99
465	3/06/2004	11965	9	24.49
479	4/06/2004	16730	9	8.99
482	4/06/2004	13643	9	8.99
595	6/06/2004	13036	9	564.99
489	7/06/2004	18715	9	53.99
491	8/06/2004	19578	9	53.99
483	8/06/2004	13634	9	120
484	9/06/2004	13668	9	7.95
463	9/06/2004	12351	9	24.49
541	9/06/2004	19623	9	28.99
578	9/06/2004	14262	9	1214.85
489	10/06/2004	17059	9	53.99

That is all well and good until your report needs some extra data that is not part of the `Sales` table. In the image above, what would happen if you wanted to know the name of the product, or the product category, or the subcategory, or anything else for that matter? Well, in the old days, you would find a `Products` table somewhere, and then you would write a `VLOOKUP` (or `INDEX`/`MATCH`) to go and fetch the extra columns of data that you needed for your reporting and bring it into the single `Sales` table. From there you could use the new columns in your pivot tables. This process of bringing in the missing columns is called *de-normalising*. After you have de-normalised, your `Sales` table ends up looking something like the one below (but of course it would be much bigger if you needed more columns).

ProductKey	OrderDate	Category	ProductName	SubCatetory	Color	ExtendedAmount
592	3/06/2004	Bikes	Mountain-500 Silver, 42	Mountain Bikes	Silver	564.99
592	3/06/2004	Bikes	Mountain-500 Silver, 42	Mountain Bikes	Silver	564.99
465	3/06/2004	Clothing	Half-Finger Gloves, M	Gloves	Black	24.49
479	4/06/2004	Accessories	Road Bottle Cage	Bottles and Cages	NA	8.99
482	4/06/2004	Clothing	Racing Socks, L	Socks	White	8.99
595	6/06/2004	Bikes	Mountain-500 Silver, 52	Mountain Bikes	Silver	564.99
489	7/06/2004	Clothing	Short-Sleeve Classic Jersey, M	Jerseys	Yellow	53.99
491	8/06/2004	Clothing	Short-Sleeve Classic Jersey, XL	Jerseys	Yellow	53.99
483	8/06/2004	Accessories	Hitch Rack - 4-Bike	Bike Racks	NA	120
484	9/06/2004	Accessories	Bike Wash - Dissolver	Cleaners	NA	7.95
463	9/06/2004	Clothing	Half-Finger Gloves, S	Gloves	Black	24.49
541	9/06/2004	Accessories	Touring Tire	Tires and Tubes	NA	28.99

Do you spot the issue with the table above? The problem with a table like this is the duplication of data. Note how the different product categories are repeated all the way down the `Category` column. The reality is that for small tables of data (including lookup tables), such repetition doesn't really matter much because the overall file size will be quite small. However, if the table becomes large (millions or billions of rows), then adding all these extra columns of information can become a big problem (literally). In the old days, you had to bring all the relevant columns into the one table by writing one or more `VLOOKUP` columns to fetch the extra columns needed. Power BI is built differently, though. It doesn't require you to bring all the columns into the one table, and this makes everything easier and more efficient.

> **Note:** The Power BI data modelling engine is a columnar database that compresses the data it loads. The details behind this are quite technical and beyond the scope of this book. However, there are a couple of key points you should be aware of. The more unique values in a column, the less the data will be compressed. In addition, the number of columns you have in your data tables is much more important than the number of rows; that is, fewer columns and more rows is better than more columns and fewer rows. This is particularly true for large tables.

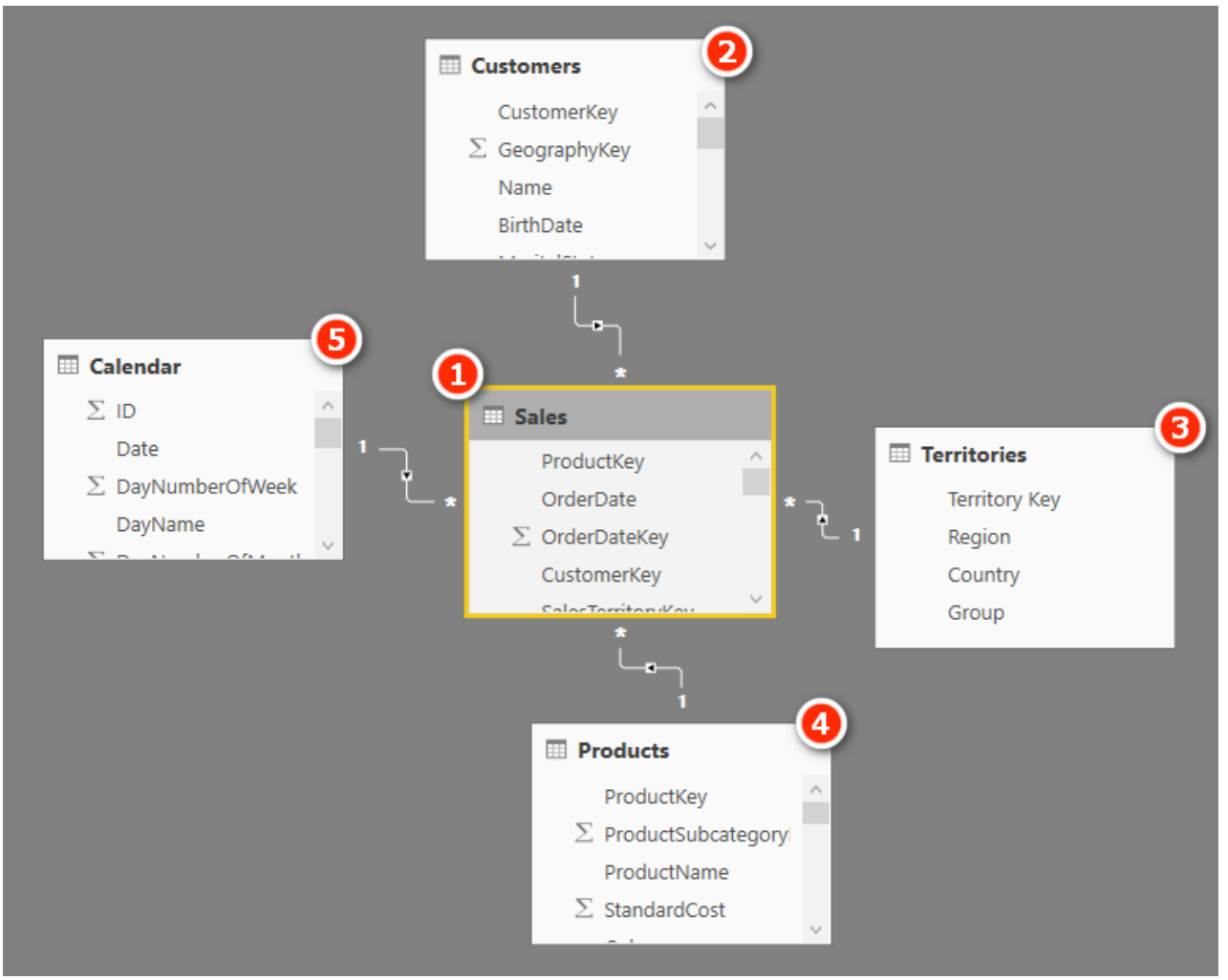
Joining Tables by Using Relationships

A better approach to solving the problem of repetitive data is to keep the repeating data in separate subtables. In the case of products, there is only one column of information that is needed in the `Sales` table to uniquely identify every single product, and that is the product code (`ProductKey`). If the `Sales` table contains the unique product key, it is possible to fetch any extra information needed from a product master table when it is needed. So rather than requiring you to write a `VLOOKUP` to go and bring the product information into the `Sales` table, Power BI allows you to load both tables into the data model and create a single relationship between them. Once the relationship has been created, the tables will work together as if they were a single unit, without the need to inefficiently create duplicate data in the `Sales` table.

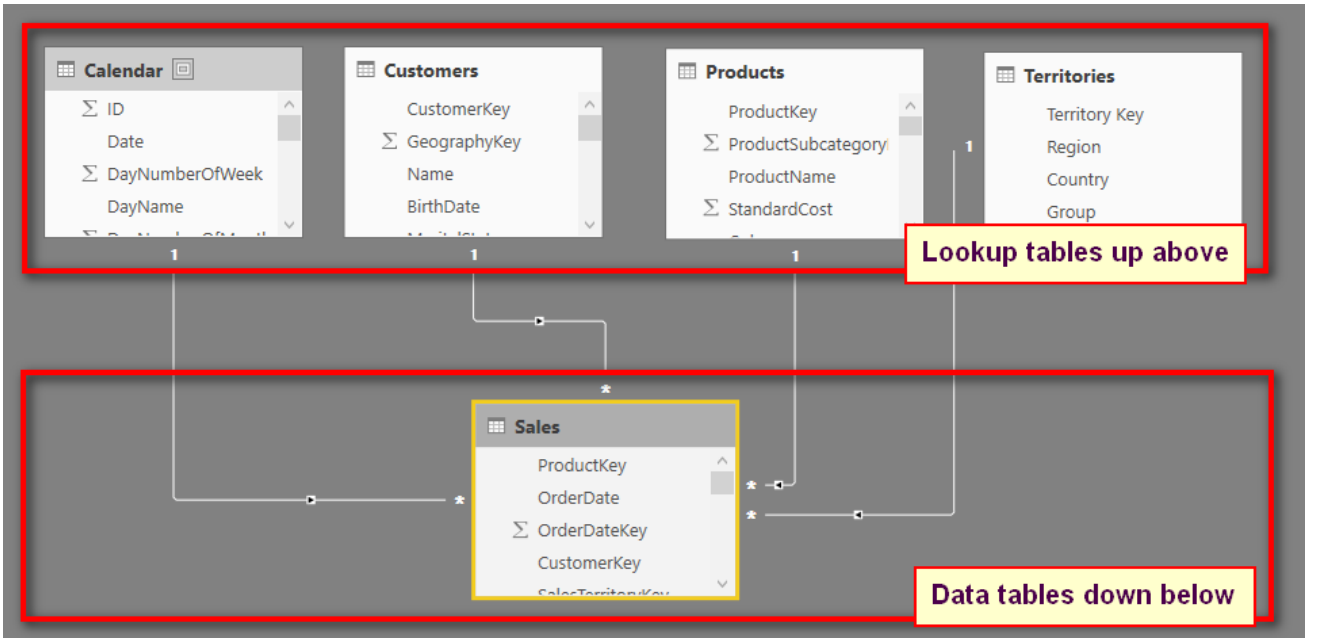
As mentioned briefly in Chapter 2, the structure of the tables and any relationships between them in a data model is sometimes referred to as a *schema*. There are a few different classes of schemas, and the following sections cover the most common types.

Star Schema

The image below shows the star schema structure used in this book. The `Sales` table (#1) is a data table and is located at the centre of the star. The other tables (#2, #3, #4, #5) are lookup tables and are shown as points on the star. This structure is called a *star schema* due to its shape.

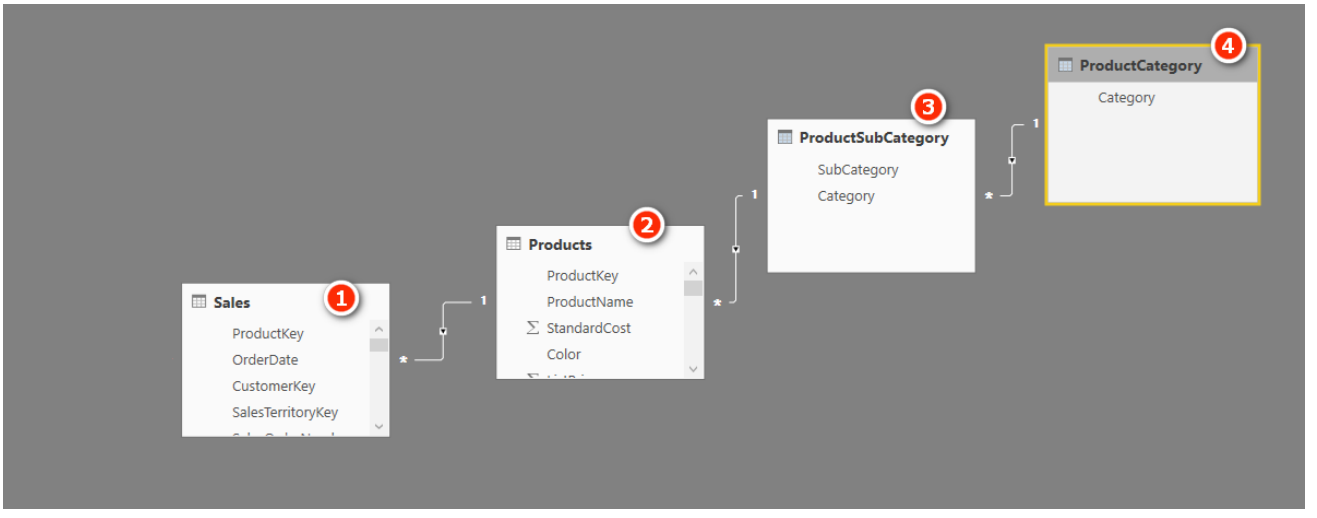


However, as you learned in Chapter 2, you can reposition the tables any way you choose, and I recommend using the Collie layout methodology, as shown below. You can see that the tables here are the same as in the diagram above, but the layout is different. The layout has no impact at all on the way Power BI operates, but it does give you a visual clue as to which are the lookup tables and which are the data tables because you have to "look up" to see the lookup tables. Also, in the old days, you would write a `VLOOKUP` to go and fetch those extra columns, so there is another link to the past between the words `VLOOKUP` and *lookup* table.



Snowflake Schemas

Sometimes when normalising data into tables, there can be multiple levels of lookup tables. Consider the image below. There is a single data table, `Sales` (see #1 below), and there are three lookup tables all chained together in a row (#2, #3, and #4). Table #4 is a lookup table of table #3, which is a lookup table of table #2, which is a lookup table of table #1.



This data structure is common in traditional transactional databases as it is the most efficient way to store the data in those systems. However, this is not the best way to structure data in Power BI. There are few reasons this approach is not the best for Power BI:

- Every relationship comes at a cost. The extra relationships will potentially have negative performance impacts on the database.
- Business users will be building reports using your database design, and they will see all the tables in the data model. The structure above is confusing to and onerous on an end user who is trying to understand.
- Power BI was built from the ground up to be very efficient in the way it stores repetitive data in columns, particularly in the smaller lookup tables, so there is simply no reason to do it as shown in the image above.

Advice on Loading Your Own Data

There are a few things you can do to get off on the right track when it is time to build your own data models:

- Where possible, keep your data tables long and skinny. If necessary, get rid of extra columns of "data" by unpivoting your data, particularly if a data table is very wide (i.e., has a lot of columns). Each additional column of data compresses less well than the last one, which means long, wide data tables can be a real issue.
- Move repeating attribute columns from your data tables and create lookup tables instead. But be careful that you don't overdo it. If a lookup table has only two columns (e.g., `Key` and `Description`), then it may be better to drop the `Key` column and just load the description directly into the data table.
- If you have lookup tables joined to other lookup tables, consider flattening them out into a single wider lookup table. This is generally a better design for Power BI.

Definitely do not just accept the table shape and structure coming from your transactional system.

Note: Transactional databases and reporting databases are not the same, so don't try to use the table structure from your transactional system in Power BI.

7: DAX Topic: The Basic Iterators SUMX() and AVERAGEX()

The functions covered in Chapter 4 are all aggregation functions. Each of those aggregation functions acts on an entire column or table and uses a specific aggregating technique to return a single value to a cell in a visual on a report.

Another class of functions can possibly return the same answers as the aggregation functions but using a different approach. These "X-functions" (i.e., any functions that have an `X` at the end of the name) are part of the family called *iterators*.

Iterators and Row Context

The main difference between iterators and the other functions we have looked at so far is that iterators have what is called *row context*, which means that a function is "aware" of *which row* it is referencing at any point in time. Rather than getting into a theoretical explanation, let's move on to working with the iterator `SUMX()` and talk about row context as we use this function.

Using SUMX(table, expression)

`SUMX()` takes two parameters: a table name and an expression to evaluate. `SUMX()` creates a row context in the specified table and then iterates through each row of the table, one row at a time, and evaluates the expression for each row as it gets to it before finally adding together the interim results for each row. Row context is a concept in DAX that involves creating "awareness" of the existence of the rows in the table so a function can iterate through them one at a time until it has touched every single row once and only once. You can think of row context as a *checklist* of all rows that `SUMX()` uses to keep track of where it is. `SUMX()` can work through the rows one at a time, metaphorically "checking off" each row to make sure none has been missed. This row context exists only in certain DAX formulas, including the X-functions (discussed in this chapter), calculated columns (see Chapter 8), and with `Filter()` (see Chapter 14).

To demonstrate the point, let's look at how to write a new version of the `[Total Sales Including Sales Tax]` measure. First, create a new matrix, put `Products[Category]` on Rows, and then write out this measure:

```
Total Sales Including Tax SUMX Version
    = SUMX(Sales, Sales[ExtendedAmount] + Sales[TaxAmt])
```

Notice that in this measure, you are not wrapping the columns in an aggregation function. In this case, you are referring to "naked columns," and that is perfectly okay inside an iterator. There is no need to wrap the columns in an aggregation function when using an X-function (or any other function that creates a row context). The way an X-function works is that it goes to the table specified (in this case, `Sales`), creates a row context for it to use as a reference, and then takes each single row in the table, one at a time, and evaluates the expression for that single row. Once it has created an interim result for each row in the table, it adds together all the interim results.

As you can see illustrated below with the red box in the image, when there is only one single row from the table in play, DAX is able to refer to the exact intersection of each column referred to in the formula and the specific row it is currently iterating over. Therefore, during each step of the iteration process, the column names in the expression are actually only referring to a single value—the value that is the intersection of the single column and the current row in the row context.

ExtendedAmount	TaxAmt
2443.35	195.468
2443.35	195.468
21.49	1.7192
21.49	1.7192
1120.49	89.6392
69.99	5.5992
69.99	5.5992
69.99	5.5992
69.99	5.5992
69.99	5.5992

One row at a time, the single value in the Sales[ExtendedAmount] column is added to the single value of the Sales[TaxAmt] column. After the first row is evaluated (and the result is stored temporarily in memory for later), `SUMX()` selects a second row and does the same thing, then a third row and does the same thing, and so on until it has iterated through every single row in the table, missing none. It iterates through every single row once and only once (not necessarily in the order you see on the screen). When it has completed this calculation for every single row in the specified table, it sums all the results together and returns a single value to the matrix cell.

> **Note:** I refer above to iterators working "one row at a time." It is convenient to think of iterators working in this way, and indeed that is the logical execution approach. In reality, though, the Power BI engine has been built and optimised to work very efficiently under the hood. In many circumstances, the actual physical execution is much more efficient than is implied by "one row at a time" logical execution. This is a very deep technical topic and is beyond the scope of this book. The key thing to note is that you should not think that iterators are inherently inefficient because the Power BI engine optimisations can make the physical execution very efficient indeed.

Practice Exercises: SUMX()

Write the following measures for practice. Find the solutions to these practice exercises in "Appendix A: Answers to Practice Exercises" on page 178.

24. [Total Sales SUMX Version]

Multiply quantity by unit price from the appropriate columns in the `Sales` table.

25. [Total Sales Including Tax SUMX Version]

Add the `ExtendedAmount` column together with the appropriate tax column in the `Sales` table.

26. [Total Sales Including Freight]

Add the `ExtendedAmount` column to the `Freight` cost.

How Did It Go?

Did you get the following matrix?

Category	Total Sales SUMX Version	Total Sales Including Tax SUMX Version	Total Sales Including Freight
Accessories	$700,760	$756,821	$718,281
Bikes	$28,318,145	$30,583,596	$29,026,099
Clothing	$339,773	$366,954	$348,267
Total	**$29,358,677**	**$31,707,371**	**$30,092,647**

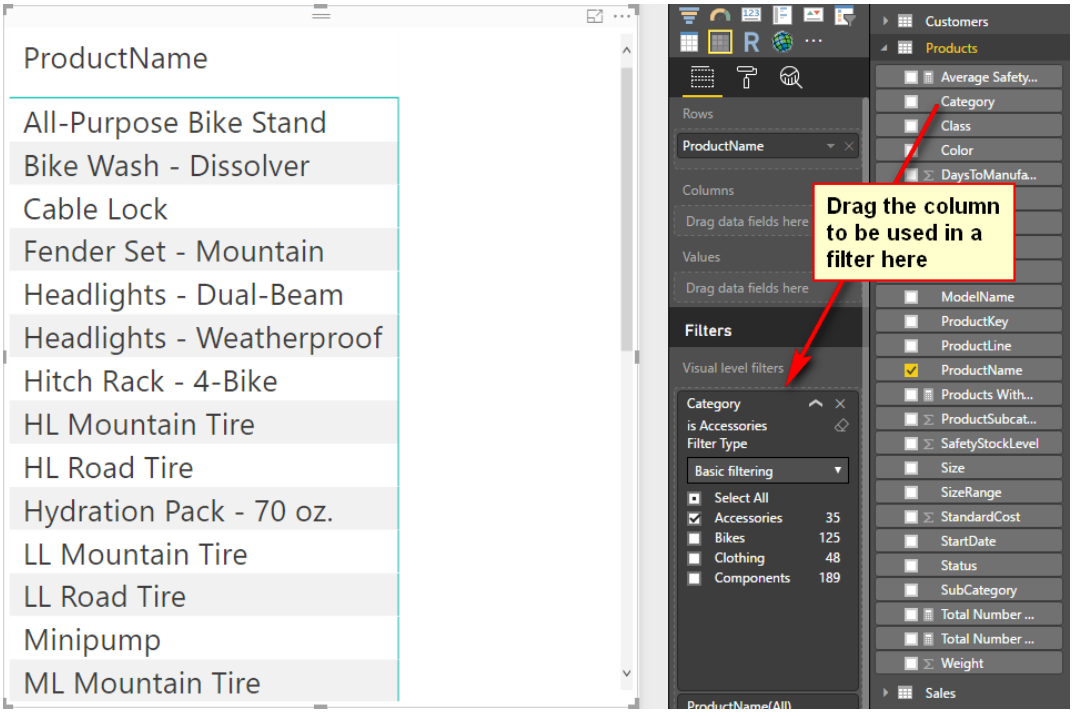
Make sure you are following these steps to minimise rework:

1. Put the measures in the correct table by right-clicking the target table and selecting New Measure. (Do not select New Measure from the menu!)
2. Give the measure a meaningful name and include spaces in the name.
3. Apply suitable formatting immediately after writing the measure.
4. Write the formula and then check to ensure that it was written correctly by adding it to a matrix to check your results.

27. [Dealer Margin]

Create a new matrix. You can select an existing matrix and then use Ctrl+C and Ctrl+V to copy and paste it as a new matrix if you like. One benefit of doing this is that any formatting you have applied to your first matrix

will be copied to the new matrix, which saves you time and effort. Put `Product Category` on Filters and then select `Accessories` from this filter. Then put `Product Name` on Rows. You should have something like what is shown below (though what is shown here is truncated). Note that I have used a filter from the Visual Level Filters section to apply the filter on `Product Category`.



Write a measure that shows the theoretical margin the dealer gets (i.e., the difference between the product list price and the product dealer price). Both columns you need for this measure are in the `Products` table. Did you get the answers shown below? (Once again, my matrix in the image below is truncated; it shows only the first nine rows; in reality, it is longer.)

ProductName	Dealer Margin
All-Purpose Bike Stand	$63.60
Bike Wash - Dissolver	$3.18
Cable Lock	$10.00
Fender Set - Mountain	$8.79
Headlights - Dual-Beam	$14.00
Headlights - Weatherproof	$18.00
Hitch Rack - 4-Bike	$48.00
HL Mountain Tire	$14.00
HL Road Tire	$13.04

When to Use X-Functions vs. Aggregators

Now you know that you can use X-functions such as `SUMX()`, and you can also use aggregators such as `SUM()`, and they do similar things but using different approaches. Which should you use? The following examples will help you figure this out. The samples below are simplistic tiny tables for illustration purposes only. In real life, tables of data are of course much larger.

Example 1: When the Data Doesn't Contain the Line Total

If your `Sales` table contains a column for quantity (`Qty` in the image below) and another column for price per unit, you need to multiply the quantity by the price per unit to calculate total sales (because the actual total doesn't exist at the line level in the table).

Date	Product	Qty	Price Per Unit
1/01/2003	A	3	2.5
1/01/2003	B	1	6.8
2/01/2003	A	5	2.5
2/01/2003	C	3	3.5

If this is the structure of your data, then you simply must use `SUMX()`, like this:

```
Total Sales 1 = SUMX(Sales,Sales[Qty] * Sales[Price Per Unit])
```

In this example, you have to calculate the totals for each row first, one row at a time. This is what the iterator functions are designed to do.

Example 2: When the Data Does Contain a Line Total

If your data contains a single column with the extended total sales for that line item, you can use `SUM()` to add up the values:

```
Total Sales 2 = SUM(Sales[Total Sales])
```

Date	Produc	Total Sales
1/01/2003	A	7.5
1/01/2003	B	6.8
2/01/2003	A	12.5
2/01/2003	C	10.5

There is no need for an iterator in this example. Note, however, that you *could* still use `SUMX()` like this to get the same answer:

```
Total Sales 2 alternate = SUMX(Sales, Sales[Total Sales])
```

Note: In the formula above, there is only a single column for the expression parameter: Sales[-Total Sales]. This is a valid DAX expression, and it will work just fine. For each row in the iteration of `SUMX`, the formula just takes the line-level total for this column. At the end, it simply adds up all the values.

So Which Should You Use, SUM() or SUMX()?

Whether you use `SUM()` or `SUMX()` comes down to personal preference and the structure of your data. For most data models, it will make little or no difference, so you can choose the one that suits you the best. However, let's take another look at the two approaches by looking at the examples from above. First, let's take another look at the table from Example 2 (see the previous image).

Every value in the column `Sales[Total Sales]` is unique (i.e., there are no duplicates). Assuming that the real table is very large and has lots of unique values, this column would not compress well.

Date	Product	Qty	Price Per Unit
1/01/2003	A	3	2.5
1/01/2003	B	1	6.8
2/01/2003	A	5	2.5
2/01/2003	C	3	3.5

Now look again at the table above again.

In this table, there are duplicate values in the `Qty` column and also in the `Price Per Unit` column. It is likely that loading the data in this table (Example 1) will result in better compression than with the data in Example 2 because of these duplicate values in each column. In real life, data that compresses the best in columns often is the most efficient. This may seem counterintuitive if you think about "iteration" as being a slow, row-by-row evaluation—and it is understandable that you may think of it that way. However, the underlying Power Pivot engine is optimised to work just as efficiently with `SUMX()` as it does with `SUM()`. (There are exceptions to this rule, but that is quite a complex topic and beyond the scope of this book.)

Avoiding Data You Don't Need

One important point to note before moving on is that you should *definitely not* have all three columns as shown across the previous two examples. It should be obvious that if you have quantity and price per unit in a table, you can "calculate" the value of total sales any time you need it. Similarly, if you have total sales and quantity in a table, you can calculate price per unit any time you like.

Generally speaking, you should not include in your data model columns of redundant data that can be calculated on-the-fly. Doing so increases file size and makes everything refresh more slowly. The general rule is to bring in the minimum number of columns you need to do the job, and it is best to bring in the columns with the lowest numbers of unique values where possible.

When Totals Don't Add Up

There is another use case that requires you to use `SUMX()` or some other iterator. I have created a small table of sample data (shown below) to explain the problem and show the solution.

Customer	Spend per Visit	Number of Visits
A	50	7
B	40	3
C	100	12
D	15	4

The table above shows four customers, with the average amount of money they have spent each time they have shopped as well as the number of times they have shopped. If you load this data into Power BI and then use aggregating functions to find the average amount spent across all customers as well as the total amount spent, you get the wrong answers, as shown below.

Customer	Avg Spent per visit Wrong	Total Number of Visits	Total Spent Wrong
A	$50	7	$350
B	$40	3	$120
C	$100	12	$1,200
D	$15	4	$60
Total	**$51**	**26**	**$1,333**

The following measures are used above:

```
Total Number of Visits = sum(VisitData[Number of Visits])
Avg Spent per visit Wrong= AVERAGE(VisitData[Spend per Visit])
Total Spent Wrong = [Avg Spent per visit Wrong] * [Total Number of Visits]
```

The first measure, `[Total Number of Visits]`, is correct because the data is additive, but the other two measures give the wrong results. This is a classic situation where you can't perform multiplication on the averages at the grand total level. Given the original sample data, the only way to calculate the correct answer is to complete a row-by-row evaluation for each customer in the table, as shown below.

Customer	Avg Spent per visit Correct	Total Number of Visits	Total Spent SUMX
A	$50.00	7	$350
B	$40.00	3	$120
C	$100.00	12	$1,200
D	$15.00	4	$60
Total	**$66.54**	**26**	**$1,730**

The table above includes a SUMX() to find the total spent, row by row. Only then does the matrix calculate the average spent per visit. Here is the complete set of correct formulas:

```
Total Number of Visits = SUM(VisitData[Number of Visits])
Total Spent SUMX = SUMX(VisitData,VisitData[Spend per Visit] * VisitData[Number of Visits])
Avg Spent per visit Correct = DIVIDE([Total Spent SUMX] , [Total Number of Visits])
```

Whenever you see that the totals in your visuals don't add up, you should try to figure out what is happening row by row in the visual that is not happening in the total row. Such a problem is always caused by some sort of filter context provided by the rows in the visual that is not replicated in the total rows. You should look at the visual and ask, "How can I simulate the row-by-row filtering that I see in the visual in the total row?" When you answer that question, you will know how to solve the problem of totals not adding up, normally using an iterating function like `SUMX()`.

Avoiding Too Many Calculated Columns

Now is a good time to talk about the most common mistake I see Excel users make when moving to Power BI. The formulas we have been writing using `SUMX()` can also be written directly into a calculated column in a table. But this *normally* is the *wrong* way to do it. Let me explain why.

To write a calculated column, right-click on the table that should receive the new column and then select New Column. You could rewrite the `[Total Sales Including Tax SUMX Version]` measure as a calculated column as shown here:

```
Total Sales Plus Tax Column = Sales[ExtendedAmount] + Sales[TaxAmt]
```

Before we move on, let me point out something with regard to the syntax of the above calculated column and the measure `[Total Sales Including Tax SUMX Version]`. Here are the two formulas:

```
Total Sales Plus Tax Column
= Sales[ExtendedAmount] + Sales[TaxAmt]
Total Sales Including Tax SUMX Version
= SUMX(Sales,Sales[ExtendedAmount] + Sales[TaxAmt])
```

Take a look at the part of the two formulas highlighted in bold. This section is identical in the two formulas; it adds together the values in these two columns while iterating in a row context. What is different with the `SUMX()` function is that there is an additional parameter that specifies *which* table `SUMX()` should iterate over. This extra parameter is not required in a calculated column because the column is physically placed in the table itself. In other words, a calculated column is an iterator that iterates over the table in which it is placed.

A calculated column has a row context—just like `SUMX()`—and as a result, it is fine to refer to naked columns. Just as with `SUMX()`, the calculated column has a row context and iterates over the rows in the calculated column one at a time. At each step in the process, there is only one row in play, and hence each column has only one possible value for each row iteration step. So you end up with a column of data like the one below, with the calculated total for each row stored in the column.

ExtendedAmount	TaxAmt	Total Sales Plus Tax Column
24.99	1.9992	$26.99
24.99	1.9992	$26.99
539.99	43.1992	$583.19
539.99	43.1992	$583.19
539.99	43.1992	$583.19

But there is one *big* problem with this approach—and I do mean *BIG*. The problem is that a calculated column always evaluates every row and stores the answer in the workbook as a value in the column in the table. This takes up space in the workbook. What's more, the compression applied to calculated columns is reportedly not as good as for imported columns, and hence the data could be stored less efficiently in the workbook. Now compare this to the `SUMX()` measure created earlier. The measure `[Total Sales Including Tax SUMX Version]` does not store any values in your workbook other than values needed to display in any visuals in your Power BI workbooks. If your `Sales` table has 60,000 rows of data, it probably doesn't matter. But if your `Sales` table has 50 million rows of data, it definitely will matter.

People with an Excel background tend to gravitate toward writing calculated columns rather than writing measures because that's what they're used to doing. They are used to living in a spreadsheet world, where they have lots of rows and columns and can refer to them in calculations. Writing measures in DAX is a different experience because you don't get to "see" the data table in front of you when you're writing the code.

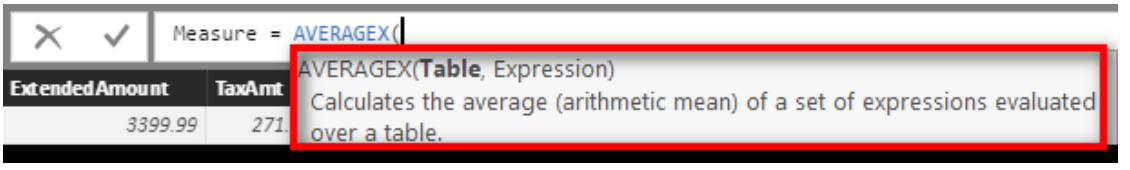
Instead, you have to visualise in your mind what you are doing. This is why I recommend creating a matrix to get immediate feedback after you write your formulas; it helps you visualise the result.

My number-one piece of advice for now is that *you shouldn't write calculated columns unless you have no other option and you know why you need them*. You'll learn more about calculated columns and when to use them in Chapter 8. Until then, you should assume that using a calculated column is not a good approach unless you know from experience what the exceptions are—and you will learn this with time and experience.

Are You Writing Your Measures Correctly?

It's time to practice using some new functions. Before you get into the following practice exercises, here is a refresher on the process you should use to write all your measures:

1. Create a new matrix on a new sheet.
2. Put Products[Category] on Rows.
3. Right-click the table where the measure will be stored and select New Measure.
4. Give the measure a descriptive name.
5. Start typing the function and pause so you can read the IntelliSense description of the function and the syntax, as shown below.



6. Immediately after writing the function, click back into the formula bar and apply any formatting you want to use.
7. Add the measure to your matrix so you can see the results.

Practice Exercises: AVERAGEX()

Set up a new matrix and put `Products[Category]` on Rows. Then write the following measures using `AVERAGEX()`. Find the solutions to these practice exercises in "Appendix A: Answers to Practice Exercises" on page 178. Don't worry about the logic of weighted averages in these exercises. These exercises are designed for simple practice, and you should ignore any real-world business logic.

28. [Average Sell Price per Item]

Find the column in the `Sales` table that gives the sell price per unit and use `AVERAGEX()` to find the average of this column.

29. [Average Tax Paid]

Find the tax column in the `Sales` table. Find the average of this column.

30. [Average Safety Stock]

There is a safety stock column in the `Products` table. You should put the results in a matrix, as shown below.

Category	Average Sell Price Per Item	Average Tax Paid	Average Safety Stock
Accessories	$19.42	$1.55	146
Bikes	$1,862.42	$148.99	100
Clothing	$37.33	$2.99	4
Components			500
Total	**$486.09**	**$38.89**	**283**

Note: There are other X-functions that are not included in this book, such as MAXX(), MINX(), COUNTAX(), and COUNTX(). You can find out how to use them by typing one of them at a time into the formula bar and reading the IntelliSense.

8: DAX Topic: Calculated Columns

It's time for a change of pace. I have deliberately left the main discussion of calculated columns until now to allow you to get accustomed to the power of measures. As mentioned in Chapter 7, the most common mistake I see Excel users make is to use too many calculated columns. And when you think about it, a calculated column is a very comfortable place for an Excel user to hang out because a table in Data view in Power BI looks and feels a lot like Excel. But as I warned previously, you should avoid using calculated columns until you know when and why to use them. Consciously avoiding calculated columns and trying to find a measure solution will make you a stronger DAX user. Trust me.

In general, you should not use a calculated column if:

- You can use a measure instead.
- You can bring the data into a table directly from the source data.
- You can create the column during data load by using Power Query.

I always recommend that you prefer the following order to source a missing column (when a measure will not do the job):

1. Get it added to the source and import it from there.
2. Create it in Power Query on data load.
3. Use a calculated column.

By pushing the column as far back to the source as possible, you increase the possibility of reuse down the track. But the truth is that this is a purist view, and it doesn't really matter that much. If you know how to do it in a calculated column and you don't know how to do it in Power Query, then there is no harm in using the calculated column. Indeed, you can and should use calculated columns when you need them. You should definitely use a calculated column when both of the following two conditions are satisfied at the same time:

- You need to filter/slice a visual based on the results of a column (i.e., you want to use the column on Filter, Slicer, Rows, or Columns). Measures don't work in this case.
- You can't bring the column of data you need in from your source data or by using Power Query (for whatever reason).

These are the most common reasons you can't get a column you need from your source data:

- It doesn't exist.
- You can't arrange to get it added (e.g., you don't have access to the source system).
- You can't get it added in a timely manner.
- You want to reuse measures that exist in your data model as part of the formula needed to create the new column.

As mentioned earlier, if possible, you should try to get the column you need added to the source data. When you do this, you get the full benefit of compression on data import; in addition, the column is available for reuse in all your future workbooks. But sometimes this simply isn't possible, and other times it is possible, but you can't wait two weeks (or two years!) to get it done. Calculated columns are useful in such cases. And if a new column becomes available in the future, you can simply delete your calculated column and replace it with the new column coming in from the source.

Here's How: Creating a Day Type Calculated Column

Let's look at an example of where you should use calculated columns. Let's say that you extract the `Calendar` table from your enterprise database, and you want a new column that shows whether each date is a weekend, but you can't arrange to have this column added for now. Of course, you could use Power Query, but this book is about DAX, so this section discusses how to create a calculated column to solve this problem.

Follow these steps to create a `Day Type` calculated column in the `Calendar` table:

1. Select the Calendar table from the fields list on the right-hand side of Power BI.
2. Right-click the table name and select New Column.

3. Immediately start typing the following over the top of `Column =` in the formula bar:

```
Day Type = IF('Calendar'[DayNumberOfWeek] = 1 || 'Calendar'[DayNumber-
OfWeek]=7,"Weekend","Weekday")
```

Note: Note the use of the two pipe symbols (||) in the formula above. The pipe can be found on your keyboard above the backslash key (which is right above the Enter/Return key). The two pipe symbols are the inline text version of a logical OR function.

4. You can also write an OR function in DAX as follows:

```
OR('Calendar'[DayNumberOfWeek] = 1, 'Calendar'[DayNumberOfWeek] = 7)
```

5. Personally, I prefer to use the two pipes because you can have as many of them as you like in a single formula. The OR() function above accepts only two parameters as inputs; if you have more than two "or" logical inputs, you need to use multiple nested OR() functions to make it work.

Note: The inline version of the logical AND is the double ampersand (&&), which equates to the AND() function.

6. If you are not currently in Data view, switch to it now and check to make sure your column is calculating correctly. You are the person writing the formulas, and hence you are personally responsible for making sure they are evaluating correctly.
7. Note that the formula you just created, as shown below, is a single formula for the entire column. Just as with Excel tables, with Power BI, it is not possible to have more than one formula in a calculated column. You therefore have to write the one formula so that it evaluates and handles all the possible scenarios you need.

Date	DayNumberOfWeek	DayName	Day Type
1/07/2001 12:00:00 AM	1	Sunday	Weekend
2/07/2001 12:00:00 AM	2	Monday	Weekday
3/07/2001 12:00:00 AM	3	Tuesday	Weekday
4/07/2001 12:00:00 AM	4	Wednesday	Weekday
5/07/2001 12:00:00 AM	5	Thursday	Weekday
6/07/2001 12:00:00 AM	6	Friday	Weekday
7/07/2001 12:00:00 AM	7	Saturday	Weekend
8/07/2001 12:00:00 AM	1	Sunday	Weekend
9/07/2001 12:00:00 AM	2	Monday	Weekday

8. Now that you have the new calculated column, go to a new page in your workbook and create a new matrix. Place `Products[Category]` on Rows, place your new column `'Calendar'[Day Type]` on Columns, and then add `[Total Sales]` to the Values section. You end up with the matrix shown below.

Category	Weekday	Weekend	**Total**
Accessories	$495,995	$204,764	**$700,760**
Bikes	$20,047,702	$8,270,442	**$28,318,145**
Clothing	$240,664	$99,109	**$339,773**
Total	**$20,784,362**	**$8,574,316**	**$29,358,677**

You have successfully extracted some new insights from the data that didn't exist before: You have used data modelling techniques to enhance the data for weekday/weekend analysis. Sweet!

Practice Exercise: Calculated Columns

Write the following calculated column in the `Calendar` table. Find the solution to this practice exercise on"Appendix A: Answers to Practice Exercises" on page 178.

31. Creating a Half-Year Column

Write a calculated column in the `Calendar` table that returns the value `H1` for the first half of each year (January through June) and `H2` for the second half of each year (July through December). *Hint:* You might want to use an `IF` statement to do this.

9: DAX Topic: CALCULATE()

CALCULATE() is the most important and powerful function in DAX. It is the only function that has the ability to modify the filter context coming from your visuals.

> **Note:** Actually, there is another function that can modify filter context: CALCULATETABLE(). This function is typically used inside DAX queries, though discussing it is beyond the scope of this book.

I am going to provide you with a solid understanding of how CALCULATE() works in this book, but you will need to continue to learn in the future; there is a lot to learn. If you want to be an expert, you need to read lots of other books and blogs to build on the foundation you get from this book.

> **Note:** You can find an up-to-date curated list of the best Power BI, Power Pivot, and Power Query books at the links provided in Chapter 21.

Altering the Standard Offering

Have you ever gone into a restaurant and looked at the menu, only to discover that the standard offering is not quite what you are after? Lots of people love Caesar salad, but many people do not like anchovies. Say that you're one of them, and you read the following on the menu:

Caesar Salad: Romaine lettuce, croutons, parmesan cheese, anchovies, and egg tossed in a creamy Caesar dressing.

When you order the salad, you alter the standard menu option and instead say, "I'll have the Caesar salad, no anchovies." CALCULATE() is a lot like that: It allows you to alter the standard offering (that you get from a visual rather than a menu) so you can get some variation that ends up being exactly what you want.

Technically speaking, CALCULATE() alters filter context. It modifies an expression (which can be a measure or another DAX formula) by applying/removing/modifying filters. The syntax of CALCULATE() is:

```
=CALCULATE(expression, filter 1, filter 2, filter n...)
```

CALCULATE() alters the filter context coming from the visual by applying none, one, or more filters prior to evaluating the expression. CALCULATE() "reruns" the built-in filter engine in Power BI—the one that makes the filters automatically propagate from the lookup tables and flow downhill to the data tables. When the filter engine is rerun by CALCULATE(), if there are any filters inside the CALCULATE() function, these filters become part of the filter context before the filter engine kicks in. (You'll find more about how this works in Chapter 10.)

> **Note:** I mentioned above that you can use none, one, or more filters inside CALCULATE(). It may seem strange that you can use none at all. Why would you want to do this? Using no filters is a special use case that is covered in Chapter 10.

Simple Filters

CALCULATE() can use two types of filters. A simple filter (or raw filter) has a column name on the left and a value on the right, as in these examples:

```
Customers[Gender] = "F"
Products[Color] = "Blue"
'Calendar'[Date] = "1/1/2002"
'Calendar'[Calendar Year] = 2003
```

You can use these simple filters as the second and subsequent parameters to CALCULATE() to alter the original meaning of an expression (which is the first parameter). Simple filters are really important in Power BI because they were designed to be easy to use and understand. Taking a filter from a lookup table and propagating it to the data tables is what the Power Pivot engine in Power BI was built and optimised to do.

Note: Under the hood, Power BI converts a simple filter into a much more complex formula that is harder for beginners to learn and understand. Take the following measure, using a simple filter:

```
Total Sales to Females
       = CALCULATE([Total Sales], Customers[Gender] = "F")
```

Under the hood, Power BI converts this into the following formula prior to execution:

```
Total Sales to Females
     = CALCULATE([Total Sales],
            FILTER(ALL(Customers[Gender]),
               Customers[Gender] = "F"
            )
       )
```

I am sure you will agree that the first formula is easier to read and understand. The Microsoft developers call this type of simple syntax "syntax sugar." The simple syntax is provided to allow beginners to use Power BI without having to first become DAX experts.

I cover the ALL() and FILTER() functions later in this book.

To see `CALCULATE()` in action (using the simple syntax), set up a new matrix like the one below, with `Products[Category]` on Rows and `[Total Sales]` on Values. (You should be getting used to this by now!)

Category	Total Sales
Accessories	$700,760
Bikes	$28,318,145
Clothing	$339,773
Total	**$29,358,677**

Then write the following measure:

```
Total Sales of Blue Products
    = CALCULATE([Total Sales], Products[Color]="Blue")
```

In the image below, can you see how the simple filter used here, `Products[Color]="Blue"`, has altered the initial filter context coming from the matrix and given a variation to the regular measure `[Total Sales]`? It is as if you have changed the recipe for the standard product on the menu and instead received a variation of that regular menu item. Think Caesar salad without anchovies.

Category	Total Sales	Total Sales of Blue Products
Accessories	$700,760	$74,354
Bikes	$28,318,145	$2,169,056
Clothing	$339,773	$35,687
Total	**$29,358,677**	**$2,279,096**

Practice Exercises: CALCULATE() with a Single Table

It's time for you to write some simple `CALCULATE()` examples that filter a single table. Set up a new matrix with `Customers[Occupation]` on Rows and `[Total Number of Customers]` on Values. You should have the matrix shown below as your starting point.

Occupation	Total Number of Customers
Clerical	2,928
Management	3,075
Manual	2,384
Professional	5,520
Skilled Manual	4,577
Total	**18,484**

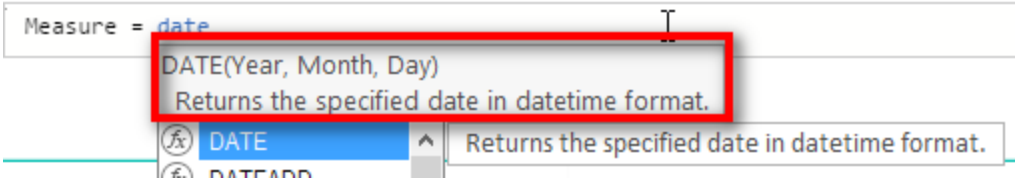
Then write the following measures, using CALCULATE(). Find the solutions to these practice exercises in"Appendix A: Answers to Practice Exercises" on page 178.

32. [Total Male Customers]

Write a new measure that modifies the `[Total Number of Customers]` measure you wrote previously to come up with a total for male customers only. You need to look for a suitable column from the `Customers` table to use in your filter.

33. [Total Customers Born Before 1950]

In this case, you need to enter the date < January 1, 1950, into the formula as the filter parameter. You need to use the `DATE()` function to be able to refer to a date. Remember that you can get help from the tooltips that IntelliSense provides when writing the measure. Just start typing `=DATE` inside the formula bar, and a tooltip pops up, explaining the purpose and syntax of the function, as shown below.



Now that you know how to write a date inside a formula, you can go ahead and write the measure `[Total Customers Born Before 1950]`.

34. [Total Customers Born in January]

This exercise is similar to Practice Exercise 33, but this time you need to use the `MONTH()` function to turn the information in the `Customers[BirthDate]` column into a month.

35. [Customers Earning at Least $100,000 per Year]

Write a measure that counts the number of customers who earn more than $100,000 per year. The following matrix shows what you should end up with. Look for a suitable column to use for the filter in the `Customers` table.

Occupation	Total Number of Customers	Total Male Customers	Total Customers Born Before 1950	Total Customers Born in January	Customers Earning at Least $100,000 per Year
Clerical	2,928	1,488	433	132	
Manual	2,384	1,251	134	128	
Skilled Manual	4,577	2,293	234	192	
Professional	5,520	2,727	609	254	792
Management	3,075	1,592	1,543	136	1,406
Total	**18,484**	**9,351**	**2,953**	**842**	**2,198**

Using CALCULATE() over Multiple Tables

In Practice Exercises 32–35 above, the `CALCULATE()` function touches only a single table; the filtering is applied to a table, and the expression is evaluated on the same table. However, `CALCULATE()` can work over multiple tables, too. When you use the `CALCULATE()` function, it first applies the filters to the relevant tables, and then it reruns the filter propagation engine and makes sure that any new filters inside the `CALCULATE()` function automatically propagate from the "one" side of the relationship to the "many" side (i.e., the filters flow downhill) before the expression is evaluated. So you can apply a filter to one or more of the lookup tables, and these filters will propagate to the data tables, and any expression that evaluates over the connected data tables will reflect the filters from the lookup tables. Are you feeling supercharged now?!

Practice Exercises: CALCULATE() with Multiple Tables

Set up a new matrix. Put `Territories[Region]` on Rows and `[Total Sales]` on Values. Note that there are now two tables involved. The initial filter context is coming from the `Territories` table (see #1 here), and the calculation `[Total Sales]` is operating over the `Sales` table (#2).

With your matrix set up as described above, write the following new measures. Find the solutions to these practice exercises in"Appendix A: Answers to Practice Exercises" on page 178.

Region	Total Sales
Australia	$9,061,001
Canada	$1,977,845
Central	$3,001
France	$2,644,018
Germany	$2,894,312
Northeast	$6,532

36. [Total Sales of Clothing]

Use the Products[Category] column in your simple filter. The filter gets applied to the lookup table, but then the measure [Total Sales] is modified by the filter (and `Total Sales` comes from the Sales table), so you use CALCULATE() with multiple tables in this formula.

37. [Sales to Female Customers]

As the name of this measure suggests, you use `CALCULATE()` to modify the standard calculated field `[Total Sales]` and create a new measure that is for sales to female customers.

38. [Sales of Bikes to Married Men]

You need to use multiple filters on two tables for this one. `CALCULATE()` can accept as many filters as you pass to it. Just separate the filters with commas.

When you have finished these three practice exercises, you should have a matrix like the one shown below.

Region	Total Sales	Total Sales of Clothing	Sales to Female Customers	Sales of Bikes to Married Men
Australia	$9,061,001	$70,260	$4,634,993	$2,205,159
Canada	$1,977,845	$53,165	$1,011,320	$517,808
Central	$3,001	$157	$124	
France	$2,644,018	$27,035	$1,271,964	$726,649
Germany	$2,894,312	$23,565	$1,539,713	$694,776
Northeast	$6,532	$106	$3,836	$2,295
Northwest	$3,649,867	$58,230	$1,843,586	$982,266
Southeast	$12,239	$301	$11,938	
Southwest	$5,718,151	$74,714	$2,881,098	$1,451,036
United Kingdom	$3,391,712	$32,240	$1,615,046	$1,031,765
Total	**$29,358,677**	**$339,773**	**$14,813,619**	**$7,611,754**

Advanced Filters

So far you have used only simple filters inside `CALCULATE()`, in this format:

```
TableName[ColumnName] = some value
```

You can also use a more advanced filter that is passed in the form of a table containing the values required for the filter. This table can be either of the following:

- A physical table
- A function that returns a table (e.g., ALL(), VALUES(), FILTER())

Importantly, both types of tables used as advanced filter parameters *retain all relationships that exist in the data model*. Advanced filters and the way the tables retain their relationships in the data model is a complex topic that is covered in more detail in the coming chapters. For now, it is enough to know that so far you have only learnt about simple filters for `CALCULATE()`, and the advanced table filters are coming later.

Making DAX Easy to Read

Now is a good time to pause and talk about how to lay out your DAX so it is easy to read. Consider this example:

```
Total Sales to Single Males in Australia =
CALCULATE([Total Sales], Customers[MaritalStatus]="S",
Customers[Gender]="M", Territories[Country]="Australia")
```

Formulas like this one that are very long can be difficult to read. The generally accepted approach is to lay out a formula by using line breaks and spaces so it is easier to see which parts of the formula belong together. There is no single right way to do this. Here is one way that I find useful:

```
Total Sales to Single Males in Australia
    = CALCULATE( [Total Sales],
        Customers[MaritalStatus] = "S",
        Customers[Gender] = "M",
        Territories[Country] = "Australia"
      )
```

To create a new line in the formula dialog box, you need to press Shift+Enter on the keyboard. Then you can use the Tab key to create indented space from the left-hand side.

In the example above, I put the first parameter in `CALCULATE()` (which is the expression) on the first line, followed by a comma. Then I placed each filter on a new line and indented them so it is easy to see that they belong to the `CALCULATE()` function.

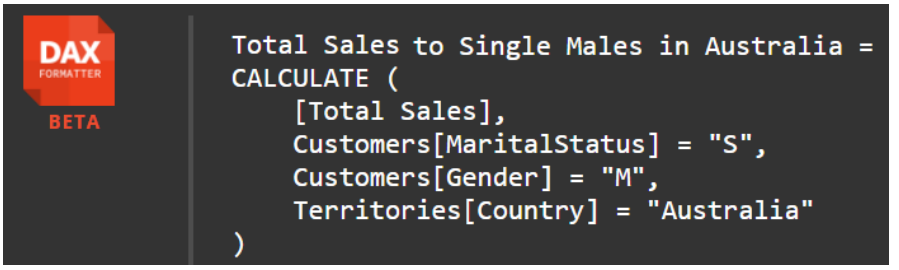
The final closing parenthesis for the `CALCULATE()` function is on a new line of its own, aligned with the `C` in `CALCULATE()` so that I know that this bracket closes the `CALCULATE()` function.

Using DAX Formatter

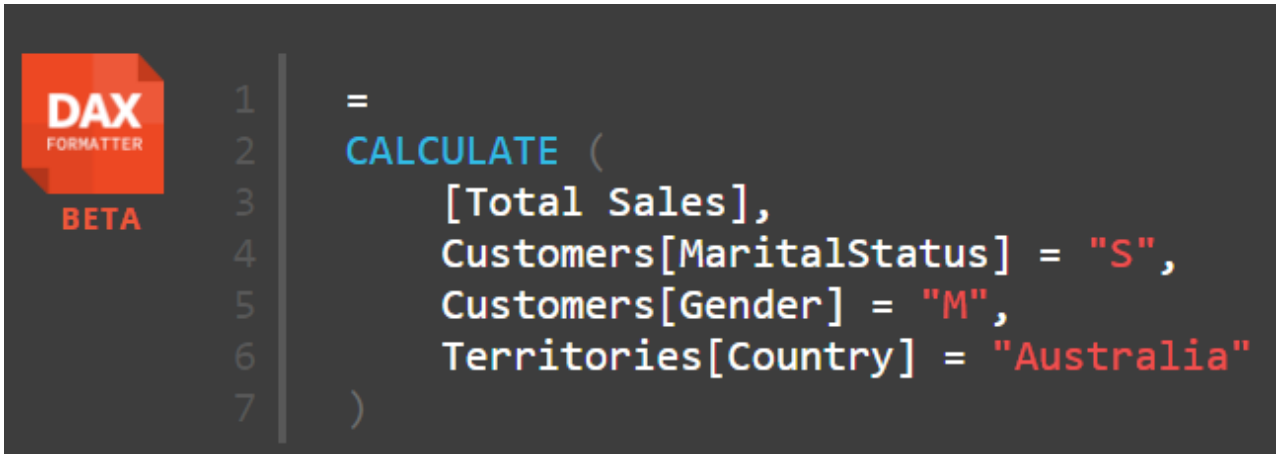
DAX Formatter is a very useful (and free) tool that you can use to help format your DAX. Marco Russo and Alberto Ferrari from SQLBI developed the free http://daxformatter.com website. You simply paste your DAX code into the website, and DAX Formatter formats the code for you. You can then cut and paste it back into the formula bar in Power BI.

When you use DAX Formatter, you have a choice about whether to include the measure name. You can omit the name if you like, but it is just as easy to copy the entire formula, including the measure name.

The image below shows the `[Total Sales to Single Males in Australia]` formula from DAX Formatter, including the measure name.



The image below shows the same formula without the measure name after DAX formatter has applied formatting.

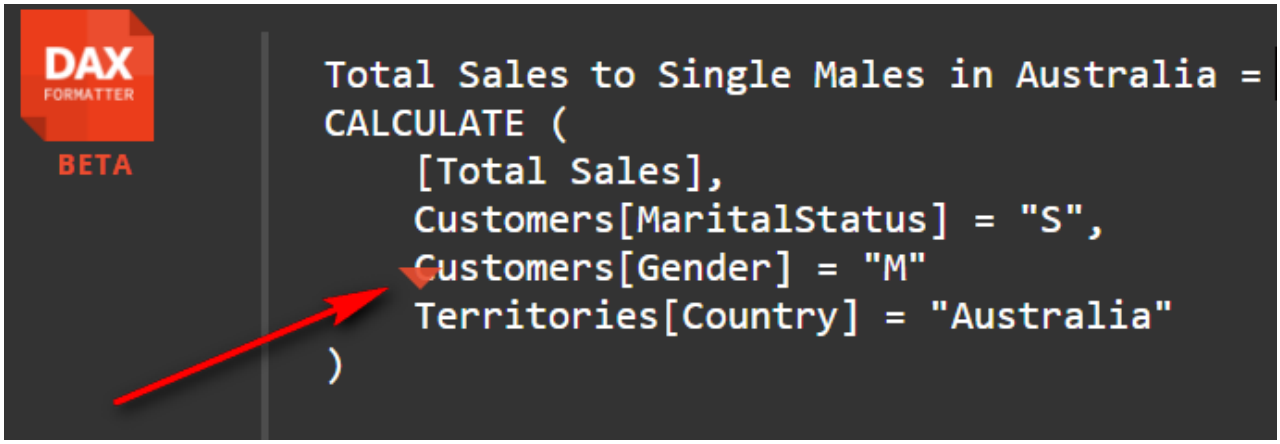


Error Checking

DAX Formatter does another important job for you: It checks whether your DAX formula is valid and written correctly. If it is not, DAX Formatter does its best to show you where any errors are located. To see this in action, try removing one of the commas from the DAX code above, such as the one after `Gender = "M"`, so it looks like this:

```
Total Sales to Single Males in Australia
= CALCULATE( [Total Sales],
    Customers[MaritalStatus] = "S",
    Customers[Gender] = "M"
    Territories[Country] = "Australia"
)
```

When you put this erroneous code into DAX Formatter, a triangle points to the part of the code where an unexpected value is found, as shown below. In this case, DAX Formatter is expecting a comma but instead finds the letter `T`—the start of the column name.



DAX Formatter is a great tool for helping you debug your DAX code when you can't work out what is wrong (although it was not strictly designed to do this). I use DAX Formatter all the time to help me with my DAX. I suggest you do, too.

> **Note:** Marco Russo told me that DAX Formatter was never designed to be an official error-checking tool, as I describe using it above. As a result, the error checking is not perfect, but in my view, it is still worth trying this approach if you are stuck.

10: Concept: Evaluation Context and Context Transition

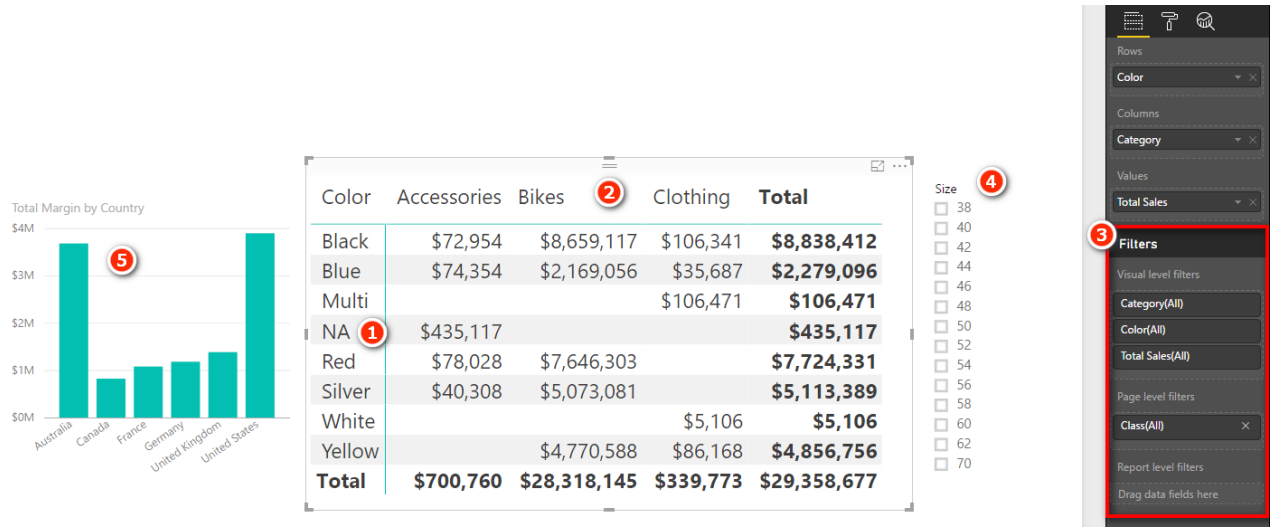
This chapter covers one of the hardest topics to understand and master in DAX. As you saw in Chapter 5, some DAX functions have what is called a *filter context*. As you saw in Chapter 7, DAX also has a *row context*. Filter context and row context are the two different types of *evaluation context,* the topic of this chapter.

A Refresher on Filter Context

In Chapter 5 I introduced a number of concepts, including *filter context* and *initial filter context*. Here is a quick refresher.

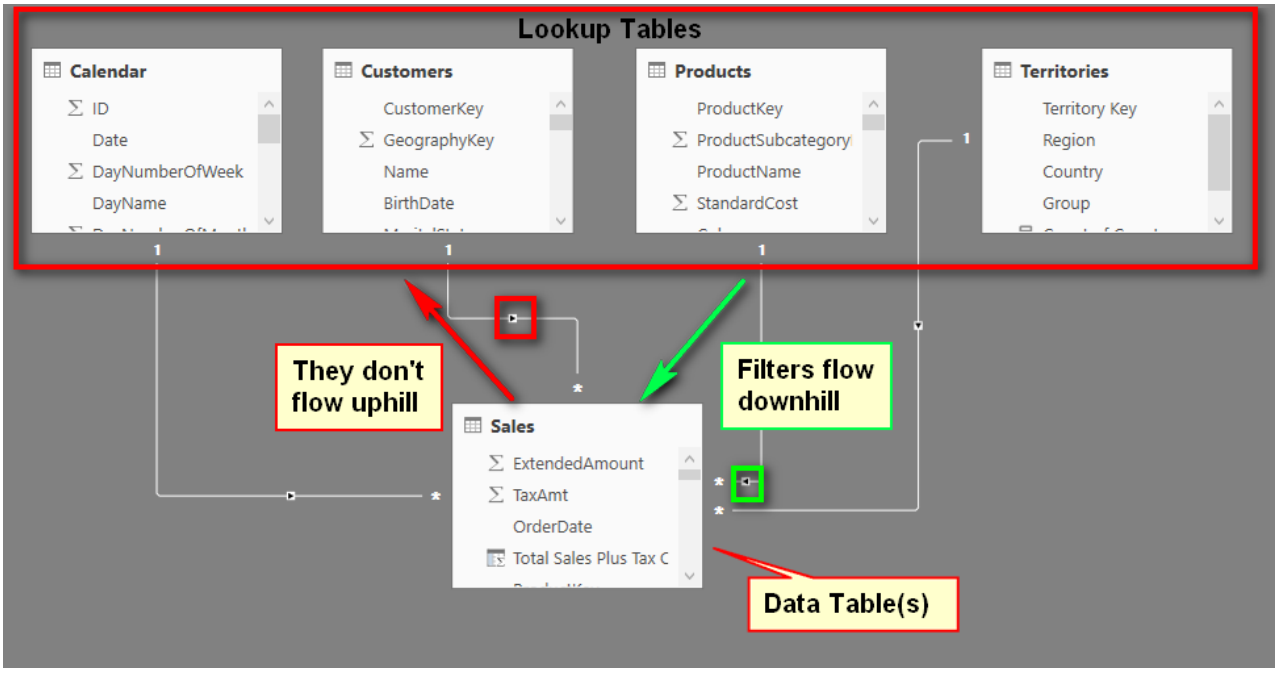
Filter context refers to any filtering that is applied to the data model in DAX. Filter context is created by a visual (e.g., a matrix) and also by the `CALCULATE()` function. The *initial filter context* is the natural filtering that is applied by a visual. The initial filter context can come from the following areas on a report:

- Rows (see #1 below)
- Columns (#2)
- Filters (#3)
- Slicers (#4)
- Any other visual on the canvas (e.g., a column chart, as shown in #5)



Don't confuse the filter context coming from Rows in a matrix with row context. These are two completely different things. Filter context is the natural "slicing" that comes from the coordinates of a matrix or another visual. The Rows section in a matrix is one of the locations that can slice your data and therefore is part of the initial filter context.

The filter context coming from a report filters the underlying tables in the data model. If the tables are joined, the filters propagate from the "one" side of the relationship to the "many" side of the relationship. But the filter does not propagate from the "many" side of the relationship to the "one" side. This is why I recommend (for Excel users anyway) laying out the data model using what I called the Collie layout methodology, described in Chapter 2 and shown again below. When you use this layout, you have less to get your head around, and you can simply visualise filter propagation flowing downhill like water.



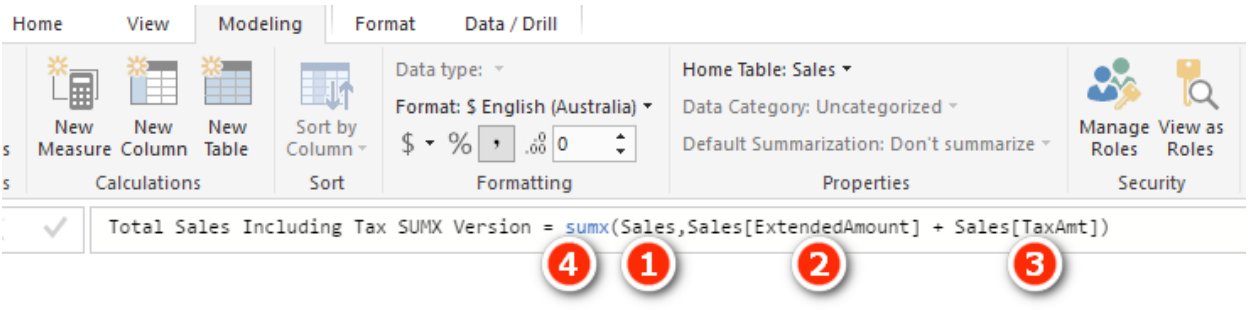
This setup provides a visual clue that the filters flow downhill through the relationships but do not flow uphill through the relationships.

> **Note:** It is possible to filter a lookup table (sitting above) based on the results in a data table (sitting below) by using DAX instead of filter propagation. But filters only ever automatically propagate through the relationships downhill.

A Refresher on Row Context

Row context refers to the ability of a special iterating function or calculated column to be "aware" of which row it is acting on at each stage of formula evaluation. Some functions (e.g., `FILTER()`, the X-functions) and all calculated columns have row context. When you think about row context, think of the function (or calculated column) iterating through the table one row at a time and *selecting* the single value (the intersection between the column and row) and then acting on that single value. Regular measures can't do this; only functions that have row context and calculated columns can perform this trick.

Let's look again at the `SUMX()` measure from Chapter 7, shown below.



`SUMX()` first creates a row context over the `Sales` table (see #1 above). It then iterates through this table one row at a time. At each row, it takes the single value that is the intersection of the `Sales[ExtendedAmount]` column and the current row (#2) and adds it to the single value that is the intersection of the `Sales[TaxAmt]` column and the current row (#3). It does this for each row in the table (#1) and then adds up all the values (#4).

Note: One thing I often get asked at this point is "Why does the formula refer to the table name twice—once in the first parameter and again inside the second parameter?" The main reason is that the table names inside the second parameter are actually a fully qualified address of the column. It is possible to have two columns with the same name in two different tables. You must specify the table name first (*TableName*`[`*ColumnName*`]`); otherwise, you may get the wrong answer from the wrong column (same column name, different table). Also, it is possible that the table in the first parameter could be (and often is) different to the tables where the columns come from in the second parameter.

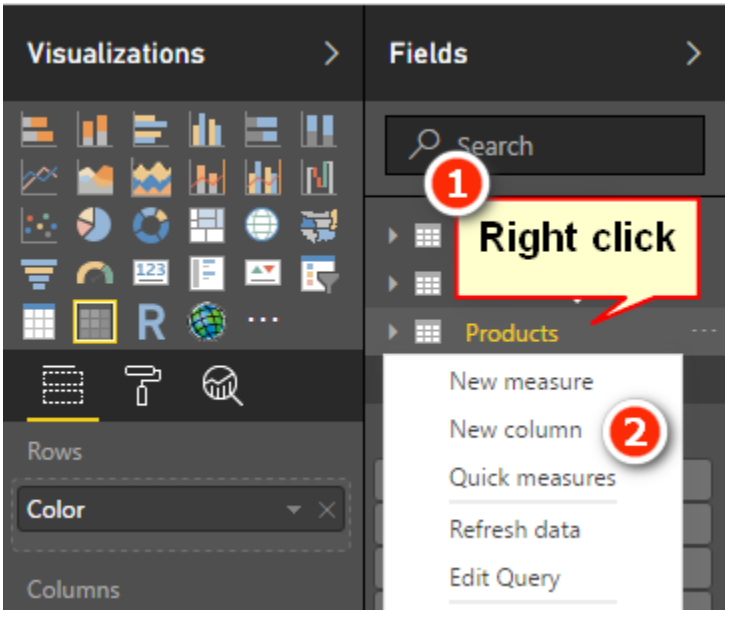
In short, think of the first parameter as the name of the table to iterate over and think of the references to the table name in the second parameter, *TableName*`[`*ColumnName*`]`, as being the fully qualified address of the column you are using.

Row Context in Calculated Columns

You already know that iterator functions and also calculated columns have row context. The main difference between an iterator function (e.g., `SUMX()`) and a calculated column is that the calculated column stores the value calculated at each row of the iteration process in the column itself. Measures do not do this. In the `SUMX()` example, the function returns the final result to the visual in your report without storing all the intermediate values (beyond the need to temporarily keep track of them during the calculation process). This is the main reason you should avoid using calculated columns, if possible: They take up storage space in your data model and hence make your files larger and generally slower.

Understanding That a Row Context Does Not Automatically Create a Filter Context

This is a very important point that you must understand clearly: A row context does not automatically create a filter context. Also, a row context does not follow relationships. To understand this better, right-click the `Products` table in the fields list (see #1 below) and select New Column (#2).



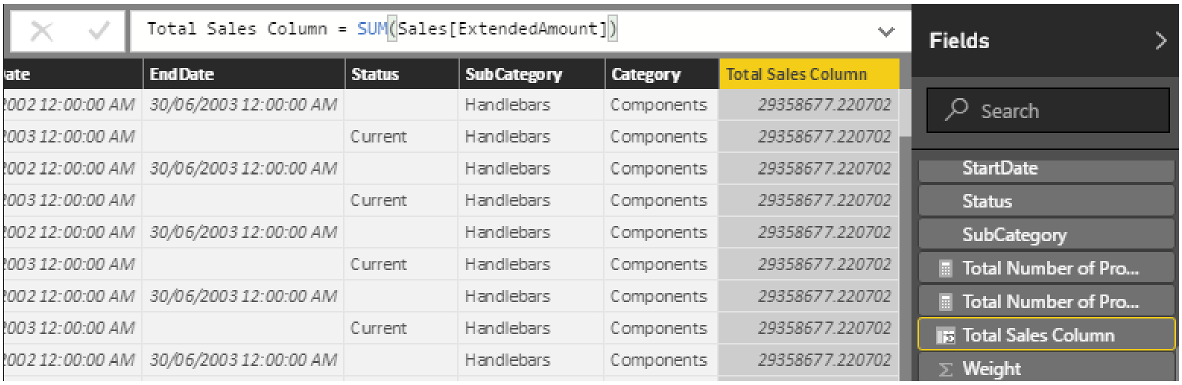
Add the following calculated column to the table but don't press Enter yet:

```
Total Sales Column = SUM(Sales[ExtendedAmount])
```

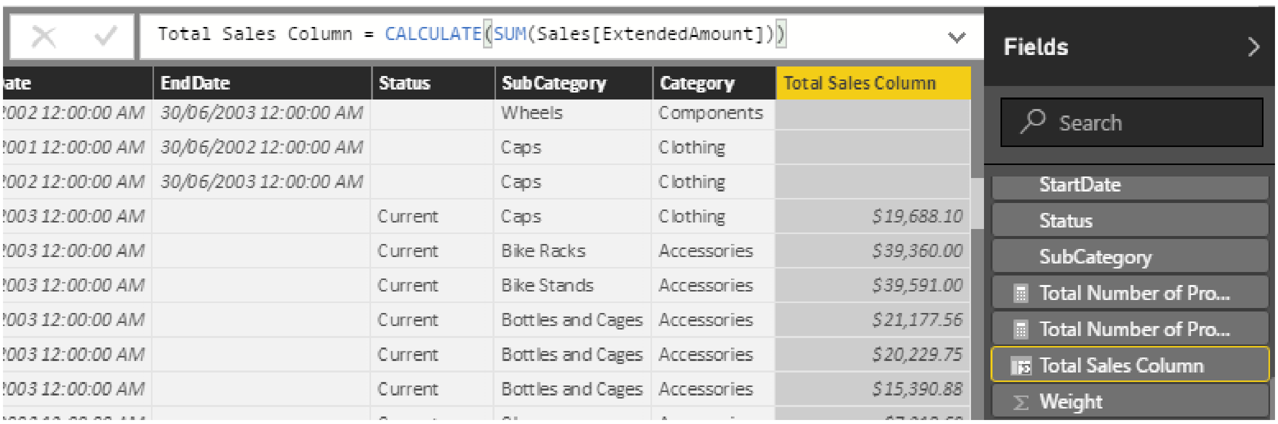
You should recognise the right side of this formula because it is exactly the same as the first formula in this book.

What value do you think will appear in every row of this new column? Do you expect it to be the total for the product in each row? Do you expect it to be the total for all products? Well, the answer may surprise you, and it is directly related to the point that a row context does not automatically create a filter context. After pressing Enter, if you are not already in Data view, you need to switch to Data view to see the column.

As you can see in the figure below, the value is the same for every single row in the table. There is no filtering on the `Sales` table (or any other table, for that matter) as a result of this formula, and hence the answer is always the same for every row. There is a row context in this formula: The rows are evaluated one at a time. But that row context does not create a filter context. Given that there is no filter context, the `Sales` table is completely unfiltered, and hence `SUM(Sales[ExtendedAmount])` must return the result of the unfiltered `Sales` table.



It is, however, possible to turn the row context from this calculated column into a filter context through a process called *context transition*. To do this, simply wrap the above formula in a `CALCULATE()` function, as shown below.



When you do this, the row context that exists in the calculated column is transformed into an equivalent filter context. The `CALCULATE()` function then "pulls the handle" on the filter engine so the filter on the `Products` table propagates through the relationship to the `Sales` table *before* the calculation is completed (for each row in the table). You therefore potentially end up with a different value in each row of the column. The value is actually the total sales for the product in each row (and some rows are blank because there are no sales).

You can think of the formula working like this:

```
= CALCULATE(SUM(Sales[ExtendedAmount]),
     Products[ProductKey] = the product represented by this row in the
table
)
```

The concept of context transition works anywhere that a row context exists—that is, in calculated columns as well as in iterators like `FILTER()` and `SUMX()`. This is the special use case mentioned in Chapter 9, where there are no filters at all needed inside `CALCULATE()` but instead `CALCULATE()` creates a new filter context from the row context by using context transition. You can add additional filters inside `CALCULATE()`, too, if you want or need to, but none are required.

The Hidden Implicit CALCULATE()

Now that you know that you can use `CALCULATE()` to convert a row context into a filter context, there is one more thing you need to know. Consider this formula from earlier in the book:

```
Total Sales = SUM(Sales[ExtendedAmount])
```

Now think back to what you read on the previous page. What happened when we added a new column in the `Products` table, as follows?

```
Total Sales Column =SUM(Sales[ExtendedAmount])
```

Do you remember? We got the value $29.3 million all the way down the new column in the `Products` table. Why? Because there is a row context in a calculated column, but there is no filter context. The `Sales` table is therefore completely unfiltered, and hence `SUM(Sales[ExtendedAmount])` simply must return $29.3 million for every row.

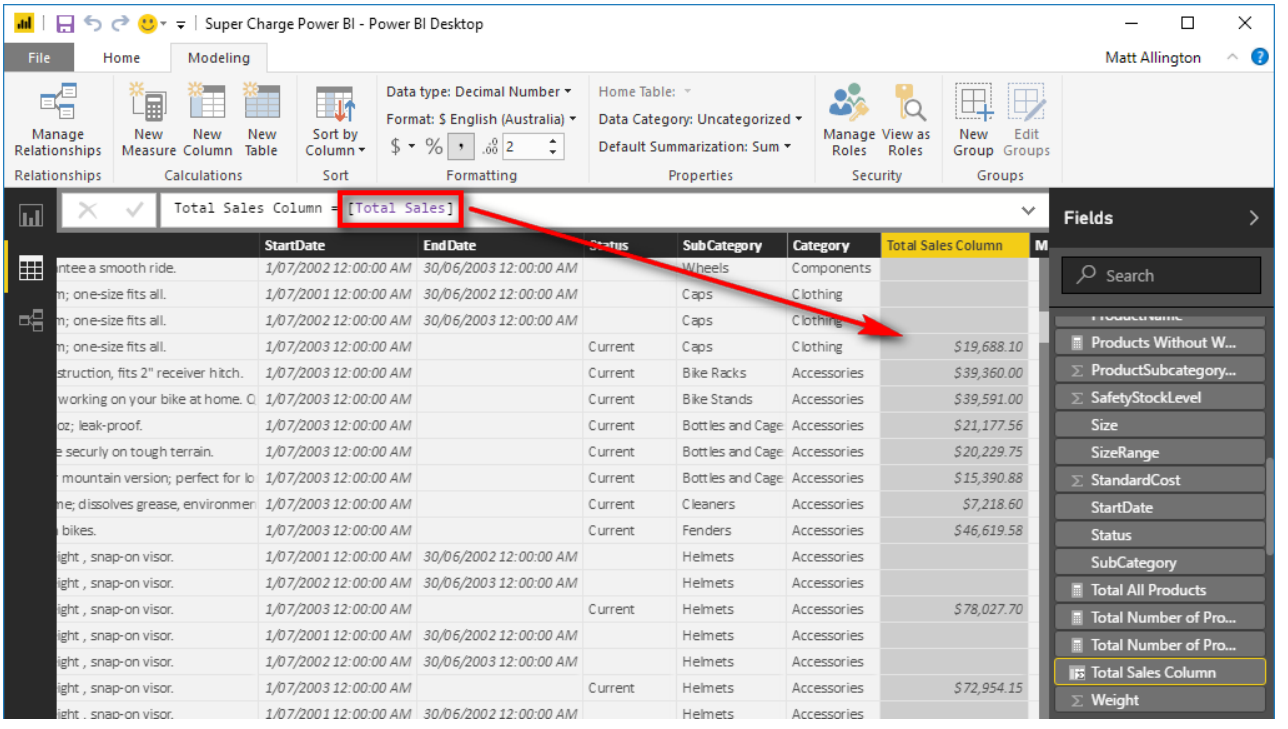
Now back to this measure:

```
Total Sales = SUM(Sales[ExtendedAmount])
```

Notice that the formula for this measure is identical to the calculated column (the first example above but with a different name, of course). So if the formula inside the measure is identical to the formula in the calculated column, you can be excused for thinking that you could substitute the formula in the calculated column with the actual measure, adjusted as shown below:

```
Total Sales Column = [Total Sales]
```

If the measure `[Total Sales]` has the same formula, won't we get the same result? Well, actually no; we get a different result from before, as shown below.



Go back and look again. Here is a summary of what you will find:

```
Total Sales Column 1 = SUM(Sales[ExtendedAmount])
```

This calculated column will return $29.3 million all the way down the column. There is a row context but no filter context, so the formula must return $29.3 million for each row in the table.

This next calculated column will return the total sales for each product in the products table (a different number for each product):

```
Total Sales Column 2 = CALCULATE(SUM(Sales[ExtendedAmount]))
```

There is a row context, and because of the `CALCULATE()` function, the row context is converted to an equivalent filter context through the process of context transition. `CALCULATE()` converts the row context from

the Products table into an equivalent filter context, and this new filter context propagates to the Sales table for each row of the calculated column.

The following calculated column also returns the total sales for each product in the Products table:

```
Total Sales Column 3 = [Total Sales]
```

This calculated column returns exactly the same result as Total Sales Column 2. If you take a look inside the measure [Total Sales], you can't actually see a CALCULATE() function; you didn't include CALCULATE() in the measure. But there is an implicit CALCULATE() there that you can't see. Every measure has an implicit CALCULATE(), and that is why this calculated column behaves like column 2 and not like column 1.

> **Note:** There is a lot to learn about context transition that is more advanced and beyond the scope of this book. Any book by Marco Russo and Alberto Ferrari that covers context transition would be a great learning resource. There are also some great videos available at http://sqlbi.com and several articles on my blog, http://xbi.com.au/blog, on the topic. I provide links to these and many other resources in Chapter 21.

If you've gotten to this point and don't fully understand context transition, don't worry, you are not alone. This is one of the hardest topics to learn and understand well. Sleep on it for a few nights, do some practice, and then come back and reread this chapter again (along with Chapter 9, on CALCULATE()). You may need to reread this content many times before it completely sinks in.

11: DAX Topic: IF(), SWITCH(), and FIND()

DAX has a number of useful functions that allow you to apply a test and then branch the formula based on the results from that test. You will most likely be familiar with this concept from the `IF()` function in Excel.

The IF() Function

The `IF()` function in DAX is almost identical to `IF()` in Excel:

```
IF(Logical Test, Result if True, [Result if False])
```

Note that the last parameter, [*Result if False*], is optional. If you omit this parameter and the result is `FALSE`, the `IF()` formula returns `BLANK()`. This is very useful because a matrix or chart in Power BI will not show values if the result in the Values section is `BLANK()`.

The SWITCH() Function

The `SWITCH()` function is a lot like `Select Case` in VBA programming. The syntax of a `SWITCH()` function is as follows:

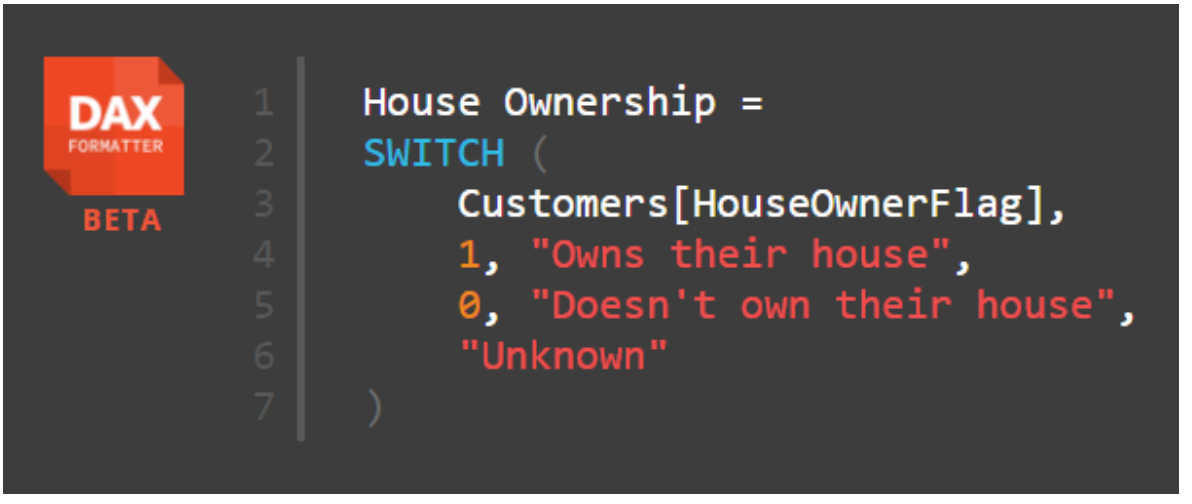
```
SWITCH(expression, value, result[, value, result]...[, else])
```

This syntax is a little confusing, so let's go through a simple example with another calculated column.

Right-click on the `Customers` table, select New Column, and enter the following formula:

```
House Ownership = SWITCH(Customers[HouseOwnerFlag],1,"Owns their
house",0,"Doesn't own their house","Unknown")
```

It is much easier to understand `SWITCH()` if you use DAX Formatter to improve the layout, as shown below. You can see in this figure that line 3 is the branching point. The possible values in the HouseOwnerFlag column are `0` and `1` in this instance. Lines 4 and 5 offer up pairs of input and output values. So if the value of `HouseOwnerFlag` is `1`, then the result `"Owns their house"` is returned. If the value of `HouseOwnerFlag` is `0`, then the result `"Doesn't own their house"` is returned.



Line 6 is a single value, and it applies to all other possible values of `HouseOwnerFlag` (although there are none in this example).

> **Note:** There is a much more exciting use of the `SWITCH()` function later in the book. See "The SWITCH() Function Revisited" on page 144 in Chapter 17.

The FIND() Function

The `FIND()` function in DAX is almost identical to the `FIND()` function in Excel. In DAX, it has this format:

```
= FIND (FindText, WithinText,[StartPos],[NotFoundValue])
```

Even though this syntax suggests that *StartPos* and *NotFoundValue* are optional, in my experience (as of this writing), you actually do need to provide values for these parameters.

An Example Using IF() and FIND()

This example shows how to create a calculated column on one of your lookup tables. As I have said previously, it is perfectly valid to create calculated columns in lookup tables, but wherever possible, it is better practice to create these columns in your source data or using Power Query. Remember why this is important:

- Calculated columns may take up more space than imported columns (but this is generally not a major issue for lookup tables).
- If you are manually creating a calculated column, it exists only in that single workbook, and you need to re-create it over and over for every other workbook where you need the column.

If it is not possible to create the calculated column in your source data for some reason, then creating a calculated column instead is a great solution, particularly for lookup tables.

In this example, you are going to create a `Mountain Products` column that doesn't exist in your lookup table. Any product with the word *mountain* in the description will be flagged as a mountain product.

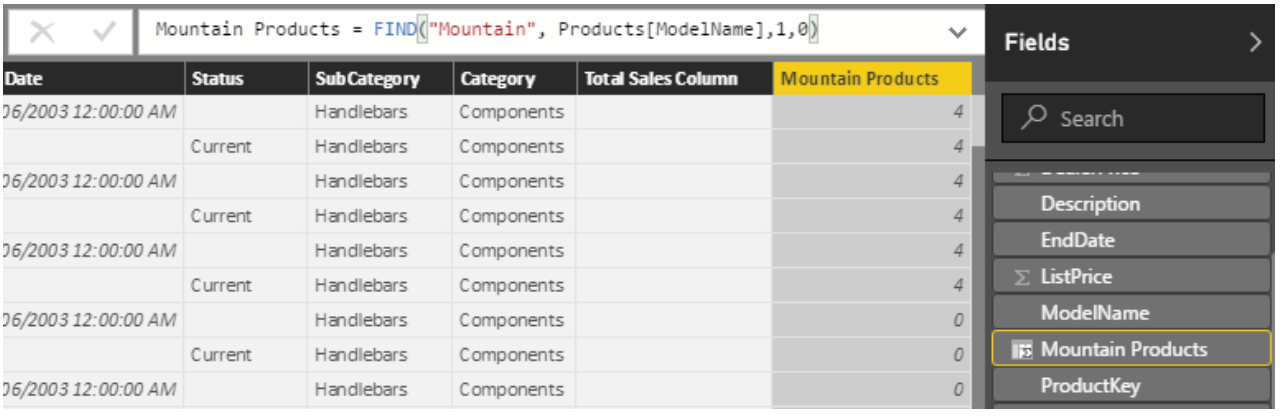
Right-click the `Products` table in the fields list, select New Column, and type the following formula:

```
Mountain Products = FIND("Mountain", Products[ModelName] ,1,0)
```

Switch to Data view so you can check the results of your new column. (Remember that it is your responsibility to ensure that the formulas are working as expected.)

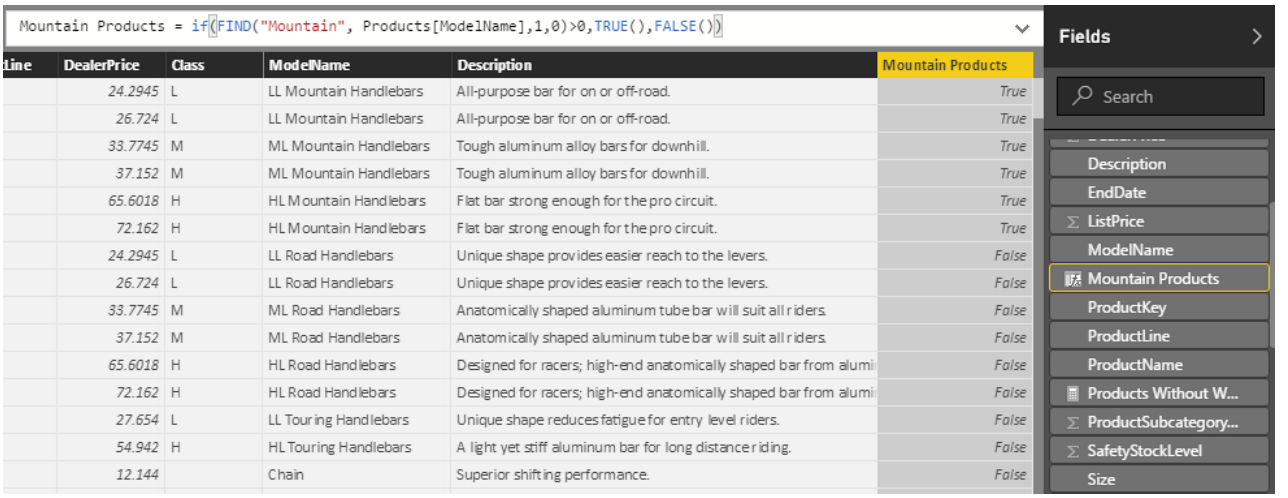
This formula searches for the word *mountain* in the `ModelName` column. Remember that because a calculated column has a row context, it is possible to refer to the column in this way, and it will calculate a result for every row in the `Products` table and store the answer in the column.

The result is an integer representing the starting position where the word *mountain* is found. If the word *mountain* is not found, then the value `0` (the last parameter in the formula) will be returned. So you get something like the table shown below.



Date	Status	SubCategory	Category	Total Sales Column	Mountain Products
06/2003 12:00:00 AM		Handlebars	Components		4
	Current	Handlebars	Components		4
06/2003 12:00:00 AM		Handlebars	Components		4
	Current	Handlebars	Components		4
06/2003 12:00:00 AM		Handlebars	Components		4
	Current	Handlebars	Components		4
06/2003 12:00:00 AM		Handlebars	Components		0
	Current	Handlebars	Components		0
06/2003 12:00:00 AM		Handlebars	Components		0

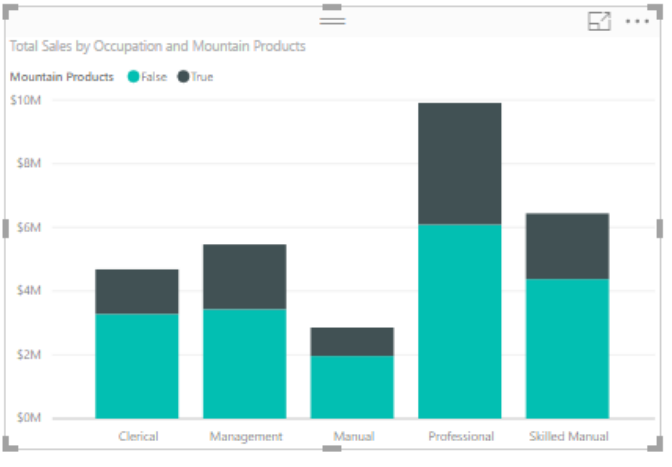
This table is not overly useful as is, but you can wrap an `IF()` statement around this formula to make it more useful. As shown below, you can use the `IF()` statement to return `TRUE` if the number is greater than zero (i.e., if it is a mountain product) and return `FALSE` if it is equal to zero (i.e., it is not a mountain product).

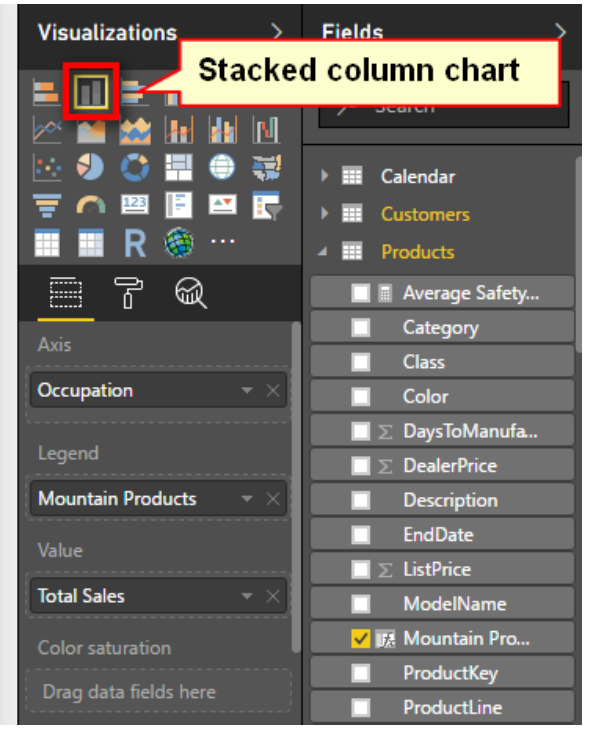


Line	DealerPrice	Class	ModelName	Description	Mountain Products
	24.2945	L	LL Mountain Handlebars	All-purpose bar for on or off-road.	True
	26.724	L	LL Mountain Handlebars	All-purpose bar for on or off-road.	True
	33.7745	M	ML Mountain Handlebars	Tough aluminum alloy bars for downhill.	True
	37.152	M	ML Mountain Handlebars	Tough aluminum alloy bars for downhill.	True
	65.6018	H	HL Mountain Handlebars	Flat bar strong enough for the pro circuit.	True
	72.162	H	HL Mountain Handlebars	Flat bar strong enough for the pro circuit.	True
	24.2945	L	LL Road Handlebars	Unique shape provides easier reach to the levers.	False
	26.724	L	LL Road Handlebars	Unique shape provides easier reach to the levers.	False
	33.7745	M	ML Road Handlebars	Anatomically shaped aluminum tube bar will suit all riders.	False
	37.152	M	ML Road Handlebars	Anatomically shaped aluminum tube bar will suit all riders.	False
	65.6018	H	HL Road Handlebars	Designed for racers; high-end anatomically shaped bar from alumi	False
	72.162	H	HL Road Handlebars	Designed for racers; high-end anatomically shaped bar from alumi	False
	27.654	L	LL Touring Handlebars	Unique shape reduces fatigue for entry level riders.	False
	54.942	H	HL Touring Handlebars	A light yet stiff aluminum bar for long distance riding.	False
	12.144		Chain	Superior shifting performance.	False

Now you have a new calculated column, and you have further enhanced your data model to be more useful. Remember that this calculated column will take up space in your file and disk. However, given the small number of unique values (only `True` and `False` in this case) and the fact that this column is in a lookup table, this column won't take up much space. The greater the number of unique values in a column, the more disk space and memory the column will consume.

You can now use this new column anywhere in your Power BI report to produce new insights that weren't previously visible in the data. Because this formula is a column in the data model, it can be used to filter a matrix, or it can be used on an axis in a chart, as shown below.





12: DAX Topic: VALUES(), HASONEVALUE(), SELECTEDVALUE(), and CONCATENATEX()

In Chapter 9 I introduced the idea that CALCULATE() can use two types of filters: simple filters and advanced filters. Simple filters are in this form:

```
TableName[ColumnName] = some value
```

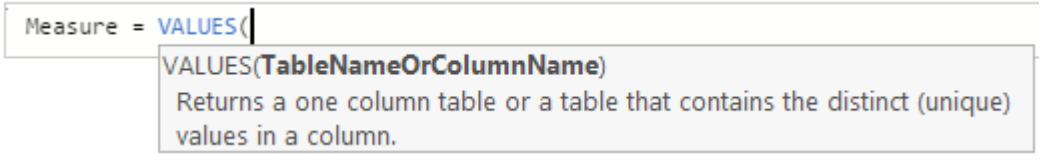
On the other hand, an advanced filter takes a table as a filter input. In other words, you use an existing table (or create a virtual table on-the-fly) that contains the rows you want included in the filter, and `CALCULATE()` applies that filter to the data model before completing the evaluation of the main expression.

Creating Virtual Tables

The tables you create using functions can be thought of as being "virtual" because they are not physically stored as part of the data model. They are created on-the-fly inside your DAX formulas and can be used during the evaluation of just the specific formulas containing the virtual table. Importantly, when you create a virtual table using a formula, the new virtual table will have a virtual relationship to the data model, and that virtual relationship will propagate the filter context in exactly the same way that the permanent relationships do. (You'll learn more about this later in the chapter.) Virtual tables are said to *retain lineage* with their source tables.

The VALUES() Function

`VALUES()` is the first function you have come to in this book that returns a (virtual) table. If you type the word `VALUES` into the formula bar and read the tooltips, you can see that this function returns a table, as shown below.



One important thing to note about `VALUES()` is that it respects the initial filter context coming from your visuals. So if you combine this fact that VALUES() respects the current filter context with the information provided by IntelliSense in the image above, you will see that `VALUES()` returns a single-column table that contains the list of all possible values in the current filter context.

It's time to work through some examples to demonstrate the point.

A Calendar Example

Set up a new matrix as shown below and put `Calendar Year` on Rows. Then write the following measure in the `Calendar` table:

```
Total Months in Calendar
    = COUNTROWS(
    VALUES('Calendar'[MonthName])
    )
```

CalendarYear
2001
2002
2003
2004

Note: The following formula does not work here:

```
Total Months in Calendar wrong
= COUNTROWS('Calendar'[MonthName])
```

This formula doesn't work because COUNTROWS() expects a table as the input, but 'Calendar'[MonthName] is not a table; rather, it is a column that is part of a bigger table (the Calendar table, in this case).

When you wrap `'Calendar'[MonthName]` inside the `VALUES()` function, this single column that is part of the `Calendar` table is converted into a table in its own right, and it retains a relationship (lineage) to the original `Calendar` table. This new table returned by `VALUES()` is still a single column, but now it technically is a table *and* a column instead of simply being a column of some other table (`Calendar`, in this case).

So `VALUES('Calendar'[MonthName])` returns a single-column table of possible values that respects the initial filter context coming from the matrix. It is not possible to put this new table created by `VALUES()` into a measure unless you wrap it inside some other formula (e.g., an aggregator). In the example above, you first create the table (the `VALUES` part of the formula) and then count how many rows are in the table by using the `COUNTROWS()` function:

```
Total Months in Calendar
    = COUNTROWS(
    VALUES('Calendar'[MonthName])
    )
```

CalendarYear	Total Months in Calendar
2001	6
2002	12
2003	12
2004	12
Total	**12**

Notice how the year 2001 has only 6 months, and the other years all have 12. This is proof that the VALUES() function respects the initial filter context from the matrix. The initial filter context for the first row in the matrix is `'Calendar'[Year] = 2001`. That filter is applied before the formula `[Total Months in Calendar]` is calculated. `VALUES()` takes this "prefiltered" table (only dates where the year = 2001 are left unfiltered) and returns a single-column table that contains a distinct list of the possible values.

Returning a Single Value

`VALUES()` returns a single-column table of unique values from another column in another table, and this new table of values respects the current filter context coming from the visual in the report. There is another very cool feature of `VALUES()` that is very powerful: In the special case where `VALUES()` returns just a single row (i.e., one value), you can refer to this value directly in your formulas.

If you take the example created above, remove `CalendarYear` from Rows, and put `MonthName` on Rows instead, you should get the following.

MonthName	Total Months in Calendar
April	1
August	1
December	1
February	1
January	1
July	1
June	1
March	1
May	1
November	1
October	1
September	1
Total	**12**

Now you can see below that with the exception of the grand total, each row in the matrix has only a single value for `[Total Months in Calendar]`. So as long as you write the formula in such a way that it operates

over only a single row of the table, you can create a measure that returns the name of the month into the matrix's Values section (i.e., not Rows or Columns).

MonthName	Total Months in Calendar	Month Name (Values)
April	1	April
August	1	August
December	1	December
February	1	February
January	1	January
July	1	July
June	1	June
March	1	March
May	1	May
November	1	November
October	1	October
September	1	September
Total	**12**	

To write this formula, you need to provide protection for the other possible scenario, where `VALUES('Calendar'[MonthName])` has more than one row in the table. This is done using the function `HASONEVALUE()`, like so:

```
Month Name (Values)
    = IF(HASONEVALUE ( 'Calendar'[MonthName] ),
      VALUES( 'Calendar'[MonthName] )
    )
```

Remember that the structure of an `IF()` statement is as follows:

```
= IF(Logical Test, Result if True, [Result if False])
```

The last parameter is optional. If you leave it out, then you are accepting the default value, `BLANK()`.

If you write the above formula without the `HASONEVALUE()` function, it will throw an error. Even if you remove the grand total from the matrix, it will still throw an error. DAX allows you to use the single value returned in a single row of the single-column table only if you protect the formula with `HASONEVALUE()`.

The New SELECTEDVALUE() Function

In August 2017, Microsoft released a new function in Power BI, called `SELECTEDVALUE()`. You may recall that earlier I discussed the concept of *syntax sugar*—an approach Microsoft developers use to make difficult formulas easier to use. Here is the syntax for `SELECTEDVALUE()`:

```
SELECTEDVALUE(ColumnName, AlternateResult)
```

`SELECTEDVALUE()` was created to replace the complex formula in the previous section. Here is the formula from above:

```
Month Name (Values)
    = IF(HASONEVALUE ( 'Calendar'[MonthName] ),
    VALUES ( 'Calendar'[MonthName] )
    )
```

The new `SELECTEDVALUE()` function allows you to rewrite the same formula as follows:

```
Month Name Alternate
    = SELECTEDVALUE ( 'Calendar'[MonthName] )
```

Under the hood, `SELECTEDVALUE()` performs the `IF HASONEVALUE` test, and it returns the single value in the column if there is just one. `AlternateResult` is `BLANK()` by default.

Note: This new function is not available in Power Pivot for Excel at the time of this writing, so you need to use the previous pattern when using Excel.

CONCATENATEX() to the Rescue

Power BI has a special DAX function called `CONCATENATEX()` that iterates over a list of values in a table and concatenates them together into a single value. By using this function, you can write a formula that returns the single value when there is just one value but concatenate all the values into a single value when there are multiple values. You could write the earlier `VALUES` formula like this:

```
Month Name (Values)
    = CONCATENATEX(VALUES ( 'Calendar'[MonthName]),
          [MonthName],", ")
```

With this formula, you get the result shown below.

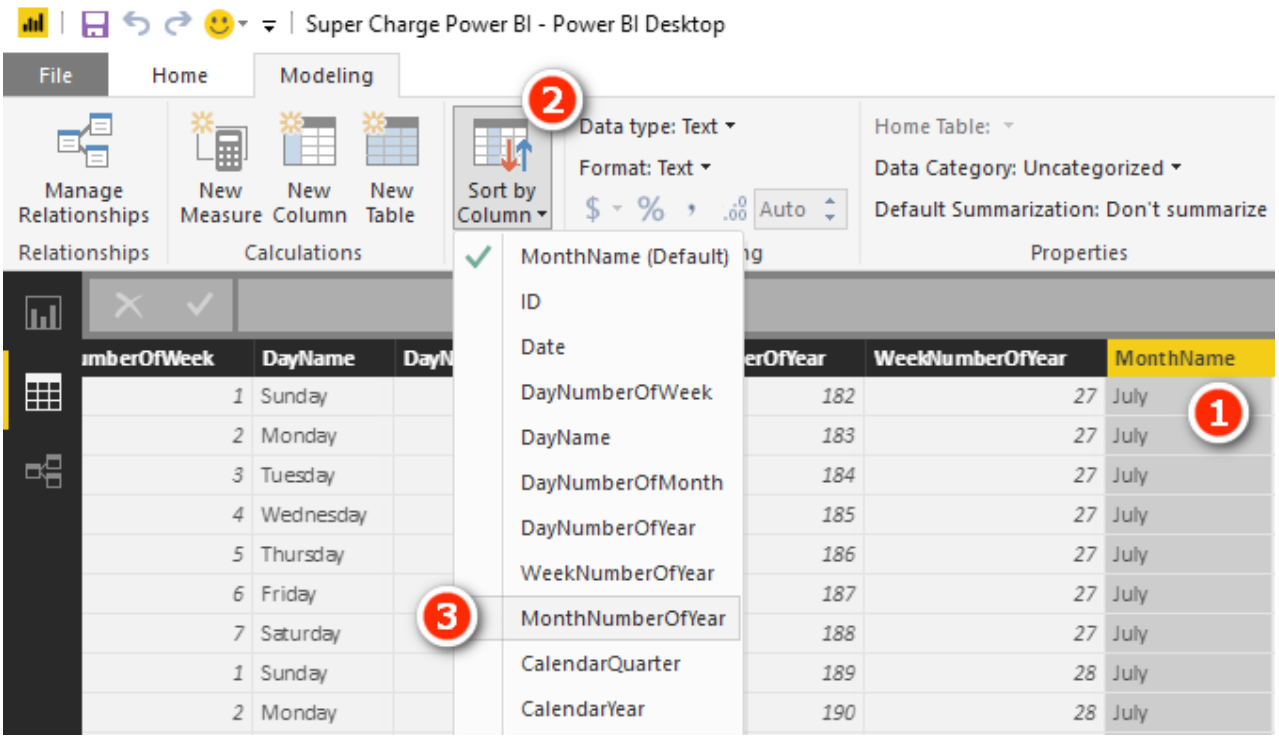
MonthName	Total Months in Calendar	Month Name (Values)
April	1	April
August	1	August
December	1	December
February	1	February
January	1	January
July	1	July
June	1	June
March	1	March
May	1	May
November	1	November
October	1	October
September	1	September
Total	**12**	**July, August, March, April, May, November, December, September, October, January, February, June**

Here's How: Changing the MonthName Sort Order

In the example above, you can see that the month names in the matrix are sorting in alphabetical order rather than in the logical month order of a calendar year. By default, all columns in all tables sort in alphanumeric order. It is, however, possible to change the sort order.

Follow these steps to change the sort order in a table:

1. Go to Data view and navigate to the Calendar table.
2. Click in the MonthName column (see #1 below), click the Sort by Column button (#2), and then select the MonthNumberOfYear column to be the sort column (#3).



3. When you return to your matrix, the rows are sorted in logical month order, as shown below.

MonthName	Total Months in Calendar	Month Name (Values)
January	1	January
February	1	February
March	1	March
April	1	April
May	1	May
June	1	June
July	1	July
August	1	August
September	1	September
October	1	October
November	1	November
December	1	December
Total	**12**	**July, August, March, April, May, November, December, September, October, January, February, June**

Note: Even though the columns are now sorting in the correct month name order, the concatenated value in the total row does not sort correctly. There is an optional `OrderBy` parameter available inside `CONCATENATEX()`, but it is not well documented and is difficult to use in this scenario. I therefore do not provide a solution to this problem in this book.

It is best practice to always load a numeric column in your lookup table for every alphabetic column that needs to be sorted in a different order. You should therefore always include a numeric column in your `Calendar` table for days of week as well as months of year.

Note: When you create a numeric sort column in a table, there must be a one-to-one match between the values in the numeric sort order and the values in the column to be sorted.

Practice Exercises: VALUES()

Create a new matrix and put `Products[Category]` on Rows and the measure `[Total Number of Products]` on Values. Then write the following measures by first creating a `VALUES()` table and then wrapping this table inside a `COUNTROWS()` function, as in the example shown earlier in this chapter. Find the solutions to these practice exercises in "Appendix A: Answers to Practice Exercises" on page 178.

39. [Number of Color Variants]

40. [Number of Sub Categories]

41. [Number of Size Ranges]

Use the column `Products[SizeRange]` for this one.

You should end up with a matrix that looks like the one below.

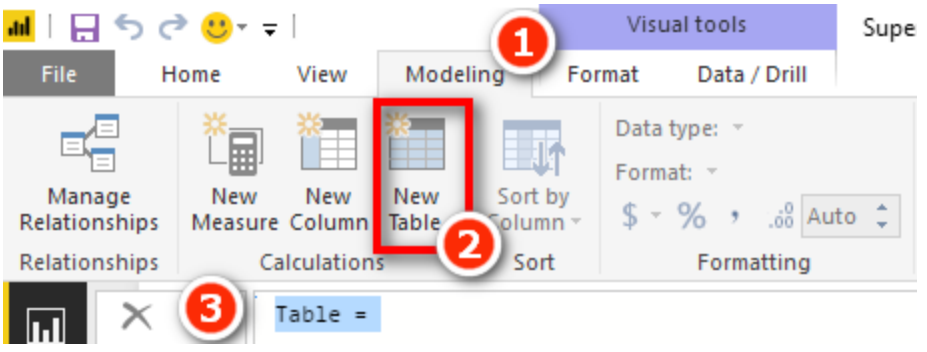
Category	Total Number of Products	Number of Color Variants	Number of Sub Categories	Number of Size Ranges
Accessories	35	6	12	2
Bikes	125	5	3	5
Clothing	48	5	8	5
Components	189	7	14	6
Total	**397**	**10**	**37**	**11**

Note: Each of these measures is the equivalent of dragging the column name and dropping it into the Values section of the matrix. When you drop a text field into the Values section of a matrix in Power BI, the matrix creates an implicit measure and uses COUNT() as the aggregating method. But recall that I recommended that you avoid doing this. The names created by implicit measures are ugly, and you need the DAX practice, so instead I recommend that you write explicit DAX measures, particularly while you are learning.

The New Table Button

Business users (i.e., those who don't have a professional IT background) often find it somewhat difficult to understand the `VALUES()` function because you can't actually "see" the table. So far I have shown you that even though you can't see the table, you can wrap the table inside a `COUNTROWS()` function—so at least you can see the number of rows in the table.

Power BI Desktop has a feature called New Table that is not currently available in Power Pivot for Excel. The New Table button is on the Modeling tab (see #1 below).

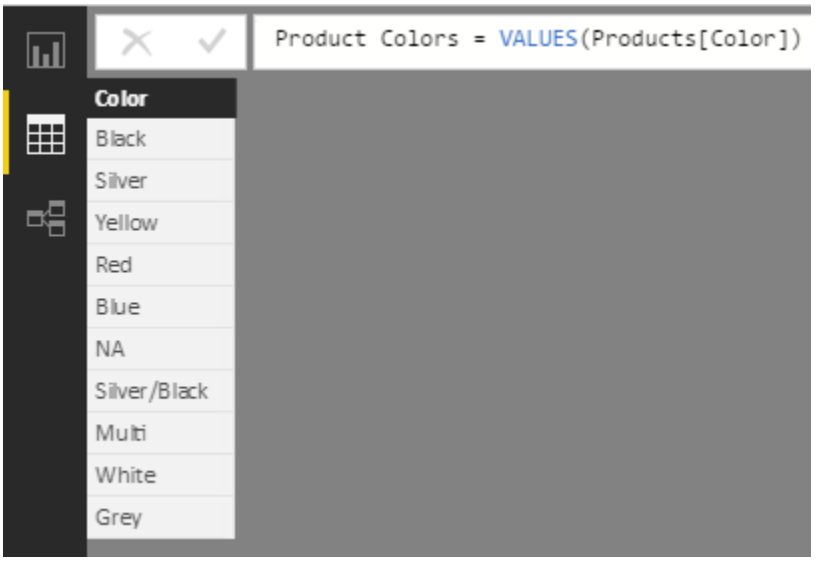


When you click this button (#2), you see that you can write a formula in the formula bar (#3). When you create a new table using this button, you can write a formula that returns a table (e.g., `VALUES()`), and the table is added to the data model.

Say that you write a table with the following DAX formula:

```
Product Colors = VALUES(Products[Color])
```

You get the new table shown below in the data model (visible in Data view).



Recall that `VALUES()` returns all the unique values in the current filter context. A new table doesn't respond to filters from Report view, so there is no initial filter context for a new table. The table above therefore shows a complete list of all possible values for `Product[color]` in the entire `Products` table (no filters applied).

The New Table button is very useful for "materialising" any virtual table so you can "see" the contents of the table. Later in the book, I will show you how you can add your own filter to one of these tables so you can see a subset of a table with a filter applied. For now, it is just good to know that if you ever can't get your head around a virtual table, you can always materialise it by using this feature so you can take a peek, and then you can delete it when you are done.

Practice Exercises: VALUES(), Cont.

Next, you should use the same matrix from the Practice Exercises 39–41 but remove the measure `[Number of Size Ranges]` from the matrix. Then write the following measures that each return a single value (the text name) into a cell in the matrix. Each formula has the word `(Values)` in the name, so it is clear that the formulas are returning the actual name value to the matrix; this is just a "note to self." In each example, make sure you wrap your `VALUES()` function in an `IF()` or `HASONEVALUE()` function, as in the example earlier in the chapter. Alternatively, you can use the `SELECTEDVALUE()` function if you prefer.

42. [Product Category (Values)]

43. [Product Subcategory (Values)]

44. [Product Color (Values)]

When you have finished, your matrix should look as shown below. Notice that two of these measures are blank. This is because the `VALUES` formula has more than one value, and hence the `IF HASONEVALUE` (or `SELECTEDVALUE`) part of the formula returns `BLANK()`; this is the default if you omit the last parameter.

Category	Total Number of Products	Number of Color Variants	Number of Sub Categories	Product Category (Values)	Product SubCategory (Values)	Product Color (Values)
Accessories	35	6	12	Accessories		
Bikes	125	5	3	Bikes		
Clothing	48	5	8	Clothing		
Components	189	7	14	Components		
Total	**397**	**10**	**37**			

45. Modifying Practice Exercise 43

Try editing the `IF()` statement for `[Product Sub Category (Values)]` so that it returns the value `More than 1 SubCategory` instead of `BLANK()`. The syntax for `IF` is `IF(`*Logical Test*`,` *Result if True*`,` *Result if False*`)`.

46. Modifying Practice Exercise 44

Now try editing the `IF()` statement for `[Product Color (Values)]` so that it returns `More than 1 Colour` instead of `BLANK()`. You should end up with something like the following (which shows answers for Practice Exercises 45 and 46).

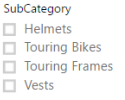
Category	Total Number of Products	Number of Color Variants	Number of Sub Categories	Product Category (Values)	Product SubCategory (Values)	Product Color (Values)
Accessories	35	6	12	Accessories	More than one SubCategory	More than one color
Bikes	125	5	3	Bikes	More than one SubCategory	More than one color
Clothing	48	5	8	Clothing	More than one SubCategory	More than one color
Components	189	7	14	Components	More than one SubCategory	More than one color
Total	**397**	**10**	**37**		**More than one SubCategory**	**More than one color**

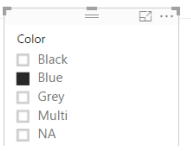
Finally, add a couple of slicers to your report from the `Products` table for color and subcategory.

> **Note:** If you add a slicer on a numeric column, it will look different to mine below. You can change the way a slicer displays from the drop-down arrow menu in the top-right corner of the slicer visual.

When you click on these slicers, the values in the matrix update to reflect the filtering in the slicer(s).

Category	Total Number of Products	Number of Color Variants	Number of Sub Categories	Product Category (Values)	Product SubCategory (Values)	Product Color (Values)
Accessories	3	1	1	Accessories	Helmets	Blue
Bikes	13	1	1	Bikes	Touring Bikes	Blue
Clothing	3	1	1	Clothing	Vests	Blue
Components	9	1	1	Components	Touring Frames	Blue
Total	**28**	**1**	**4**		**More than one SubCategory**	**Blue**





13: DAX Topic: ALL(), ALLEXCEPT(), and ALLSELECTED()

The DAX functions `ALL()`, `ALLEXCEPT()`, and `ALLSELECTED()` are all very similar in what they do. Let's start with `ALL()` and then look at the other two variants.

The ALL() Function

The `ALL()` function removes all current filters from the current filter context. For this reason, `ALL()` can be considered the "remove filters" function. The easiest way to understand this is with an example.

Create a new matrix and put `Products[Category]` on Rows and put the `[Total Number of Products]` measure you created earlier on Values. You get the matrix shown below.

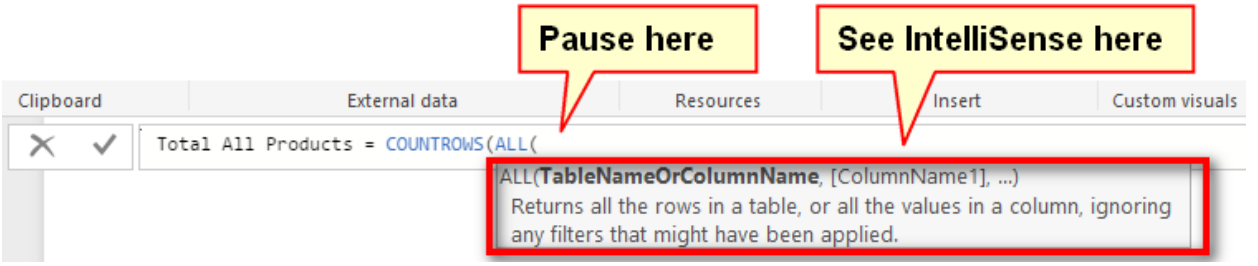
Category	Total Number of Products
Accessories	35
Bikes	125
Clothing	48
Components	189
Total	**397**

Technically what is happening above is that the first row in the matrix is filtering the Products table so that only products that are of type Products[Category]="Accessories" are visible in the underlying table; the other products are all filtered out. They are not really visible, but you can imagine what the underlying table in your data model would look like with a filter applied to `Accessories` behind the scenes. After the matrix applies a filter to the underlying tables, the measure `[Total Number of Products]` counts the rows that survive the filter. It does this for every cell in the matrix, one at a time, including the total cell. There is no filter applied to the total cell, so the measure counts all rows in the table (a completely unfiltered copy of the table).

Now create a new measure that uses the `ALL()` function:

```
Total All Products = COUNTROWS(ALL(Products))
```

The `ALL()` function returns a table. You can't see the table, but you can wrap `COUNTROWS()` around it so you can see how many rows are in the table. As you type this formula, if you pause while typing, IntelliSense displays the syntax for `ALL()`, as shown below.



In this case, IntelliSense is saying you that you can pass either a table or a single column as the first parameter for `ALL()`. In this example, you are passing the entire table. When you have finished typing this formula, add the measure to your matrix, and you should get the results shown below in your matrix.

Category	Total Number of Products	Total All Products
Accessories	35	397
Bikes	125	397
Clothing	48	397
Components	189	397
Total	**397**	**397**

You can see from this matrix that the new measure (the one on the right) is ignoring the initial filter context coming from the rows in the matrix. What is happening here is that first of all, the initial filter context is set by the row `Products[Category]`, but the `ALL()` function always returns an unfiltered copy of the table, and hence it returns the entire `Products` table instead of the filtered `Products` table. So `COUNTROWS()` returns `397` for every row in the matrix—including the total, as before.

Using ALL() as an Advanced Filter Input

The most common use of `ALL()` is as an advanced filter table input to a `CALCULATE()` function. Let's look at an example using `ALL()` as a table input to `CALCULATE()`.

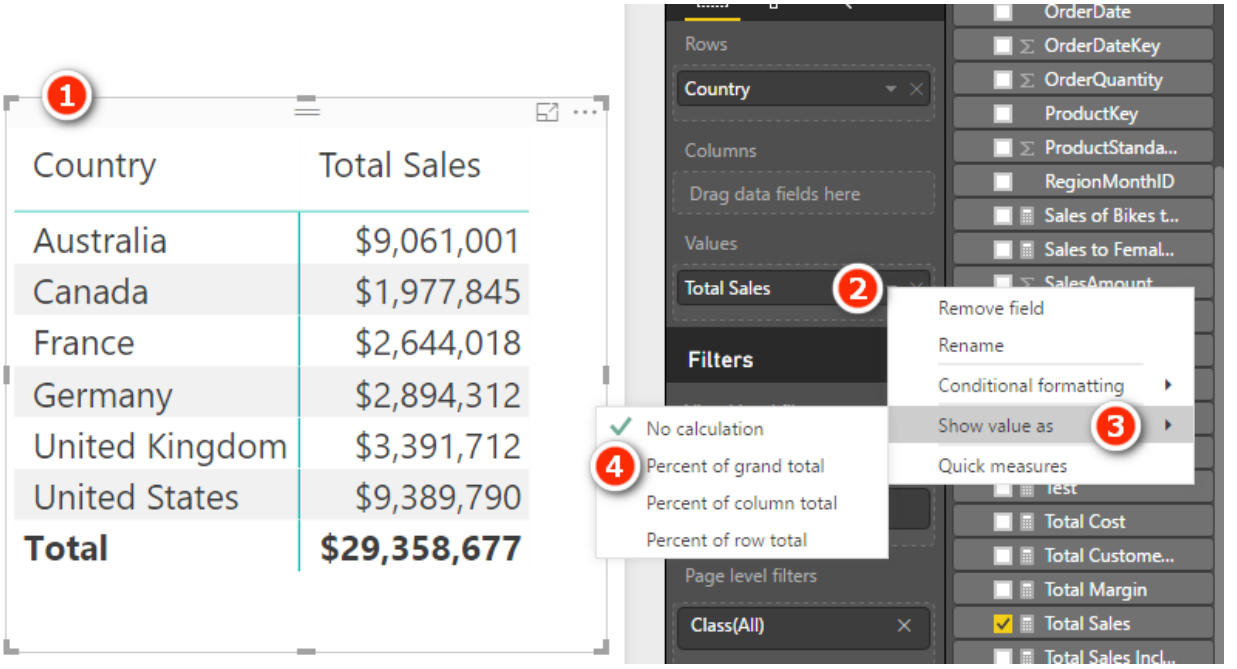
A good use for `ALL()` inside `CALCULATE()` is to remove the filters that are naturally applied to a visual so that you can access the number that is normally in the total line of the visual. Once you can access the equivalent total in a visual (such as a matrix) from any row in the matrix, you can then easily create a measure that finds the percentage of the total, which would be very useful indeed. The concept will make more sense as you work through the following example.

Calculating the Country Percentage of Total Sales

Say that you want to calculate the country percentage of global sales. First of all, set up a matrix with `Territories[Country]` on Rows and `[Total Sales]` on Values, as shown below.

Country	Total Sales
Australia	$9,061,001
Canada	$1,977,845
France	$2,644,018
Germany	$2,894,312
United Kingdom	$3,391,712
United States	$9,389,790
Total	**$29,358,677**

It is then possible (as shown below) to select the matrix (see #1 below), click on the drop-down arrow next to the measure `[Total Sales]` (#2), and then select Show Value As (#3) and Percent of Grand Total (#4).



But if you do this, you are only changing the display format of the result and not actually calculating the percentage as part of your data model. This means you can't use these percentages inside other measures, and you also can't reference the percentages from cube formulas (discussed in Chapter 19). Of course, you are also not learning to write DAX!

Writing Your Own DAX Measures

As illustrated in the preceding section, it is much better practice to create a new measure that will return the actual value as a reusable asset in your data model. You can do this in two steps.

> **Tip:** Remember that it is good practice to break the problem you are solving into pieces and solve one piece of the puzzle at a time.

Step 1: Create a Grand Total Measure

Right-click the `Sales` table and create the following new measure:

```
Total Global Sales
    = CALCULATE([Total Sales] , All(Territories) )
```

Don't forget to apply suitable formatting immediately, before moving on.

As you know, the first parameter of `CALCULATE()` is an expression, and the subsequent parameters are filters that modify the filter context. In this case, you are passing a table as the filter context. This table is `ALL(Territories)`, which is actually an *unfiltered copy* of the entire `Territories` table.

After you add the new measure to the matrix, your matrix looks as shown below. Do you see that the new measure is ignoring the initial filter context coming from the matrix? `CALCULATE()` is the only function that can modify the filter context. In this case, `CALCULATE()` is replacing the initial filter context on `Territories[Country]` with a new filter context (an unfiltered copy of the `Territories` table).

Country	Total Sales	Total Global Sales
Australia	$9,061,001	$29,358,677
Canada	$1,977,845	$29,358,677
France	$2,644,018	$29,358,677
Germany	$2,894,312	$29,358,677
NA		$29,358,677
United Kingdom	$3,391,712	$29,358,677
United States	$9,389,790	$29,358,677
Total	**$29,358,677**	**$29,358,677**

Step 2: Create the Percentage of Total

After you have created the measure `[Total Global Sales]`, it is easy to create a new measure to calculate the country percentage of global sales, as follows:

```
% of Global Sales
    = DIVIDE([Total Sales] , [Total Global Sales])
```

Make sure you format this measure so that Format is set to Percentage and Decimal Places is set to 1. You end up with the matrix shown below.

Country	Total Sales	Total Global Sales	% of Global Sales
Australia	$9,061,001	$29,358,677	30.9%
Canada	$1,977,845	$29,358,677	6.7%
France	$2,644,018	$29,358,677	9.0%
Germany	$2,894,312	$29,358,677	9.9%
NA		$29,358,677	
United Kingdom	$3,391,712	$29,358,677	11.6%
United States	$9,389,790	$29,358,677	32.0%
Total	**$29,358,677**	**$29,358,677**	**100.0%**

The final step is to remove the `[Total Global Sales]` measure from the matrix.

Note: You don't actually need the interim measures you write to be placed in the matrix in order for the [% of Global Sales] measure to work. But you should notice how much easier it is to visualise

what is happening when you write these measures in the context of a matrix. When you do it this way, you can easily see how the [Total Global Sales] value is the same, regardless of the country in the matrix, and hence you can immediately see that you just need to divide the country sales by this total global sales amount, and it is going to work.

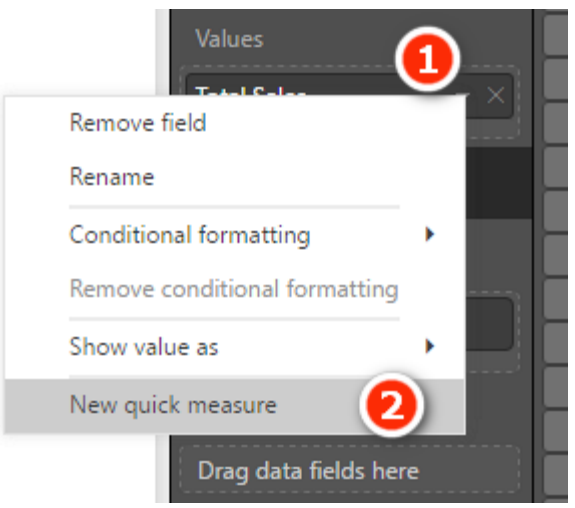
The final matrix is shown below, with some conditional formatting applied to make it easier to read.

Country	Total Sales	% of Global Sales
Australia	$9,061,001	30.9%
Canada	$1,977,845	6.7%
France	$2,644,018	9.0%
Germany	$2,894,312	9.9%
United Kingdom	$3,391,712	11.6%
United States	$9,389,790	32.0%
Total	**$29,358,677**	**100.0%**

Using the Quick Measures Option

You may already know—or may have noticed in one of the earlier images—that there is a New Quick Measure option available in Power BI. If you click the drop-down button next to Total Sales (see #1 below), you see the New Quick Measure option (#2).

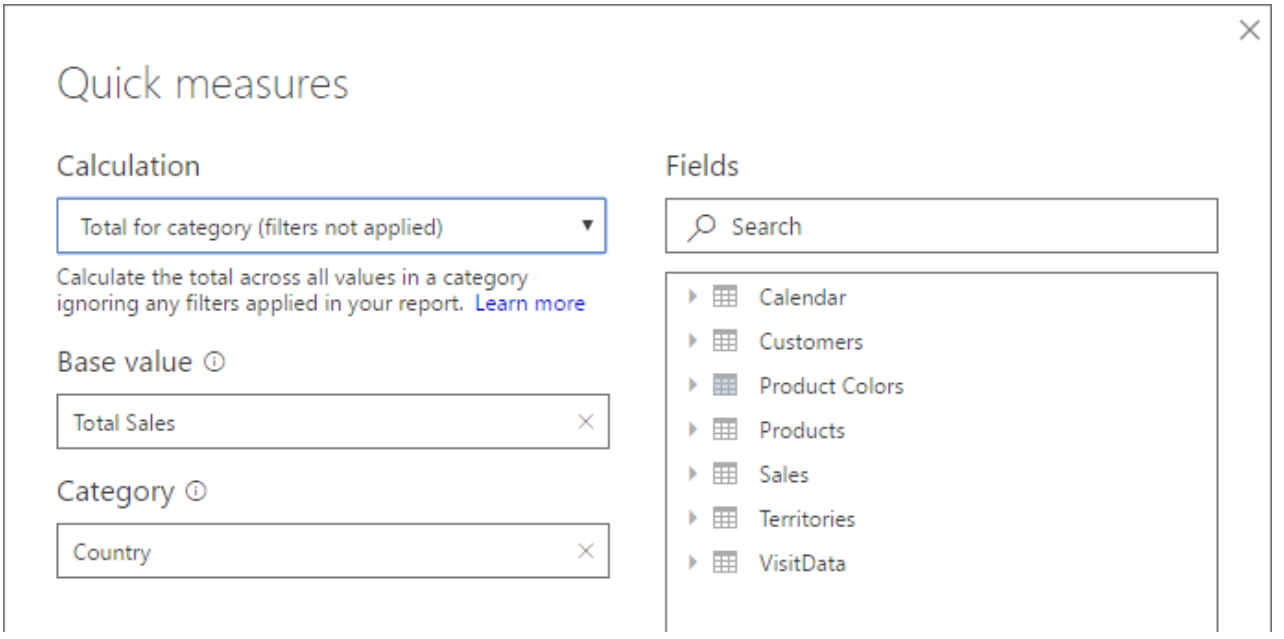
Note: At this writing, Quick Measures is still in preview. If you can't see the New Quick Measure option in the menu (as shown below), you can turn this preview feature on via the following menu: File\Options and Settings\Options\Preview Features\Quick Measures



If you select the New Quick Measure option, Power BI will help you write a measure, and you don't have to have any knowledge DAX at all. Using New Quick Measure is much better than creating implicit measures, as discussed earlier in this chapter. Some reasons are that you can rename the measure, edit the measure, and reuse the measure inside other measures. The image below shows Total for Category (Filters Not Applied) selected for the calculation in a new quick measure. All you have to do is drag a column or measure from the

fields list on the right into the relevant placeholders on the left. In the example below, I have added `[Total Sales]` to Base Value and `Territories[Country]` to Category.

> **Note:** Did you notice that I didn't need to tell you that `[Total Sales]` is a measure and `Territories[Country]` is a column? This is best practice at work. You should always omit the table name from the measure and always include the table name in front of a column; if you do, others in the DAX authoring community will understand what you mean.



The measure that Power BI creates is shown below.

```
Total Sales total for Country =
CALCULATE('Sales'[Total Sales], ALL('Territories'[Country]))
```

Note that this is a real measure—almost identical to the one that was hand-written above. However, at this writing, the New Quick Measure feature is not using best practice. Can you spot what is wrong with the measure above? The first parameter in `CALCULATE()` is a measure, but the measure is referencing the table name before the measure name. You should never do this—never ever! Always add the table name before a column name; do not add the table name before a measure name. The good news, however, is that because this is a real measure, you can simply edit what the New Quick Measure feature created and correct the syntax yourself. (I expect that this "poor practice" will be fixed at some point in the future. I have already reported it to Microsoft.)

> **Note:** Using the New Quick Measure option is a great help for writing more complex DAX functions quickly and without a lot of knowledge about how it all works. However, I do not spend any more time talking about the feature in this book because this is a book about writing your own DAX formulas. If you want to use New Quick Measure, then do so by all means. I don't recommend that you use New Quick Measure in place of learning how the DAX language works; rather, I recommend that you use New Quick Measure as a tool to help you learn the DAX language.

Passing a Table or a Column to ALL()

Before we finish with `ALL()`, it is worth pointing out that this next measure would return exactly the same result as `[Total Global Sales]` in the matrix example in the section "Step 1: Create a Grand Total Measure," above:

```
Total All Country Sales
    = CALCULATE([Total Sales],
      ALL(Territories[Country])
    )
```

Notice that this measure passes a single *column* instead of the entire table to the `ALL()` function. So in this specific matrix (shown below), the values for `[Total Global Sales]` and `[Total All Country Sales]` are identical.

Country	Total Sales	Total Global Sales	Total All Country Sales
Australia	$9,061,001	$29,358,677	$29,358,677
Canada	$1,977,845	$29,358,677	$29,358,677
France	$2,644,018	$29,358,677	$29,358,677
Germany	$2,894,312	$29,358,677	$29,358,677
NA		$29,358,677	$29,358,677
United Kingdom	$3,391,712	$29,358,677	$29,358,677
United States	$9,389,790	$29,358,677	$29,358,677
Total	**$29,358,677**	**$29,358,677**	**$29,358,677**

However, the measure `[Total All Country Sales]` would not work (i.e., it would not remove the filter) if there were some other column on Rows in the matrix (i.e., something other than `Country`). To test this, remove `Territories[Country]` from Rows in the matrix and replace it with `Territories[Region]`. You get the result shown below.

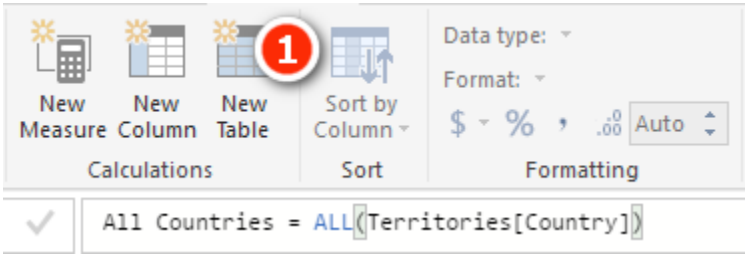
Region	Total Sales	Total Global Sales	Total All Country Sales
Australia	$9,061,001	$29,358,677	$9,061,001
Canada	$1,977,845	$29,358,677	$1,977,845
Central	$3,001	$29,358,677	$3,001
France	$2,644,018	$29,358,677	$2,644,018
Germany	$2,894,312	$29,358,677	$2,894,312
NA		$29,358,677	
Northeast	$6,532	$29,358,677	$6,532
Northwest	$3,649,867	$29,358,677	$3,649,867
Southeast	$12,239	$29,358,677	$12,239
Southwest	$5,718,151	$29,358,677	$5,718,151
United Kingdom	$3,391,712	$29,358,677	$3,391,712
Total	**$29,358,677**	**$29,358,677**	**$29,358,677**

Notice the difference between passing the entire table name to the `ALL()` function and passing a single column. `[Total Global Sales]` removes the filter from the entire `Territories` table, but `[Total All Country Sales]` removes filters only from the `Territories[Country]` column of the table. In the image above, there is no filter on the `Territories[Country]` column of the table, and hence `ALL()` has no effect on the visual.

Remove `[Total All Country Sales]` from the matrix before proceeding.

The New Table Option Revisited

Now is a good time to revisit the New Table feature introduced earlier in the book. Remember that you can use the New Table button (see #1 below) to "materialise" a table into the data model. Normally you can't "see" a table function in a DAX formula, and that makes it hard to understand what the table function is doing. But if you take the table function and add a new table, as shown below, you can actually see the table that is generated.



After creating the table above, switch to Data view and click on the new table. You can see that this table consists of a list of six countries plus NA.

This table is just a test table. It isn't connected to the data model at all, but it can be if you want to connect it. I just like to materialise tables like this as part of the learning experience because it is helpful to be able to visualise table functions. You should delete such test tables when you are done so you don't get confused.

The ALLEXCEPT() Function

`ALLEXCEPT()` allows you to remove filters from all columns in a table except the ones that you explicitly specify. Consider the following example:

```
Total Sales to Region or Country
    = CALCULATE([Total Sales],
      All(Territories[Region],Territories[Country])
    )
```

`ALLEXCEPT()` solves the problem implied above, where you need to specify many columns individually in the case that you want most but not all columns in your formula. The above formula works when you have `Territories[Country]` on Rows and also when you have `Territories[Region]` on Rows, but it does not work with `Territories[Group]` on Rows. If you have a lot of columns in your table, you have to write a lot of DAX code to make such a formula work for all but a few of the columns. This is where `ALLEXCEPT()` comes into play. The above formula can be rewritten as follows:

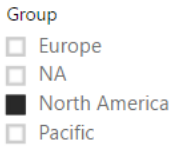
```
Total Sales to Region or Country 2
    = CALCULATE([Total Sales],
      ALLEXCEPT(Territories, Territories[Group]))
```

Note: You must first specify the table that is to be included and then specify the exception columns.

The ALLSELECTED() Function

The `ALLSELECTED()` function is useful when you want to calculate percentages, as shown above, and you have a filter applied (say, via a slicer) but you want the total in your matrix to add up to 100%.

Say that you're working with the same matrix used earlier in this chapter but now with a slicer that filters on `Territories[Group]`. Notice below that `[% of Global Sales]` adds up to 38.7%; this is correct because the other countries that make up the remaining 61.3% have been filtered out by the Group slicer.

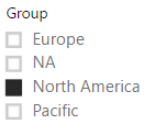


Region	Total Sales	% of Global Sales
Canada	$1,977,845	6.7%
Central	$3,001	0.0%
Northeast	$6,532	0.0%
Northwest	$3,649,867	12.4%
Southeast	$12,239	0.0%
Southwest	$5,718,151	19.5%
Total	**$11,367,634**	**38.7%**

But say that you want to see the percentage of each region out of all the values in the matrix (in this example, just the regions in the group North America). This is where `ALLSELECTED()` comes in. `ALLSELECTED()` removes the filters from the matrix but respect the filters in the slicer.

Add the following measure to the matrix above:

```
Total Selected Territories
    = CALCULATE([Total Sales], ALLSELECTED(Territories))
```



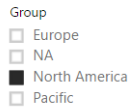
Region	Total Sales	% of Global Sales	Total Selected Territories
Canada	$1,977,845	6.7%	$11,367,634
Central	$3,001	0.0%	$11,367,634
Northeast	$6,532	0.0%	$11,367,634
Northwest	$3,649,867	12.4%	$11,367,634
Southeast	$12,239	0.0%	$11,367,634
Southwest	$5,718,151	19.5%	$11,367,634
Total	**$11,367,634**	**38.7%**	**$11,367,634**

Notice how the interim measure `[Total Selected Territories]` returns the same value as the total of the items in the matrix. Using the same steps as before, you can now write a new measure `[% of Selected Territories]` and then remove the interim measure [Total `Selected Territories]` from the matrix.

Now write the following measure:

```
% of Selected Territories
    = DIVIDE([Total Sales] , [Total Selected Territories])
```

Remember to format this new measure using percentage and one decimal place.



Region	Total Sales	% of Global Sales	% of Selected Territories
Canada	$1,977,845	6.7%	17.4%
Central	$3,001	0.0%	0.0%
Northeast	$6,532	0.0%	0.1%
Northwest	$3,649,867	12.4%	32.1%
Southeast	$12,239	0.0%	0.1%
Southwest	$5,718,151	19.5%	50.3%
Total	**$11,367,634**	**38.7%**	**100.0%**

Using Interim Measures

Remember that it is good practice to split a problem into pieces and solve one piece of the problem at a time. My advice is to get used to creating interim measures first and then writing the final measure that you actually need. Doing this helps you visualise each step of the process and makes it easier to get each part of the end-state formula correct before you proceed to the next step.

It is, of course, possible to write one single measure that does all the steps you just went through. This is what it would look like:

```
% of Selected Territories ONE STEP
    = DIVIDE([Total Sales] ,
      CALCULATE([Total Sales],
      ALLSELECTED(Territories)
      )
    )
```

But this all-in-one formula is much harder to write, read, and debug—particularly when you are learning to write DAX. It's not wrong; it's just harder, and life is too short to do things that are harder than they need to be.

Practice Exercises: ALL(), ALLEXCEPT(), and ALLSELECTED()

It's time for some practice. Create a new matrix and put `Customers[Occupation]` on Rows and the measure `[Total Sales]` on Values. You get the matrix shown below.

Occupation	Total Sales
Clerical	$4,684,787
Management	$5,467,862
Manual	$2,857,971
Professional	$9,907,977
Skilled Manual	$6,440,081
Total	**$29,358,677**

Then, using the principles covered in this chapter, create the following measures by first creating the interim measure you need and then creating the final measure. Find the solutions to these practice exercises in "Appendix A: Answers to Practice Exercises" on page 178.

47. [Total Sales to All Customers]

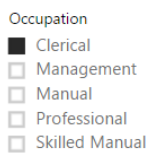
48. [% of All Customer Sales]

Now add a slicer for Customers[Gender] to the report you have just created and filter by Gender = M, as shown below.

Occupation	Total Sales	Total Sales to All Customers	% of All Customer Sales
Clerical	$2,421,327	$29,358,677	8.2%
Management	$2,793,527	$29,358,677	9.5%
Manual	$1,463,060	$29,358,677	5.0%
Professional	$4,773,493	$29,358,677	16.3%
Skilled Manual	$3,093,651	$29,358,677	10.5%
Total	**$14,545,059**	**$29,358,677**	**49.5%**

Note how `[% of All Customer Sales]` doesn't add to 100%. This is correct because the other 50.5% of customers are filtered out with the slicer.

Set up another matrix with Customers[NumberCarsOwned] on Rows, `Customers[Occupation]` on Slicer, and `[Total Sales]` on Values. Your job is to create the other measure in this matrix: `[% of Sales to Selected Customers]`. When you are done, your matrix should look like the one below, with the last column showing the percentage of sales to customers based on the number of cars they own.



NumberCarsOwned	Total Sales	% of Sales to Selected Customers
0	$2,660,886	56.8%
1	$1,204,496	25.7%
2	$790,154	16.9%
3	$28,141	0.6%
4	$1,109	0.0%
Total	**$4,684,787**	**100.0%**

Remember that in this case, you want to create an interim measure first, so you actually need to create the following two measures and then remove the first one from the matrix.

49. [Total Sales to Selected Customers]

50. [% of Sales to Selected Customers]

Create the following two measures. The first one is an interim formula and can be removed from the matrix once you have finished the second formula. Find the solutions to these practice exercises in "Appendix A: Answers to Practice Exercises" on page 178

51. [Total Sales for All Days Selected Dates]

52. [% Sales for All Days Selected Dates]

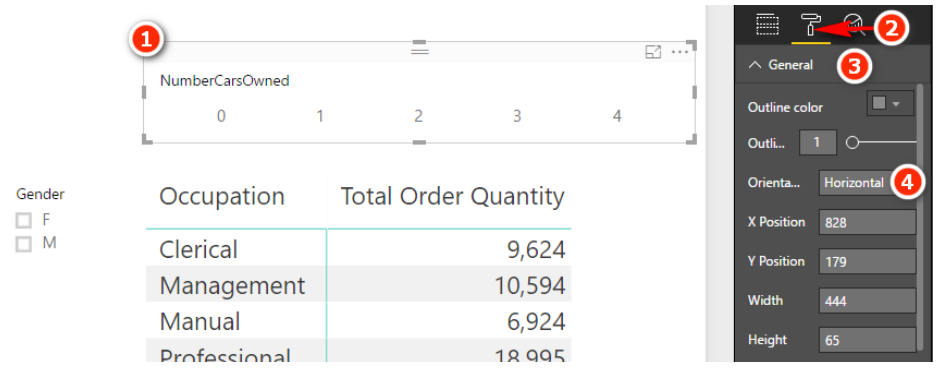
Here's How: Using ALLEXCEPT()

I don't use `ALLEXCEPT()` much, and you may not either, but it is still good to work through an example of how it can be used. This section will give you some practice while also demonstrating one possible use case.

Say that you want to compare the percentage of sales across all occupations and see how it changes depending on the other customer filters. Follow these steps:

1. Set up a new matrix and place Customers[Occupation] on Rows.
2. Add slicers for Gender and NumberCarsOwned.
3. Put `[Total Order Quantity]` on Values. You should have the setup shown below. Note that the total order quantity will change as you click on the slicers.

Here I have set the `NumberCarsOwned` slicer to be horizontal by selecting the slicer (see #1 below), going to the Format pane (#2), selecting General (#3), and then setting Orientation to Horizontal (#4).



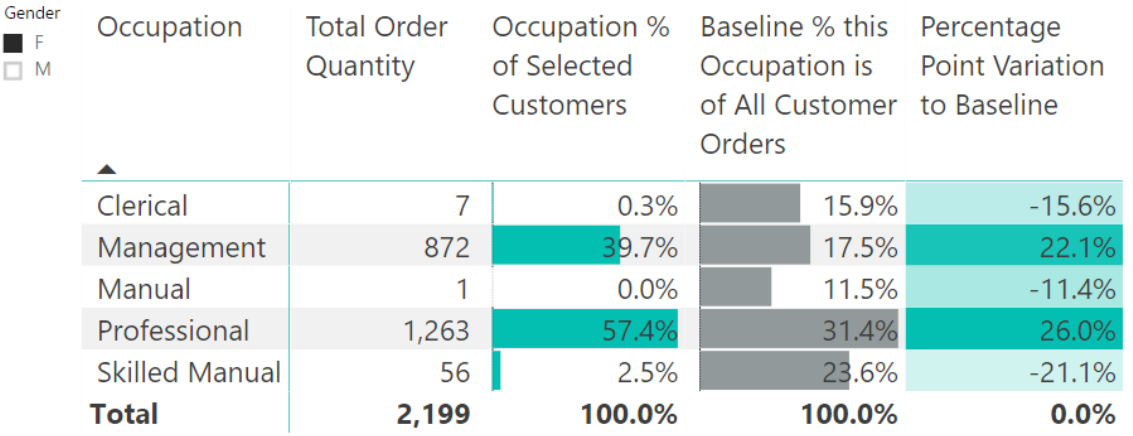
A number of steps are required to get to the end state (which is shown below). The following practice exercises show the measures you need to create, in the proper order, to get to the end state. As you create each

measure, check that the results you see in your matrix make sense. Once again, this is the reason to write DAX in the context of a matrix: It makes it easier to get your head around what you are doing.

The following matrix shows all the measures you need to write so you have an overview of what you'll accomplish with the following measures. Note that there are some slicers applied to the report already.

Occupation	Total Order Quantity	Total Orders All Customers	Baseline Orders for All Customers with this Occupation	Baseline % this Occupation is of All Customer Orders	Total Orders Selected Customers	Occupation % of Selected Customers	Percentage Point Variation to Baseline
Clerical	7	60,398	9,624	15.9%	2,199	0.3%	-15.6%
Management	872	60,398	10,594	17.5%	2,199	39.7%	22.1%
Manual	1	60,398	6,924	11.5%	2,199	0.0%	-11.4%
Professional	1,263	60,398	18,995	31.4%	2,199	57.4%	26.0%
Skilled Manual	56	60,398	14,261	23.6%	2,199	2.5%	-21.1%
Total	**2,199**	**60,398**	**60,398**	**100.0%**	**2,199**	**100.0%**	**0.0%**

The image below shows the end state you are working toward, with just the final measures included.



As you can imagine, with this matrix, it is possible to select different combinations of gender and number of cars and then compare the variation between the baseline order quantity and the order quantities for the selected filter.

Practice Exercises: ALL(), ALLEXCEPT(), and ALLSELECTED(), Cont.

Write the following DAX formulas one at a time and check to make sure each looks correct before moving to the next one. Find the solutions to these practice exercises in "Appendix A: Answers to Practice Exercises" on page 178.

53. [Total Orders All Customers]

To check this measure, click on the slicers and note that `[Total Order Quantity]` should change, but `[Total Orders All Customers]` should not change based on the slicers.

54. [Baseline Orders for All Customers with This Occupation]

This measure should also not change when you make changes to the slicers. However, note that you should get a different value for each occupation—unlike with [Total Orders All Customers] above. This will be the baseline for comparison.

55. [Baseline % This Occupation of All Customer Orders]

This measure converts the baseline measure above into a percentage of the baseline for all orders. The description of this measure should help you work out how to write the DAX. Test the slicers again and make sure this new baseline percentage doesn't change with the slicers.

56. [Total Orders Selected Customers]

This measure should adjust depending on the selections you have in the slicers. *Hint:* Use `ALLSELECTED()`.

57. [Occupation % of Selected Customers]

You can use the interim measures above to create this measure. Click the slicers a few times and see which values change. This new measure should change based on the values you select in the slicers.

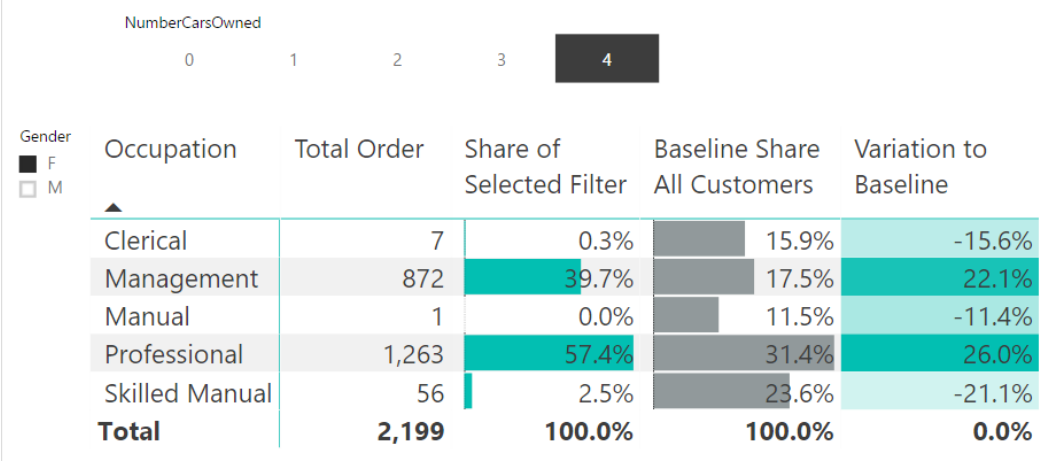
58. [Percentage Point Variation to Baseline]

This measure is the percentage of selected customers (Practice Exercise 57) minus the baseline (Practice Exercise 55).

Now you should have an interactive report that allows you to drill into customer attributes (gender and number of cars owned) to see the impact on the mix of business vs. the baseline of all customers.

It is worth pointing out here that sometimes it may be useful to change the descriptions of the final measures as they appear in a matrix. So while `[Baseline % This Occupation of All Customer Orders]` is a good name for your measure because you know what it means, when you use this measure in a specific matrix, it may be a good idea to rename it. You can do this by selecting the matrix, going to the Values section on the right-hand side of the screen, and double-clicking the measure name you want to change. The name changes just for this single visual (the matrix in this case).

After giving your measures new names, you might end up with something like the matrix shown below.



Occupation	Total Order	Share of Selected Filter	Baseline Share All Customers	Variation to Baseline
Clerical	7	0.3%	15.9%	-15.6%
Management	872	39.7%	17.5%	22.1%
Manual	1	0.0%	11.5%	-11.4%
Professional	1,263	57.4%	31.4%	26.0%
Skilled Manual	56	2.5%	23.6%	-21.1%
Total	**2,199**	**100.0%**	**100.0%**	**0.0%**

Note: If you change the description in the matrix, the easiest way to change it back is to remove the measure from the visual and then add it back again.

14: DAX Topic: FILTER()

`FILTER()` is a very powerful function in DAX. When `FILTER()` and `CALCULATE()` are combined, these two functions allow you to alter the filter context in your matrixes any way you want. However, before we move to using `FILTER()` with `CALCULATE()`, I think it is worth looking at `FILTER()` on its own with a couple simple examples.

> **Note:** These examples demonstrate how the FILTER() function works, but you probably would not actually write such formulas in real DAX. These formulas are created here just for demonstration purposes.

The syntax of `FILTER()` is as follows:

```
= FILTER(Table, myFilter)
```

Table is any table (or function that returns a table, such as `ALL()`), and *myFilter* is any expression that evaluates to a `TRUE`/`FALSE` answer.

The `FILTER()` function returns a table that contains zero or more rows from the original table. Said another way, the table returned by `FILTER()` can contain zero rows, one row, two rows, or any other number of rows, up to and including the total number of rows in the original table. The purpose of `FILTER()`, therefore, is to determine which rows will be returned to the final table result after you use the `myFilter` test.

`FILTER()` is an iterator, and it can therefore complete granular analysis to determine which rows will be included in the final table. Generally, it is fine to use `FILTER()` over lookup tables, but it's somewhat more risky to use it over data tables, particularly if they are very large (millions of rows). Whether you should use `FILTER()` depends on your data, on the quality of your DAX formulas inside `FILTER()`, and on what you need to achieve.

Let's work through an example. Set up a new matrix with `Customers[Occupation]` on Rows and the measure `[Total Number of Customers]` on Values. The matrix, shown below, indicates how many customers there are in the entire customer database for each occupation type.

Occupation	Total Number of Customers
Clerical	2,928
Management	3,075
Manual	2,384
Professional	5,520
Skilled Manual	4,577
Total	**18,484**

But what if you want to know how many customers in the database have an income of more than $80,000 per year? Consider the following formula:

```
=FILTER(Customers, Customers[YearlyIncome] >= 80000)
```

The result of this formula is a table of customers, and this new virtual table of customers includes all the customers that have an income greater than or equal to $80,000 per year. But note that it is a table, and you can't put a table of values into a matrix. So if you want to see the result (in this case, the total count of rows) inside a matrix, you have to wrap this table that was returned by `FILTER()` inside a function that returns a value instead of a table of values (such as an aggregator).

It is possible to count the number of rows in this table by wrapping the formula above inside another formula, like this:

```
Total Customers with Income of $80,000 or above
    = COUNTROWS(
    FILTER(
        Customers, Customers[YearlyIncome]>=80000
    )
)
```

If you write this formula and put it in the matrix, it looks as shown below. You should do this now for practice.

Occupation	Total Number of Customers	Total Customers with Income of $80,000 or above
Clerical	2,928	
Management	3,075	1,963
Manual	2,384	
Professional	5,520	1,976
Skilled Manual	4,577	443
Total	**18,484**	**4,382**

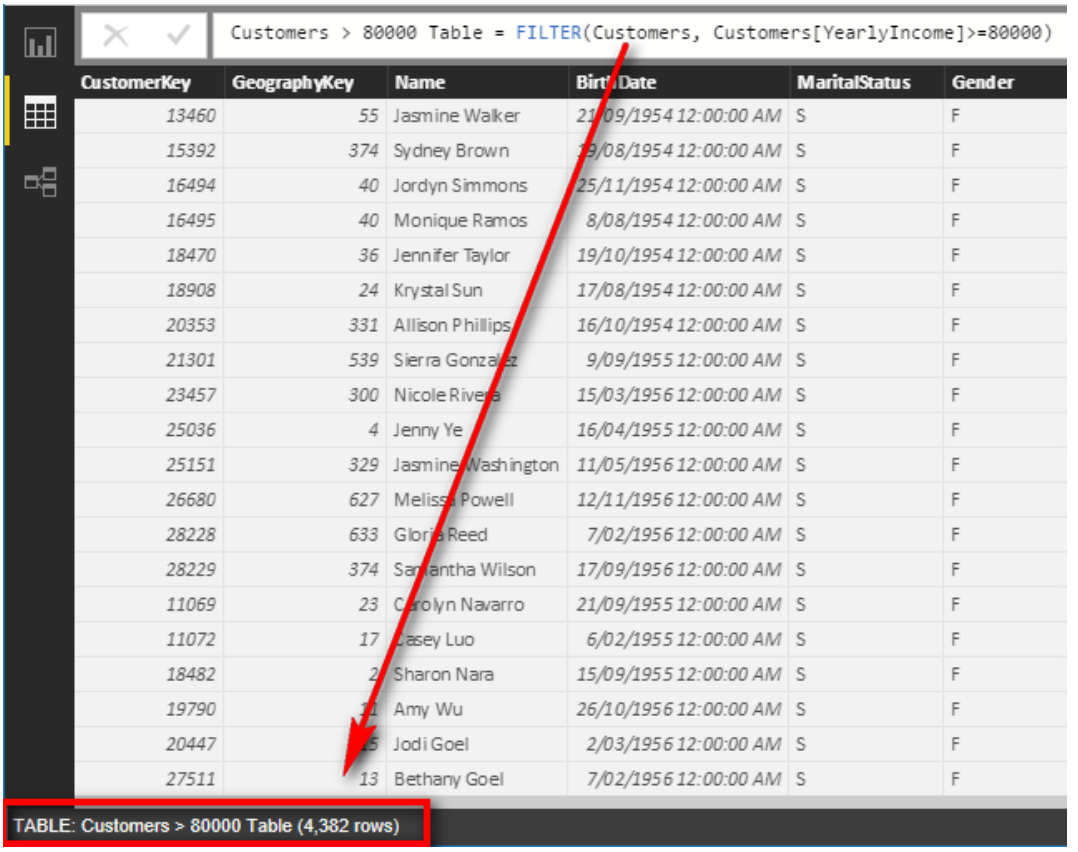
You can see from the matrix that not all occupations have customers that earn this amount of money.

New Table Strikes Again

Don't forget about that cool New Table button you've seen a couple times already. If you want to "see" the filtered copy of the `Customers` table from the preceding section—just as a test and to get your head around what is happening under the hood—create a new table with the following formula:

```
Customers > 80000 Table = FILTER(Customers, Customers[YearlyIn-
come]>=80000)
```

If you switch to this new table in Data view, you can see the table and also check how many rows are in it. As you can see below, it is indeed a filtered copy of the original `Customers` table.



These are the key points to take away from this example:

- FILTER() returns a table. It is a virtual table unless it is materialised using the New Table option.
- The virtual copy of the table that is used inside the measure above (not the materialised copy) retains a link to the original table and can have an effect on the other tables in the data model. (You'll learn more about this later in this chapter.)
- You can't put the table returned by Filter() into a matrix as is because you simply can't put a table into a matrix. But you can count how many rows there are in the table and put that answer into the matrix. This is exactly what happens with this measure.

So How Does FILTER() Actually Work?

It is essential that you understand how the `FILTER()` function works before moving on. Let's look just at the `FILTER()` portion of the formula above:

```
FILTER(Customers, Customers[YearlyIncome]>=80000)
```

`FILTER()` is an iterator just like `SUMX()`, covered in Chapter 7. As such, `FILTER()` first creates a row context on the specified table (the first parameter) and then iterates through each row in the table to check whether the row passes the test. If an individual row passes the test, it is retained in the final table result. If an individual row fails the test, it is omitted from the final table result.

> **Note:** As mentioned in Chapter 7, it is convenient to think of iterators working one row at a time, and indeed that is the logical execution approach. In reality, though, the Power BI engine has been built and optimised to work very efficiently under the hood. You should not think that iterators are inherently inefficient because the Power BI engine can make the physical execution very efficient indeed.

Let's look at a simple example. Assume that the `Customers` table has five rows, as shown below.

Row	CustomerKey	YearlyIncome
1	11003	$ 70,000
2	11004	$ 80,000
3	11005	$ 70,000
4	11007	$ 60,000
5	11008	$ 80,000

> The Row column has been added here to assist in the explanation of how `FILTER()` works; it does not actually exist in the Customers table.

Here again is the `FILTER()` portion of the formula from above:

```
FILTER(Customers, Customers[YearlyIncome]>=80000)
```

This is what the `FILTER()` function does (logically speaking):

1. It first creates a new row context over the Customers table. The row context allows FILTER() to keep track of which row it is looking at, and it also provides the capability to isolate a single row (one at a time) and refer to the single value that is the intersection between the single row and any column(s) in the table.
2. Now that there is a row context, FILTER()goes to row 1 and asks the question (from the `myFilter` portion of the formula) "Is the value in the column Customers[YearlyIncome] for the customer in this row greater than or equal to $80,000?" If the answer is yes, row 1 survives the filter test, and the row is kept in the final table result. If the answer is no, row 1 is discarded from the final table result.

So in this case (row 1), the yearly income is $70,000, so it fails the test, and row 1 is discarded from the final table result.

3. FILTER() then moves to the second row (using the row context to keep track of where it is) and asks the same question again for this new row: "Is the value in the column Customers[YearlyIncome] for the customer in this row greater than or equal to $80,000?" If the answer is yes, the row survives the filter test, and the row is kept in the final table result. If the answer is no, it fails the test, and the row is discarded from the final table result. In the case of row 2, it passes the test, and hence row 2 is retained.
4. FILTER()works down the table one row at a time and tests each row against the filter test. It decides which rows to keep and which rows to discard by checking each row, one at a time, against the filter test.
5. When the last row has been evaluated, FILTER() returns a table that contains just the rows that passed the test, as shown below. All rows that failed the test are discarded.

Row	CustomerKey	YearlyIncome
2	11004	$ 80,000
5	11008	$ 80,000

So it is clear from the image above that if you now count the rows of the table on the left (which is the result of `FILTER()`), you get the answer `2` (i.e., there are two rows in this filtered table).

If you now refer back to the earlier example, it is clear how the formula gave you the results. Here is the formula, shown again for convenience:

```
Total Customers with Income of $80,000 or above
    = COUNTROWS(
    FILTER(
        Customers, Customers[YearlyIncome]>=80000
    )
)
```

The measure `[Total Customers with Income of $80,000 or Above]` iterates through the `Customers` table using the `FILTER()` function, checking the value in the `Customers[YearlyIncome]` column of each individual customer to see if the value is greater than or equal to $80,000. `FILTER()` returns a table of all customers that passed this test, and then `COUNTROWS()` counts them. As shown below, the formula finds that 1,963 customers with the occupation *management* pass this test, and 4,382 customers in total pass the test.

Occupation	Total Number of Customers	Total Customers with Income of $80,000 or above
Clerical	2,928	
Management	3,075	1,963
Manual	2,384	
Professional	5,520	1,976
Skilled Manual	4,577	443
Total	**18,484**	**4,382**

As mentioned earlier, FILTER() is most commonly used as an advanced filter inside CALCULATE(). FILTER() is an iterator, and it therefore allows a very granular level of evaluation of a table and is a very powerful tool for altering the filter context of a matrix any way you want and at a level of detail that is not possible by using a simple filter inside CALCULATE().

Consider the following formula:

```
Total Customers with Income of $80,000 or above 2
    = CALCULATE(COUNTROWS(Customers),
    Customers[YearlyIncome]>=80000)
```

This formula returns exactly the same result as the FILTER() version above. In this version, the filter portion of the formula uses a simple filter, or raw filter. A simple filter has a column name on one side of the formula (in this case, Customers[YearlyIncome]) and a value on the right side (in this case, 80000).

CALCULATE() is designed to accept this type of simple syntax without using the FILTER() function. But in reality, as mentioned earlier, this is just "syntax sugar" created by the developers to make it easier for you to write measures. Under the hood, the formula above is converted to the following formula:

```
Total Customers with Income of $80,000 Under the Hood
    = CALCULATE(COUNTROWS(Customers),
    FILTER(ALL(Customers),Customers[YearlyIncome]>=80000)
    )
```

Note the inclusion of ALL(Customers) instead of just Customers as the first parameter of the FILTER() function. I cover why the ALL() function is needed here in more detail in Chapter 15.

There is a limit to what CALCULATE() can do with one of these simple filters. It works only if you have a column name compared to a value. But what if you want to do something more complex, like check whether another measure is greater than a value? Let's look at another example.

Example: Calculating Lifetime Customer Purchases

Say that you want to know how many customers have purchased more than $5,000 of goods from you over the course of time. You can't use a simple CALCULATE() filter in this case because customers may have purchased from you on many occasions, and you don't have a column that contains a single value that tells the total sales for each customer. If you tried to write this formula using a simple filter, it would look like this:

```
Customers with Sales Greater Than $5,000 Doesn't Work
    = CALCULATE(COUNTROWS(Customers),[Total Sales] > 5000)
```

This formula includes a measure on the left side—and that is not allowed with a simple filter! A simple filter *must* have a column compared to a value, so a simple filter doesn't work in this scenario.

This is where you have to write your own FILTER() functions. Now take a look at the following formula:

```
Customers with Sales Greater Than $5,000
    = CALCULATE(
        COUNTROWS(Customers),
        FILTER(Customers,
        [Total Sales] >= 5000
    )
)
```

It is worth stepping through how FILTER() works in this example because there is a slight difference from the earlier example.

> **Note:** The FILTER() portion of the formula is *evaluated first*. With CALCULATE(), the filter portion is *always* evaluated first (for both simple filters and advanced filters).

Let's start by looking at just the FILTER() portion of this formula:

```
FILTER(Customers, [Total Sales] >= 5000)
```

Here's what's happening in this portion of the formula:

1. FILTER() creates a row context for the Customers table. FILTER() then goes to row 1 in the Customers table and applies a filter to that single customer.

2. **This is very important:** Because of the implicit CALCULATE() inside the measure [Total Sales], context transition occurs, and the row context is converted to an equivalent filter context. Because of the context transition, the filter then propagates down through the relationship from the Customers table to the Sales table and, hence, filters the Sales table so that only sales for this one customer are unfiltered. Do you remember this from the end of Chapter 10?
3. The measure [Total Sales] is evaluated after the Sales table is filtered for this one single customer.
4. FILTER() then asks the question "Is the value of [Total Sales] for this one customer in this first row of the Customers table greater than or equal to $5,000?" If the answer is yes, then this customer survives the filter test, and the row is kept in the final table result. If the answer is no, the customer is discarded from the final table result.
5. FILTER() then moves to the second row in the row context of the Customers table. Because of the implicit `CALCULATE()` inside the measure [Total Sales], the row context is converted to a filter context (context transition), and the filter propagates the filter for the second customer from the Customers table through to the Sales table, evaluates the measure [Total Sales] against the rows in the Sales table that remain, and then checks whether [Total Sales] for this second customer is greater than $5,000. Once again, if the answer is yes, the customer survives the filter and remains in the final table results. Any customer that fails the test is discarded.
6. FILTER() proceeds through every customer in the Customers table, testing each one to see if the [Total Sales] of all the records for that specific customer in the Sales table is greater than $5,000. All customers that pass the test are retained in the final table, and the ones that fail the test are discarded from the final table.

When the `FILTER()` portion finishes its work, `FILTER()` returns a table of customers that passed the test. You can imagine the `myFilter` portion of the original formula looking like this:

```
Customers with Sales Greater Than $5,000
    = CALCULATE(
        COUNTROWS(Customers),
        Only_Use_The_Table_of_Customers_Provided_By_FILTER
    )
```

The `FILTER()` formula above determines which customers passed the test and returns this table of customers to `CALCULATE()`. The resulting filtered table of customers is then accepted and applied as a filter by CALCULATE(). Finally, CALCULATE() evaluates the COUNTROWS(Customers) potion of the formula. There are actually 1,732 customers out of a total of 18,484 customers that passed the `FILTER()` test. By the time `COUNTROWS()` is executed, just the 1,732 customers that passed the `FILTER()` test remain, so `COUNTROWS()` returns 1,732.

Practice Exercises: FILTER()

Set up a new matrix with `Products[Category]` on Rows and `[Total Sales]` on Values and then write the following two formulas. Find the solutions to these practice exercises in "Appendix A: Answers to Practice Exercises" on page 178.

59. [Total Sales of Products That Have Some Sales but Less Than $10,000]

What you need to do here is get `FILTER()` to iterate over the `Products` table to determine whether each product has some sales and also whether the total of those sales is less than $10,000.

Note that you can use the double ampersand operator (`&&`) if you need more than one condition in your filter expression:

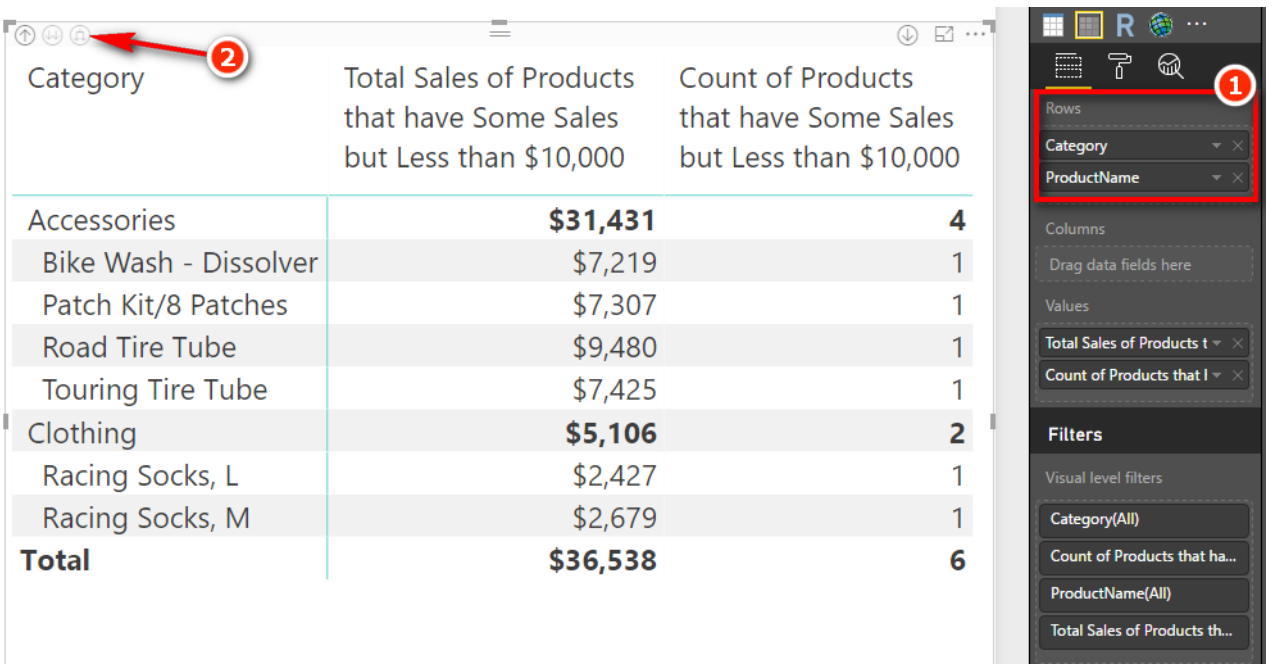
```
Condition1 && Condition2
```

Alternatively, you can use two separate `FILTER` functions.

60. [Count of Products That Have Some Sales but Less Than $10,000]

You should end up with a matrix like the one below.

Category	Total Sales	Total Sales of Products that have Some Sales but Less than $10,000	Count of Products that have Some Sales but Less than $10,000
Accessories	$700,760	$31,431	4
Bikes	$28,318,145		
Clothing	$339,773	$5,106	2
Total	**$29,358,677**	**$36,538**	**6**



Revisiting Filter Propagation

Earlier in this chapter, we looked at a `FILTER()` example with a measure on the left side of the filter test and a value on the right side:

```
Customers with Sales Greater Than $5,000
    = CALCULATE(COUNTROWS(Customers),
    FILTER(Customers,
        [Total Sales] >= 5000
    )
)
```

Let's revisit how `FILTER()` operates in the formula above. (You can't hear this too many times.) The `FILTER()` portion is evaluated first:

1. FILTER() creates a row context for the Customers table. FILTER() then goes to row 1 in the Customers table, and because of the implicit CALCULATE() inside the measure [Total Sales], context transition occurs, converting the row context from FILTER() into an equivalent filter context.
2. Because of the context transition, the filter on the first row of the Customers table propagates down through the relationship with the Customers table and filters the Sales table so that only sales for this customer are unfiltered.

3. The measure [Total Sales] is evaluated after the Sales table is first filtered for this one single customer (returning the value $8,249 in this case, as you should see in the results of your practice exercise).

Context Transition Revisited

Let me go over this again as it is very important. Do you remember what the formula for `[Total Sales]` is? The formula is as follows:

```
[Total Sales] = SUM(Sales[ExtendedAmount])
```

Given that [Total Sales] evaluates to exactly the same result as `SUM(Sales[ExtendedAmount])`, what do you expect will happen if you substitute `SUM(Sales[ExtendedAmount])` into the formula above, like this:

```
Customers with sales greater than $5,000 Version2
    = CALCULATE(COUNTROWS(Customers),
     FILTER(Customers,
          SUM(Sales[ExtendedAmount]) >= 5000
     )
)
```

You should go ahead and create this formula and see what happens. When you do this, you create a matrix like the one below.

Category	Total Sales	Customers That Have Purchased	Customers with Sales Greater Than $5,000	Customers with sales greater than $5,000 Version2
Accessories	$700,760	15,114		18,484
Clothing	$339,773	6,852		18,484
Bikes	$28,318,145	9,132	1,712	18,484
Total	**$29,358,677**	**18,484**	**1,732**	**18,484**

The new formula returns the entire table of customers instead of the value you are looking for. Why is this? There is a very important difference between a measure and the formula inside a measure. Technically speaking, when you write the following measure:

```
Total Sales = SUM(Sales[ExtendedAmount])
```

what is actually happening under the hood is the following:

```
Total Sales = CALCULATE(SUM(Sales[ExtendedAmount]))
```

Power BI adds a `CALCULATE()` function and wraps it around your formula. You can't see this `CALCULATE()`, but it is there. We call this "invisible" `CALCULATE()` an "implicit `CALCULATE()`." Okay, let's get back to the problem at hand. Go back into the new measure you just created and wrap the `SUM()` function inside a `CALCULATE()` as follows:

```
Customers with Sales Greater Than $5,000 Version2
    = CALCULATE(COUNTROWS(Customers),
    FILTER(Customers,
        CALCULATE(SUM(Sales[ExtendedAmount])) >= 5000
    )
)
```

When you manually place `CALCULATE()` like this, it is called an "explicit" `CALCULATE()`. Once you make this change, you get the expected result, as shown below.

Country	Total Sales	Customers That Have Purchased	Customers with Sales Greater Than $5,000	Customers with sales greater than $5,000 Version2
Australia	$9,061,001	3,591	719	719
Canada	$1,977,845	1,571	42	42
France	$2,644,018	1,810	163	163
Germany	$2,894,312	1,780	158	158
United Kingdom	$3,391,712	1,913	280	280
United States	$9,389,790	7,819	370	370
Total	**$29,358,677**	**18,484**	**1,732**	**1,732**

Note: I have swapped the column Products[Category] with Territories[Country] to show an alternate view of the data. The measures work regardless of which column you have on Rows on the matrix.

The point is that without the CALCULATE() function wrapped around SUM(Sales[ExtendedAmount]), something stops working. It doesn't matter if there is an implicit `CALCULATE()` that you can't see (inside another measure) or if there's an explicit `CALCULATE()` that you add yourself. You simply must have a `CALCULATE()` if you want this formula to work. Why is this?

Remember that Chapter 10 said that a row context does not automatically create a filter context. Chapter 10 was talking about the row context in a calculated column, but it is exactly the same *in a function* that has a row context—in this case, the FILTER() function. The CALCULATE() function tells Power BI to "run the filter engine again." If you don't have this extra `CALCULATE()`, the filter that is first applied to the `Customers` table *will not* propagate to the `Sales` table as described above. It is the second `CALCULATE()` (either implicit or explicit) that causes the filter on the `Customers` table to propagate through the relationship to the `Sales` table *before* the rest of the `FILTER()` expression is evaluated for each row in the table.

So let's step through what happens without the extra `CALCULATE()` by going back to this version, which didn't work:

```
Customers with sales greater than $5,000 Version2
    = CALCULATE(COUNTROWS(Customers),
    FILTER(Customers,
        SUM(Sales[ExtendedAmount]) >= 5000
    )
)
```

Of course, the `FILTER` portion is evaluated first, just as before. And then the following happens:

1. FILTER() creates a row context on the Customers table.
2. FILTER() then goes to the first row in the table. No filter context currently exists because a row context doesn't automatically create a filter context.
3. SUM(Sales[ExtendedAmount]) is then evaluated over the entire Sales table for *all customers* and returns the value $29,358,677. There is no filter context, and hence the `Sales` table is completely unfiltered. Since $29 million is greater than $5,000, this customer survives the filter test.
4. FILTER() then goes to the next customer, and exactly the same thing happens again. It doesn't matter what customer is selected in the row context in the Customers table; there is no filtering through to the Sales table, and therefore the result of every iteration step is that every customer passes the test. Because they all pass the test, all customers are returned in the answer.

Note: The only way to create a filter context in this example is if you use a second CALCULATE() function (implicit or explicit) wrapped around `SUM(Sales[ExtendedAmount])` to tell Power BI to convert the row context into a filter context; then you need to propagate the filter down through the existing relationships before completing the evaluation. If this CALCULATE() (implicit or explicit) is omitted, then *no filtering happens*, and, as a result, the Sales table is completely unfiltered.

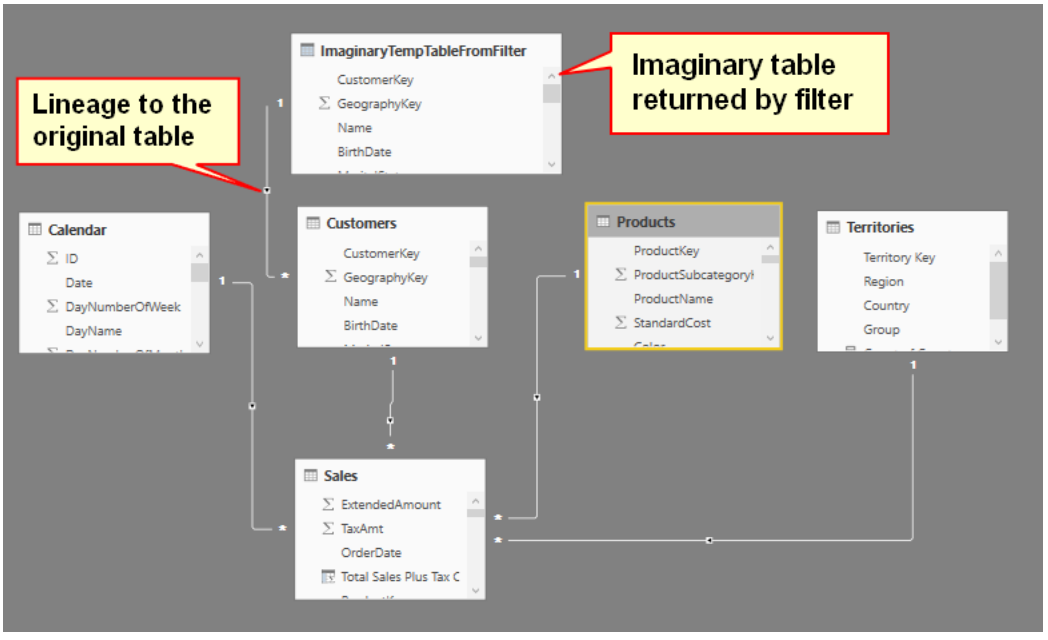
This topic can take some time to get your head around. If this is not crystal clear, I recommend that you go back through this chapter (and also Chapter 10) and work through the examples again until it is clear in your mind.

Virtual Table Lineage

Let's jump back to the version of the measure we've been working on that works:

```
Customers with Sales Greater Than $5,000
    = CALCULATE(COUNTROWS(Customers),
    FILTER(Customers,
        [Total Sales] >= 5000
    )
)
```

When you think about the new table returned by `FILTER()` in the above formula, your first inclination may be to think about it as a standalone table, but that's not the case. Any virtual table object returned by a table function in DAX always retains a link to the data model for the life of the evaluation of the measure or calculated column; this is called *lineage*. I find it useful to visualise an imaginary temporary table being created above the real table in the data model, as shown below.



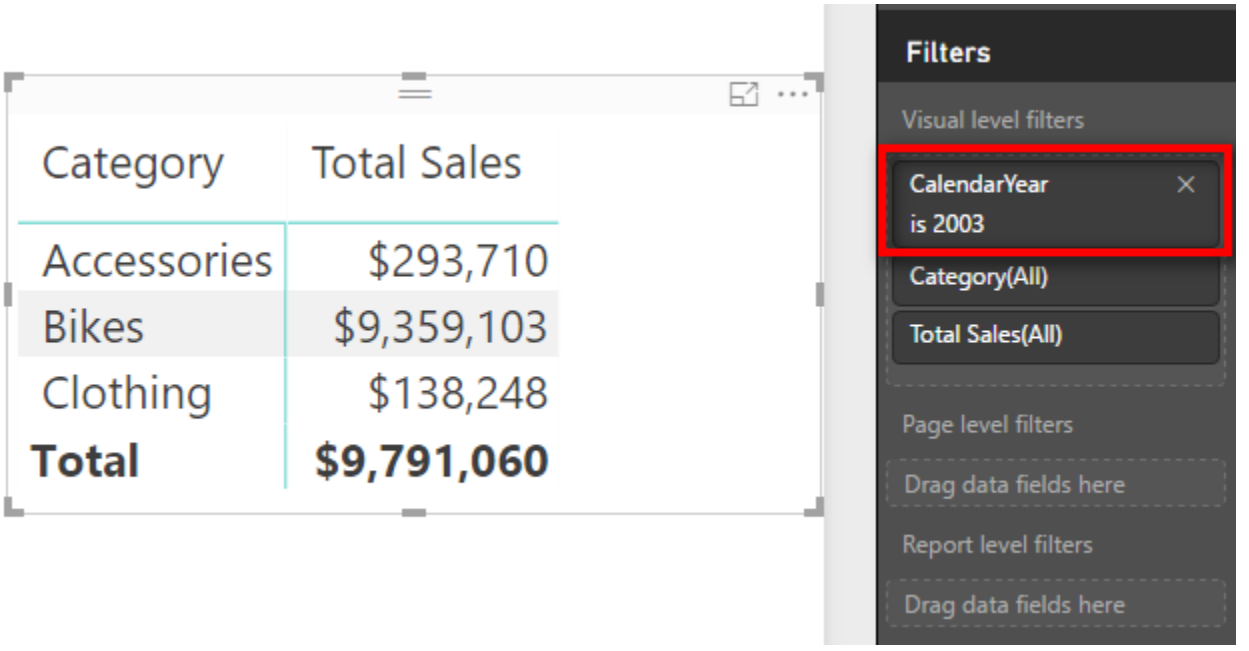
This new temporary table contains a subset of rows as determined by `FILTER()`, but, importantly, it has the link back to the original table (lineage). Therefore, when this new temporary table is used inside `CALCULATE()` (as is the case here), `CALCULATE()` tells Power BI to run the filter propagation again, and this new table therefore filters the original table and any other tables that are connected downstream to the original table.

Remember that the image above is just an illustration of what is happening under the hood. The imaginary table is never materialised, and you can't see it, but it behaves as if it were part of the data model, as illustrated here.

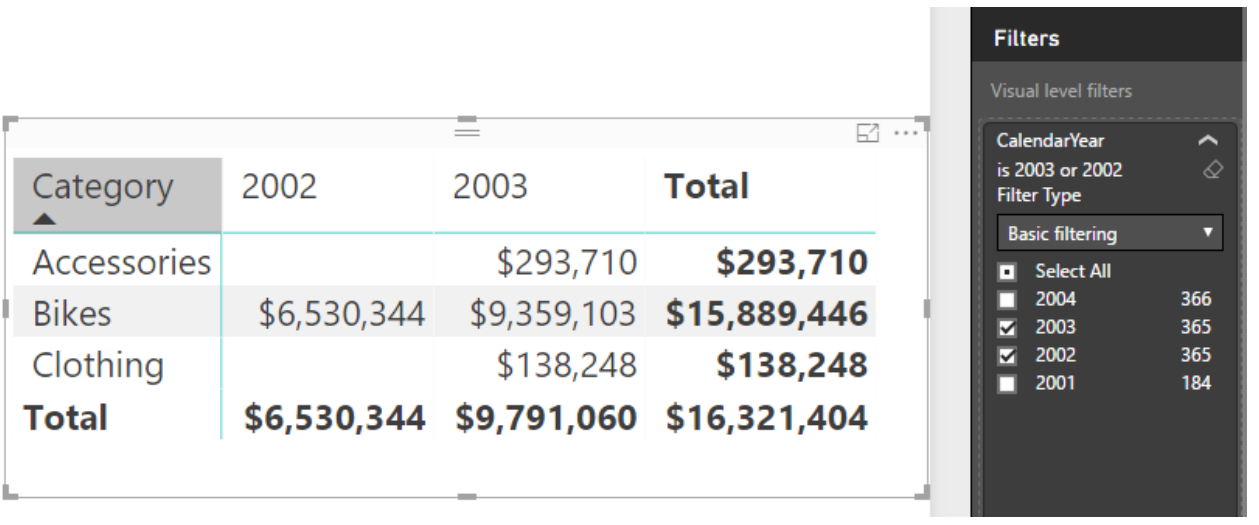
Note: Don't confuse lineage as described above with what happens when you use the New Table button in Power BI. Lineage only happens with virtual tables used inside DAX formulas such as measures and calculated columns. At the end of the execution of a DAX formula, a virtual table ceases to exist, and the lineage is gone. On the other hand, the New Table button creates a new (physical) table that is stored in the data model. This new table does not retain lineage to the table it came from. If you want a table created by the New Table button to filter your data model, you need to connect the new table by using a relationship in Relationships view.

15: DAX Topic: Time Intelligence

Time intelligence is a very important and powerful feature in DAX. *Time intelligence* refers to the ability to write formulas that refer to other time periods within a visual without needing to change the time filters. Consider the following matrix, which shows sales for the year 2003.



Now what if you wanted to see the sales for the preceding year as well as the change in sales compared to the preceding year? Well, one thing you could do is to toggle the filter between the calendar years 2003 and 2002 to see the results for the preceding year, or you could bring `CalendarYear` and place it on Columns and then filter out the years that you are not interested in, as shown below.



But doing it this way is a bit of a hack, and it isn't reusable in other matrixes without doing further hacks. And besides, you can't calculate the change compared to the preceding year.

Using Time Intelligence Functions

You can use time intelligence functions to create new relative measures, such as `[Total Sales Last Year]`, as discussed above without having to change the date selections in the matrix to see the prior year. This makes everything easier to do, and it also means you can build visuals, like the one shown below, that would not be possible any other way.

CalendarYear	Total Sales	Change in Sales vs Prior Year
2001	$3,266,374	$3,266,374
2002	$6,530,344	$3,263,970
2003	$9,791,060	$3,260,717
2004	$9,770,900	-$20,161
Total	**$29,358,677**	**$9,770,900**

DAX comes bundled with a number of inbuilt time intelligence functions, and you can also write custom time intelligence functions yourself when needed. There are some limitations to the inbuilt time intelligence functions, and they work only under certain circumstances. These are a couple of the rules for using inbuilt time intelligence functions:

- You must have a Calendar table that contains a contiguous range of dates that covers every day in the period you are analysing. Every date must exist once and only once in the Calendar table. You can't skip any dates (e.g., you can't skip weekend dates just because you don't work weekends).
- Inbuilt time intelligence works only on a standard calendar—that is, a calendar like one that you might hang on a wall, where the start of the year is January 1, the end of the year is December 31, the last day of May is May 31, etc. A standard calendar can also be customised for different financial years (e.g., you can set the end date for a calendar to be June 30 instead of December 31, or any other date for that matter).

If for some reason these rules can't be met, then you can't use the inbuilt time intelligence functions. In such a case, you can write your own custom time intelligence functions from scratch, using `FILTER()`. The DAX for this tends to be a bit complex, but don't worry, you can learn it and I explain it later in this chapter.

Nonstandard Calendars

In some cases, you may need to use a nonstandard calendar for your reports. These are some examples of when you could not use a standard calendar:

- When you are building a data model using weekly or monthly time periods and using a weekly or monthly calendar instead of a daily calendar. (Note that you could load your data weekly or monthly and still use a daily calendar—and this would still work with inbuilt time intelligence, as long as all the other criteria were still met.)
- If you use an ISO or 445 calendar for your accounting periods. This is very common in the retail industry, where businesses want to have regular trading periods. In the case of a 445 calendar, there are 2 months that consist of 4 calendar weeks followed by 1 month with 5 calendar weeks. This helps smooth the months so they all start on a Monday and finish on a Sunday (for example) while also having 91 days in the quarter (91 × 4 quarters = 364 days)
- With 13 4-week periods instead of calendar months.
- If you had a calendar that uses time as well as date (e.g., an hourly calendar).

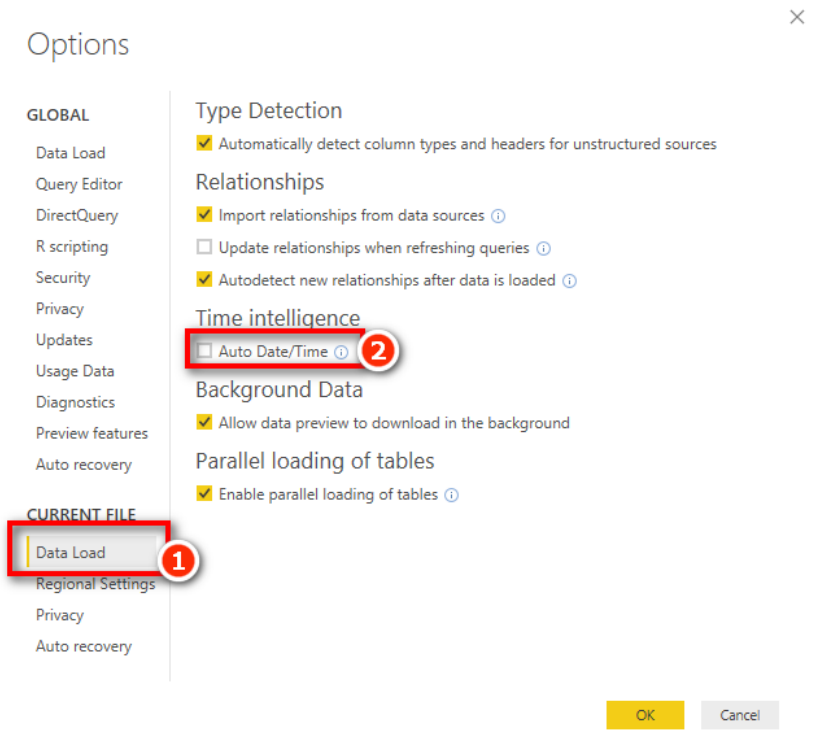
There are so many variations that it is impossible to mention them all here, and it is also impossible for Power BI to cater for them all with inbuilt functions. So the rule is, if you have a standard calendar, you can use the inbuilt functions. If you don't have a standard calendar, then you need to write your own custom time intelligence using `FILTER()`.

Here's How: Turning Off Auto Date/Time

At this writing, Power BI has a feature called Auto Date/Time that acts like an automatic time intelligence feature for anyone who doesn't want to learn about calendar tables and time intelligence. I personally do not like this feature. It automatically creates time intelligence–like behaviour for every table in your data model that has a date column—which can make your data models very large very quickly. Besides, you are learning to write DAX, so why not learn to do it properly with a dedicated calendar table? I recommend that you turn off the Auto Date/Time feature. If you want to do so, follow these steps:

1. Select File, select Options and Settings, and select Options.
2. Navigate to the Current File section and choose Data Load (see #1 below).
3. Uncheck Auto Date/Time (#2).

At this writing, it is not possible to turn off the Auto Date/Time feature by default; you must do it for each workbook.



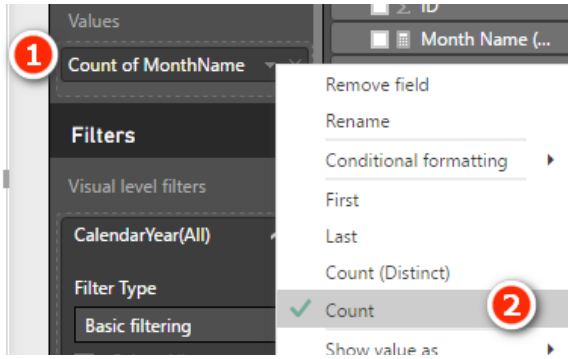
Inbuilt Time Intelligence

Before using the inbuilt time intelligence functions, you need to validate that the prerequisite requirements are covered.

Using a Contiguous Date Range

In the sample data that you have been working with, the `Calendar` table already contains all the days of the year for the period that covers the `Sales` table. It is easy to check this. Just create a new matrix, put `'Calendar'[CalendarYear]` on Rows, and drop any string-based column (such as `MonthName`) into the Values section. After adding `MonthName` to the Values section (see #1 below), click the drop-down arrow and change the settings so that Values displays Count (#2).

CalendarYear	Count of MonthName
2001	184
2002	365
2003	365
2004	366
Total	**1280**



You're right. I did tell you never to create implicit measures unless you are just doing a quick test. This is one of those cases where it is fine to use them, though. These are not wrong; it is just that you can't reuse implicit measures inside other formulas. In this case, I don't need to reuse this measure, so it is fine. As you can see above, the `Calendar` table has half a year for 2001 plus a full year for each of the following three years (including a leap year for 2004). Now that I have confirmed the data in my `Calendar` table, I can just remove this implicit measure from my visual as I don't need it any more.

The SAMEPERIODLASTYEAR() Function

Let's look at an inbuilt time intelligence function you can use to easily write the `[Total Sales Last Year]` measure discussed earlier.

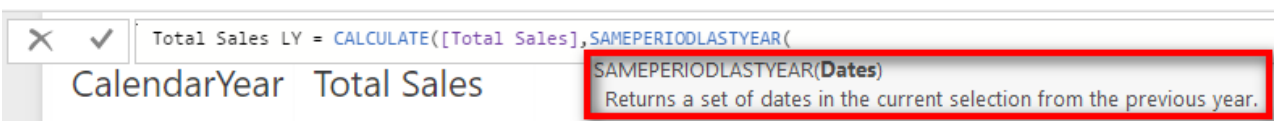
First, set up your matrix like the one below, with `'Calendar'[CalendarYear]` on Rows and `[Total Sales]` on Values.

CalendarYear	Total Sales
2001	$3,266,374
2002	$6,530,344
2003	$9,791,060
2004	$9,770,900
Total	**$29,358,677**

Right-click on the `Sales` table, select New Measure, and write the following measure:

```
Total Sales LY
    = CALCULATE([Total Sales],
        SAMEPERIODLASTYEAR('Calendar'[Date])
    )
```

As shown below, if you pause after typing `SAMEPERIODLASTYEAR(`, IntelliSense says that this function will return a list of dates from the current filter context but time shifted back by a year.



You should recognise that `SAMEPERIODLASTYEAR()` is a table of values and that that table is being used inside `CALCULATE()` as an advanced filter.

> **Note:** The word `Calendar` is a reserved word in Power BI. Calendar is actually a function that will return a calendar table. Personally I never use this function as I think there are better ways to create calendar tables. It is still okay to call a calendar table `Calendar`, but you must always add single quotes when referencing the table inside your formulas, as shown in the formula above.

Also notice in the IntelliSense that `SAMEPERIODLASTYEAR()` takes a single `Dates` parameter as its only input. All inbuilt time intelligence functions ask for this `Dates` parameter, and it always refers to the date column in the `Calendar` table.

How Does SAMEPERIODLASTYEAR() Work?

In Chapter 14 I explained that `CALCULATE()` can take a table as an advanced filter input, and you can imagine the new table being connected to the data model. The table inside `CALCULATE` then filters the rest of the tables in the data model (in this case, the `Calendar` table and the `Sales` table) before `CALCULATE()` completes the calculation. It is exactly the same with `SAMEPERIODLASTYEAR()`, as shown below:

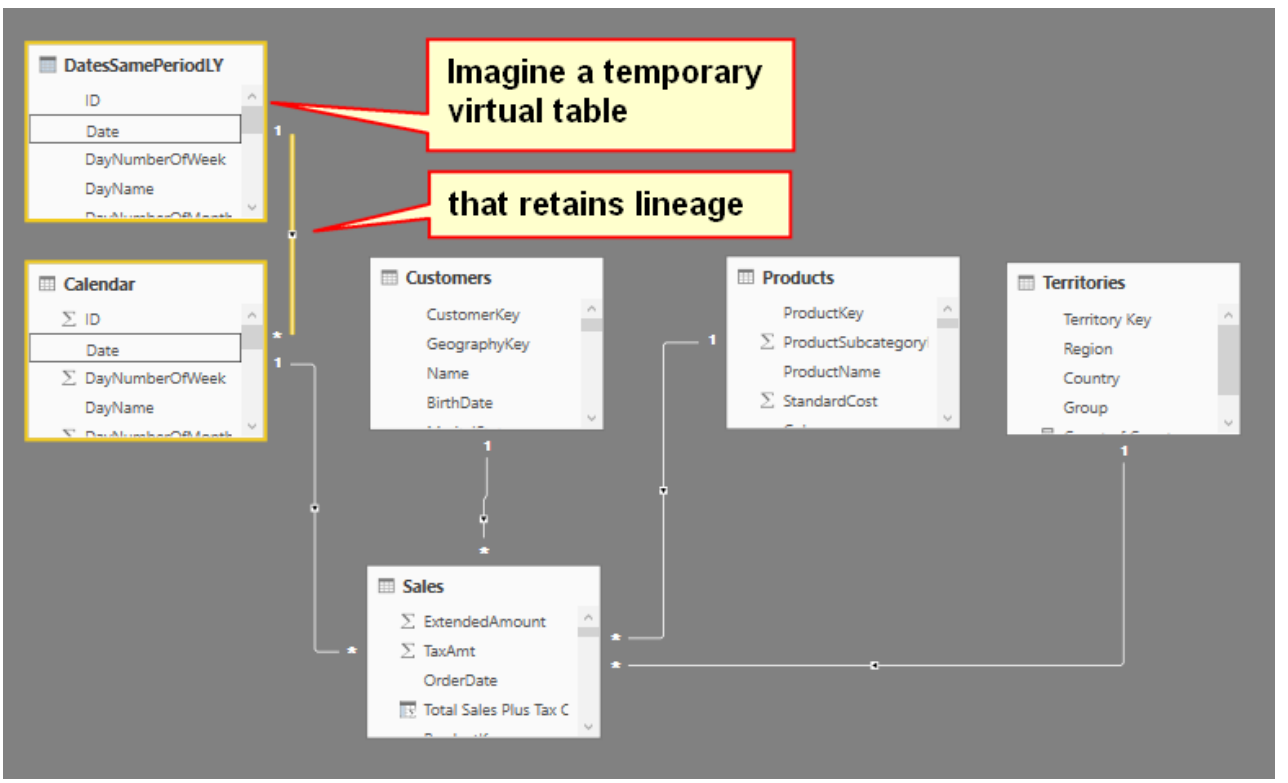
```
Total Sales LY
    = CALCULATE([Total Sales],
        SAMEPERIODLASTYEAR('Calendar'[Date])
    )
```

In this instance, `SAMEPERIODLASTYEAR()` returns a table of dates that are the same dates coming from the matrix for the selected year, but `SAMEPERIODLASTYEAR()` time shifts the original dates back by one year.

Consider the cell highlighted in the matrix below. The function `SAMEPERIODLASTYEAR()` first reads the filter context from the current matrix to see which dates apply for "this year." In this case, the filter is on `CalendarYear`, and the filter for this cell is 2003 (see #1 below). So the dates for "this year" are all dates from January 1, 2003, through to December 31, 2003. The `SAMEPERIODLASTYEAR()` function then takes the dates from the current filter context in the matrix, removes the current filters, and then time shifts them back one year before returning a table of dates from January 1, 2002, through December 31, 2002.

CalendarYear	Total Sales	Total Sales LY
2001	$3,266,374	
2002	$6,530,344	$3,266,374
2003 ①	$9,791,060	$6,530,344
2004	$9,770,900	$9,791,060
Total	**$29,358,677**	**$19,587,777**

You can imagine the new table created by `SAMEPERIODLASTYEAR()` as a temporary table sitting above the `Calendar` table and retaining a relationship to the original `Calendar` table, as shown below. Remember that this is logically how it works; you can't actually see this table.

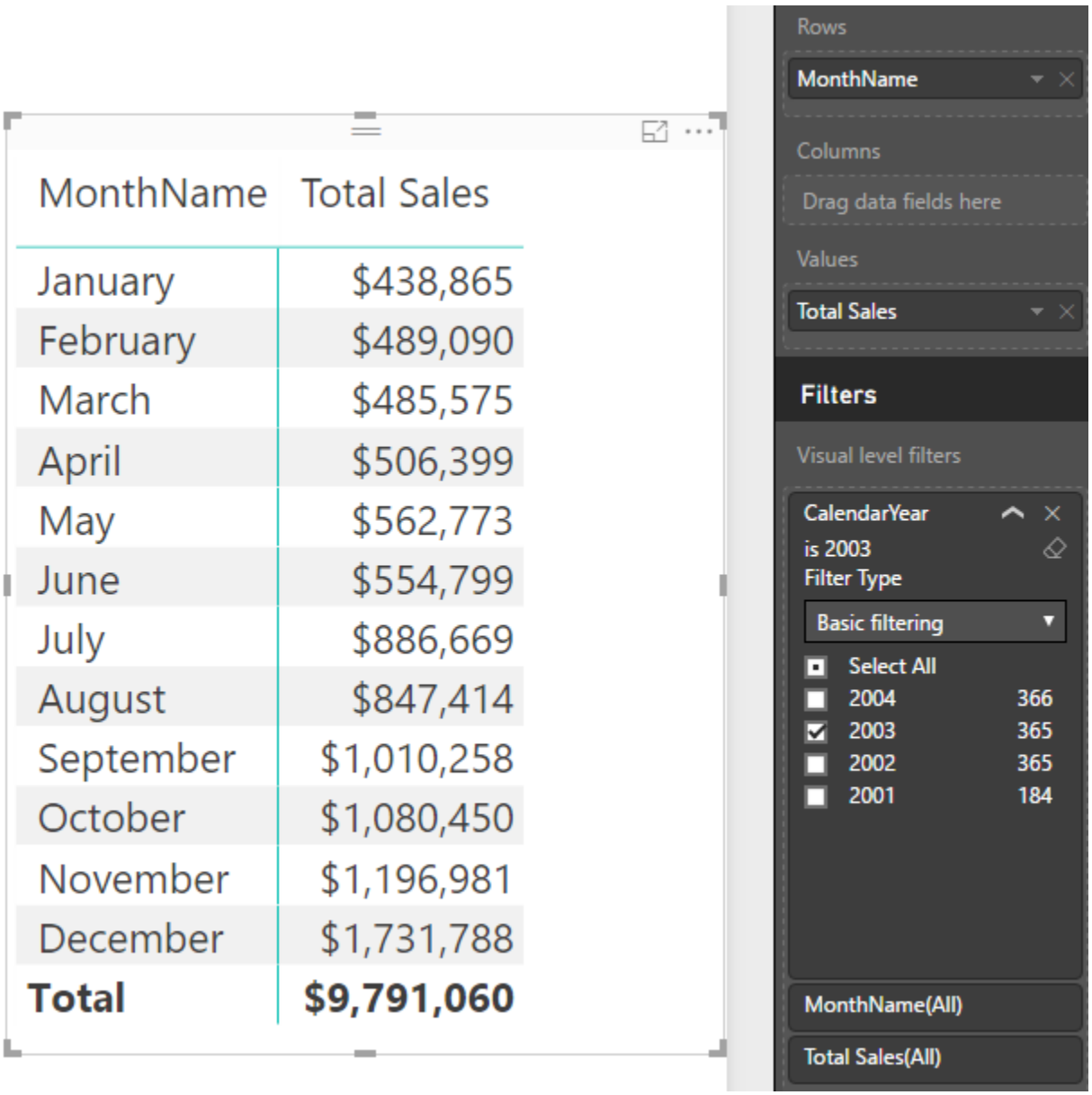


This table is then passed to `CALCULATE()`, and `CALCULATE()` uses this temporary table to rerun the filter propagation. The temporary table (the table of dates from `SAMEPERIODLASTYEAR()`) filters the `Calendar` table, which then filters the `Sales` table before the calculation for `[Total Sales LY]` is evaluated.

Tip: Read the paragraph above a couple of times if you need to until you have it clearly in your head.

Calculating Sales Year to Date

A very common business need is to calculate figures on a year-to-date (YTD) basis. Fortunately, there is an inbuilt function for this. Before you write any YTD formula, it is a good idea to set up a matrix that will give you immediate feedback if your formula is performing as expected. It is also important to set up your matrix so that you have a continuous date range. Set up a new matrix like the one shown below before proceeding. Note the filter on `CalendarYear = 2003`.



MonthName	Total Sales
January	$438,865
February	$489,090
March	$485,575
April	$506,399
May	$562,773
June	$554,799
July	$886,669
August	$847,414
September	$1,010,258
October	$1,080,450
November	$1,196,981
December	$1,731,788
Total	**$9,791,060**

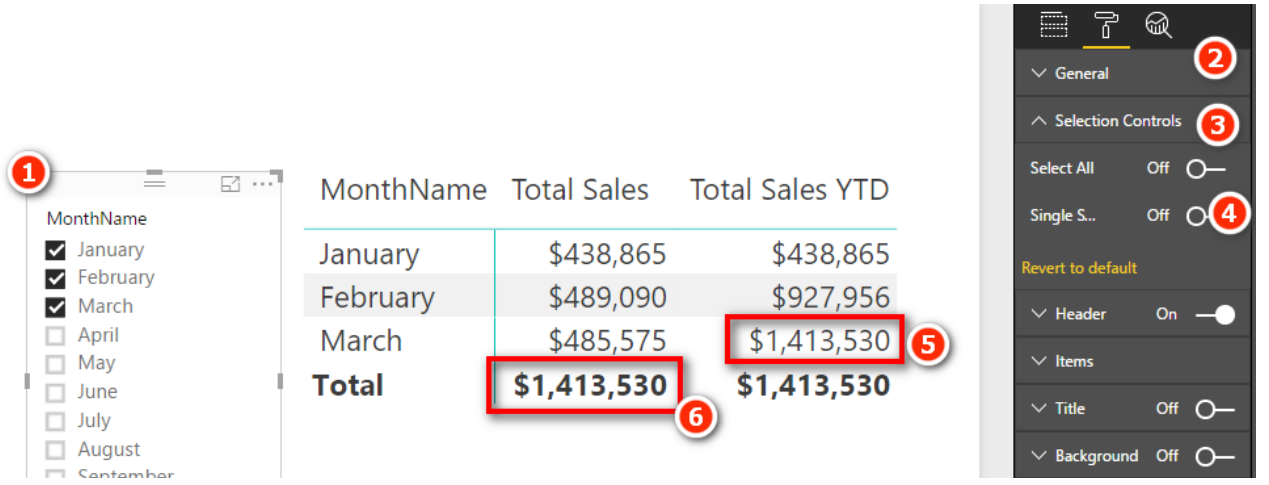
Note how the periods in the matrix are contiguous (i.e., the months of the year 2003). If you didn't have a filter on `CalendarYear = 2003` but instead had `CalendarYear = ALL`, the matrix would show the total sales for January across all years, for February across all years, etc. This would not be a contiguous range, and hence the formula would not work.

Now right-click on the `Sales` table and write the following measure:

```
Total Sales YTD = TOTALYTD([Total Sales], 'Calendar'[Date])
```

Apply appropriate formatting to the measure and then add the measure to your matrix.

When you are done, it is very easy to check whether the formula is working correctly. As you can see below, I have added a new slicer (see #1 below) and turned off Single Select. To do this, go to the Format pane and select General (#2), select Selection Controls (#3), and turn off the Single Select option (#4). You can then select the months January, February, and March in the slicer, and it is easy to compare the YTD value for March YTD (#5) with the summed total of January, February, and March (#6).



Note: It is very important that you test your measures after you write them. You are a data modeller now! Along with this title comes the responsibility to check that the measures you write are returning the expected results.

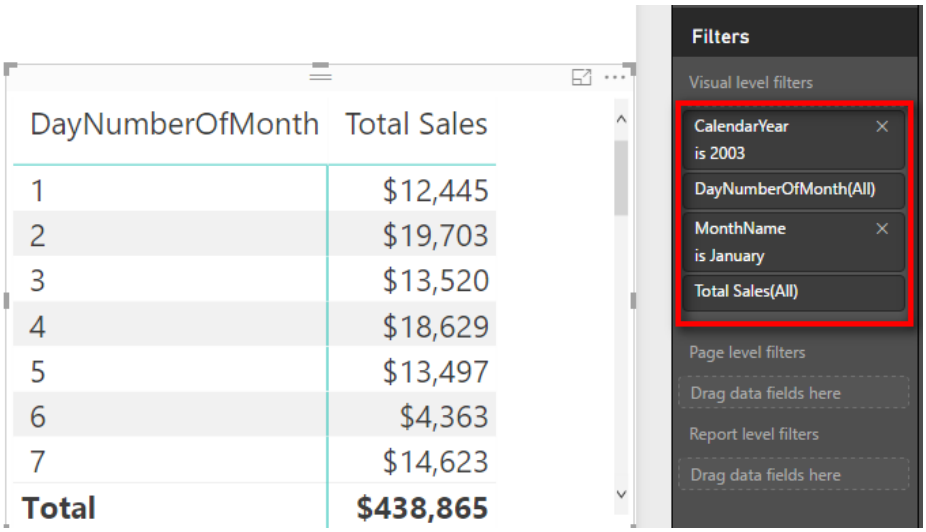
This is a really good example of the benefit of writing measures in the context of a matrix. The immediate feedback you get allows you to check whether your formula is correct and is well worth the effort. Once you have written a formula, you can apply some conditional formatting to your matrix, as shown below, to get another visual clue about whether all is working well.

MonthName	Total Sales	Total Sales YTD
January	$438,865	$438,865
February	$489,090	$927,956
March	$485,575	$1,413,530
April	$506,399	$1,919,930
May	$562,773	$2,482,702
June	$554,799	$3,037,501
July	$886,669	$3,924,170
August	$847,414	$4,771,584
September	$1,010,258	$5,781,842
October	$1,080,450	$6,862,291
November	$1,196,981	$8,059,273
December	$1,731,788	$9,791,060
Total	**$9,791,060**	**$9,791,060**

Practice Exercises: Time Intelligence

When writing the previous formula, you may have noticed from the IntelliSense tooltip that there are two other functions that are very similar: `TOTALMTD()` and `TOTALQTD()`. In this section you'll get some practice using these two functions. Before you do these two exercises, make sure you set up a matrix like the one below that will give you feedback if your formula is correct. Set up the matrix like this:

1. Place `CalendarYear` and `MonthName` on Filter.
2. Filter for CalendarYear = 2003 and MonthName = January.
3. Put 'Calendar'[DayNumberOfMonth] on Rows.



Write formulas for the following measures. Find the solutions to these practice exercises in "Appendix A: Answers to Practice Exercises" on page 178.

61. [Total Sales Month to Date]

62. [Total Sales Quarter to Date]

> **Tip:** Did you set up a matrix with suitable values on rows like I showed with `[Total Sales YTD]` earlier in this chapter? Placing `MonthName` on rows will not work for Practice Exercise 61. Instead you need to put a column such as `DayNumberOfMonth` in the matrix if you want to be able to "see" that the formula is working correctly.

Changing Financial Year-Ending Dates

Many of the inbuilt time intelligence functions allow you to specify a different end-of-year date. In such a case, there will be an optional parameter where you specify the year-end date:

```
Total Sales FYTD
    = TOTALYTD([Total Sales],
    'Calendar'[Date],"YearEndDateHere"
    )
```

Here's an example for a financial year ending June 30:

```
Total Sales FYTD
    = TOTALYTD([Total Sales],
    'Calendar'[Date],"30/6")
```

Note that this example uses a non-U.S. date format. If you are using the U.S. date format, then it would be as follows:

```
Total Sales FYTD USA
    = TOTALYTD([Total Sales], 'Calendar'[Date],"6/30")
```

Notice that there is no need to specify a year when referring to the year-end date. It is simply day and month.

Practice Exercises: Time Intelligence, Cont.

Write formulas for the following measures. Find the solutions to these practice exercises in "Appendix A: Answers to Practice Exercises" on page 178.

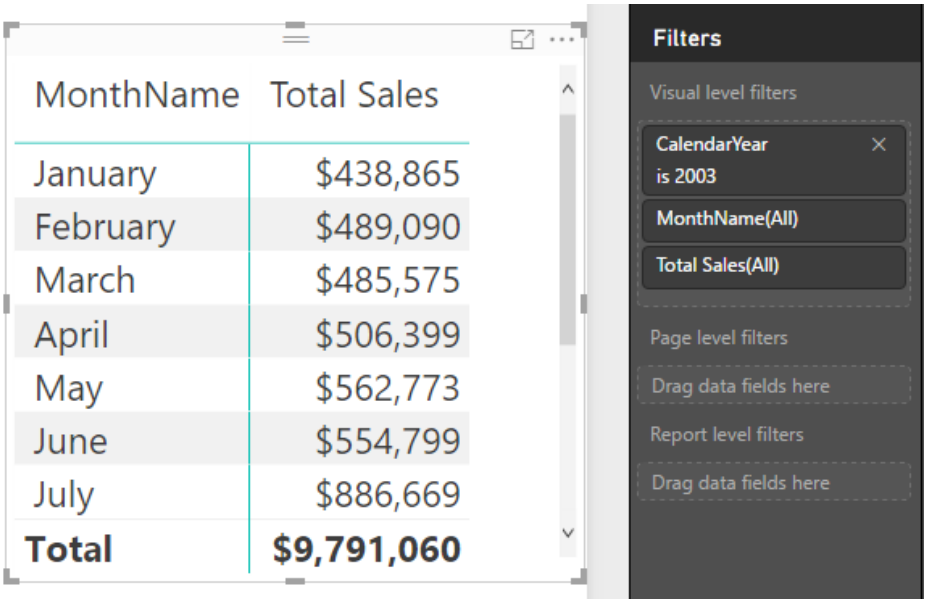
63. [Total Sales FYTD 30 June]

64. [Total Sales FYTD 31 March]

Format your matrix by selecting Conditional Formatting, Data Bars to make it easier to spot the pattern.

Practicing with Other Time Intelligence Functions

There are a lot of inbuilt time intelligence functions, and it's easy to tell what most of them do. `PREVIOUSMONTH()`, `PREVIOUSQUARTER()`, and `PREVIOUSDAY()`, for example, all return tables of dates referring to the previous period and probably don't need further explanation. To see how they work, set up a matrix with contiguous months, as shown below.



Practice Exercises: Time Intelligence, Cont.

Write the following formulas. Find the solutions to these practice exercises in"Appendix A: Answers to Practice Exercises" on page 178.

> **Tip:** As always, I suggest that you set up a suitable matrix with a suitable column on Rows and `[Total Sales]` on Values before writing these measures. Doing this will help you check whether the measures are working and also should help you comprehend how the measures work. If you don't understand, go back and re-read the section about `SAMEPERIODLASTYEAR()`. These functions in this practice section work in exactly the same way.

65. [Total Sales Previous Month]

Given that the `PREVIOUSMONTH()` function will return a table of dates, you need to embed the time intelligence formula inside a `CALCULATE()` function.

66. [Total Sales Previous Day]

You need to set up a suitable matrix that gives you immediate feedback about whether your formula is working. Put 'Calendar'[DayNumberOfMonth] on Rows and make sure you filter for a single month.

67. [Total Sales Previous Quarter]

As with Practice Exercise 66, you need to set up a suitable matrix for context. You can work out how to do this one yourself.

Writing Your Own Time Intelligence Functions

As mentioned earlier in this chapter, writing your own time intelligence functions is a bit harder than using the inbuilt functions, particularly when you are learning. However, once you get the hang of it, you will find it quite easy, and this will also be a good sign of how much progress you are making in your understanding of DAX.

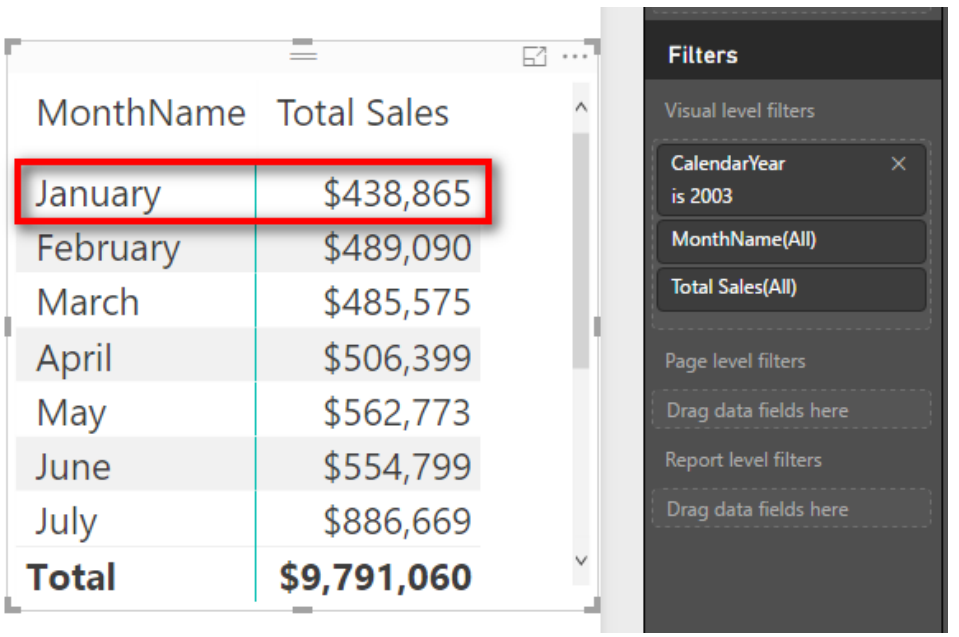
There are a couple of strange things in the syntax that you need to get your head around before you can fully understand what you're doing. The good news is that I explain these things in this section, and you will be writing your own custom time intelligence functions in no time at all. These are the two concepts you need to get your head around:

- Concept 1: Thinking "whole of table" when thinking about filter context
- Concept 2: Knowing how to use MIN() and MAX()

Let me cover these concepts before we get into any examples. That way, by the time you reach the examples, you will be primed and ready to go.

Concept 1: Thinking "Whole of Table" When Thinking About Filter Context

Consider the single row highlighted in the matrix below. (This is the same matrix you were just looking at above.)



This matrix is filtered for 'Calendar'[CalendarYear]= `2003` in the filter. Also, the highlighted row (January) is also filtered—by `'Calendar'[MonthName]="January"`, which appears in the Rows drop zone for the matrix. When these two filters are combined, the single cell/value for `[Total Sales]` is filtered for the period January 2003. So there are only 31 days that are used in the `Calendar` table in the data model for this cell. With this in mind, it is possible to imagine this filter applied on the back end.

> **Tip:** Practice using your imagination to think about what these "filtered" tables would look like after the filters have been applied. (For example, in the example above, the Calendar table would have only 31 days visible.) This is all happening in computer memory, on-the-fly. You can't peek in the back end and see this filtering happening, but it is important that you be able to imagine it happening in your mind. Thinking about what is happening behind the scenes like this will make it easier to write custom time intelligence formulas.

When thinking about the filtering that is being applied, you should think about *the whole table*, not just the two columns with filters applied. It is clear that there is only one month visible (January) and only one year visible (2003), but it is also true that there are 31 `DayNumberOfMonth` values visible (those from 1 through 31), and there are 4 different `WeekNumberOfYear` values (1 through 4). It is possible to reference any and all of these other columns and values in your DAX formulas after the initial filter context is applied, and this makes it very powerful indeed.

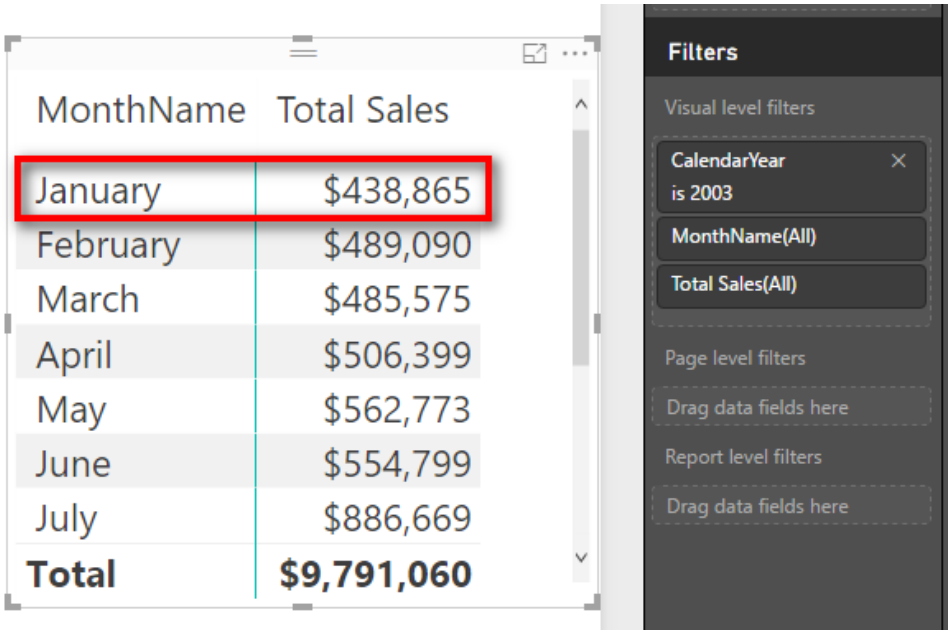
This is one of the main reasons you should also include an ID column in your `Calendar` table if you are going to write custom time intelligence functions. As you can see in the next image, after you filter the `Calendar` table based on January and 2003, there are actually 31 rows in the table, and the ID numbers of those rows run from 550 to 580. You can reference these ID values that remain in the filtered table in your DAX formulas to write very powerful DAX. But you need to be able to think "whole of table" to be able to understand how to do this.

ID	Date	DayNumberOfWeek	DayName	DayNumberOfMonth	DayNumberOfYear	WeekNumberOfYear	MonthName
549	31/12/2002 12:00:00 AM	3	Tuesday	31	365	53	December
550	1/01/2003 12:00:00 AM	4	Wednesday	1	1	1	January
551	2/01/2003 12:00:00 AM	5	Thursday	2	2	1	January
552	3/01/2003 12:00:00 AM	6	Friday	3	3	1	January
553	4/01/2003 12:00:00 AM	7	Saturday	4	4	1	January
554	5/01/2003 12:00:00 AM	1	Sunday	5	5	2	January
555	6/01/2003 12:00:00 AM	2	Monday	6	6	2	January
556	7/01/2003 12:00:00 AM	3	Tuesday	7	7	2	January
557	8/01/2003 12:00:00 AM	4	Wednesday	8	8	2	January
558	9/01/2003 12:00:00 AM	5	Thursday	9	9	2	January
559	10/01/2003 12:00:00 AM	6	Friday	10	10	2	January
560	11/01/2003 12:00:00 AM	7	Saturday	11	11	2	January
561	12/01/2003 12:00:00 AM	1	Sunday	12	12	3	January
562	13/01/2003 12:00:00 AM	2	Monday	13	13	3	January
563	14/01/2003 12:00:00 AM	3	Tuesday	14	14	3	January
564	15/01/2003 12:00:00 AM	4	Wednesday	15	15	3	January
565	16/01/2003 12:00:00 AM	5	Thursday	16	16	3	January
566	17/01/2003 12:00:00 AM	6	Friday	17	17	3	January
567	18/01/2003 12:00:00 AM	7	Saturday	18	18	3	January
568	19/01/2003 12:00:00 AM	1	Sunday	19	19	4	January
569	20/01/2003 12:00:00 AM	2	Monday	20	20	4	January
570	21/01/2003 12:00:00 AM	3	Tuesday	21	21	4	January
571	22/01/2003 12:00:00 AM	4	Wednesday	22	22	4	January
572	23/01/2003 12:00:00 AM	5	Thursday	23	23	4	January
573	24/01/2003 12:00:00 AM	6	Friday	24	24	4	January
574	25/01/2003 12:00:00 AM	7	Saturday	25	25	4	January
575	26/01/2003 12:00:00 AM	1	Sunday	26	26	5	January
576	27/01/2003 12:00:00 AM	2	Monday	27	27	5	January
577	28/01/2003 12:00:00 AM	3	Tuesday	28	28	5	January
578	29/01/2003 12:00:00 AM	4	Wednesday	29	29	5	January
579	30/01/2003 12:00:00 AM	5	Thursday	30	30	5	January
580	31/01/2003 12:00:00 AM	6	Friday	31	31	5	January
581	1/02/2003 12:00:00 AM	7	Saturday	1	32	5	February
582	2/02/2003 12:00:00 AM	1	Sunday	2	33	6	February

Concept 2: Knowing How to Use MIN() and MAX()

It is very common to use the `MIN()` and `MAX()` functions inside `FILTER()` when you write custom time intelligence functions. (You can also use `FIRSTDATE()` and `LASTDATE()` if you prefer.) You'll learn more detail in the examples that follow, but for now there is one key concept about `MIN()` and `MAX()` that you should understand: Whenever you use an aggregation function around a column in a DAX formula, *it will always respect the initial filter context coming from the visual.*

So let's go back to the matrix from before, shown here again for convenience.



You know that the matrix has filtered the `Calendar` table so that only 31 days remain. Given that `MIN()` and `MAX()` always respect the current filter context, what would be the results of the following DAX formulas for the highlighted row in the matrix above? Answer in your head before moving on. It will help you if you can imagine the filtered copy of the table in your head.

1. `=MIN('Calendar'[Date])`
2. `=MAX('Calendar'[Date])`
3. `=MIN('Calendar'[ID])`
4. `=MAX('Calendar'[ID])`

The answer to Question 1 is, of course, January 1, 2003—the first date in the filter context. It's not the first date in the `Calendar` table but the first date in the current filter context. And the answer to Question 2 is January 31, 2003, the last date in the filter context. But, importantly, the answers to Questions 3 and 4 are 550 and 580, respectively, even though this ID column was not part of the filter.

So you can think of `MIN()` and `MAX()` as tools that can "harvest" the value from the current filter context, in any available column across the whole table, and you can use this harvested value in your DAX formulas. Remember this fact about MIN() and MAX() when you get into the examples below.

> **Note:** If you want to validate the answers 550 and 580, go to the `Calendar` table, find the rows January 1, 2003, and January 31, 2003, and check `'Calendar'[ID]` for each row.

Writing Custom Time Intelligence Functions

Now you are going to write a custom version of `[Total Sales YTD]`, using `CALCULATE()` and `FILTER()`. I strongly encourage you to write this formula yourself for practice. There is a lot that can (and will) go wrong when you type your own custom time intelligence functions, and you will need *lots* of practice to get it right. There are square bracket sets, sets of parentheses, new line spacing to make it easier to read, commas to be added in the right places, etc. *So make sure you actually write the following formula on your own computer.* Go ahead and do that now before moving on to the explanation:

```
Total Sales YTD Manual = CALCULATE([Total Sales],
    FILTER(ALL('Calendar'),
        'Calendar'[CalendarYear]=MAX('Calendar'[CalendarYear])
        && 'Calendar'[Date] <=MAX('Calendar'[Date])
    )
)
```

Also make sure you set up a matrix like the one you used earlier so that you can get immediate feedback about whether your formula is correct.

This formula needs a bit of explanation. I have used http://daxformatter.com in the following pages to make it easier to refer to the lines in the formula. I mentioned DAX Formatter in Chapter 9, and you can see here that it is a great tool for helping you read DAX formulas.

You can see below that lines 4 through 8 are all part of a `FILTER()` function because you can see that the `)` on line 8 is left aligned to the `F` in `FILTER()` on line 4.

```
1  Total Sales YTD Manual =
2  CALCULATE (
3      [Total Sales],
4      FILTER (
5          ALL ( 'Calendar' ),
6          'Calendar'[CalendarYear] = MAX ( 'Calendar'[CalendarYear] )
7              && 'Calendar'[Date] <= MAX ( 'Calendar'[Date] )
8      )
9  )
```

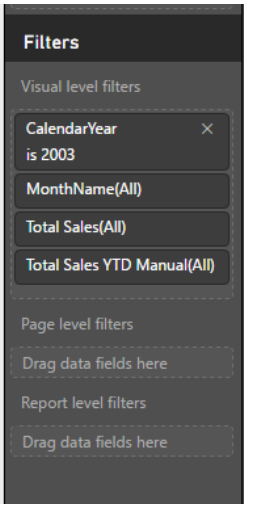
This `FILTER()` function returns a table to the function `CALCULATE()`. `CALCULATE()` then applies a filter for this table of dates and propagates this filter to the `Sales` table prior to evaluating `[Total Sales]`. Let's look more closely at lines 6 and 7 in the `FILTER()` function. Line 6 reads:

```
'Calendar'[CalendarYear] = MAX('Calendar'[CalendarYear])
```

Okay, I hear you saying, "How can the calendar year be equal to the `MAX()` of the calendar year?" What is really happening is that there is *a column name* on the left side of the equals sign, and there is *a* `MAX()` *function* on the right side. Remember from earlier in this chapter that whenever you see `MIN()` or `MAX()` in a formula like this, it always respects the current filter context. So the way to read line 6 of this formula is as follows: "Add a filter to the table so that the column `'Calendar'[CalendarYear]` is equal to the maximum value in my current filter context coming from my matrix."

For example, in the matrix below, the maximum of the highlighted row is March 31, 2003, and hence `MAX('Calendar'[CalendarYear])` = 2003.

MonthName	Total Sales	Total Sales YTD Manual
January	$438,865	$438,865
February	$489,090	$927,956
March	$485,575	$1,413,530
April	$506,399	$1,919,930
May	$562,773	$2,482,702
June	$554,799	$3,037,501
July	$886,669	$3,924,170
Total	**$9,791,060**	**$9,791,060**



See how you need to think "whole of table" here? The initial filter context is applied over the month of March 2003, but the `MAX()` formula is working over the year column. Imagine this filter context acting on the table in your data model by mentally applying the filters: There were 31 rows left in the `Calendar` table, and for each of these rows, the value in `'Calendar'[CalendarYear]` was 2003. As a result (in this case), the `MIN()` of `'Calendar'[CalendarYear]` would also return 2003, as would `SUM()` and `AVERAGE()`, for that matter. So line 6 is really saying "filter my table where `'Calendar'[CalendarYear]` = *the current filter context year*," which is 2003 in this case.

Let's move on. Line 7 starts with the double ampersand operator (which means *and*—i.e., do both line 6 *and* line 7) and then says:

```
'Calendar'[Date] <=MAX('Calendar'[Date])
```

The same applies here as with line 6. `MAX('Calendar'[Date])` reads the initial filter context from the matrix and hence returns the value March 31, 2003, for the highlighted row in the matrix. Therefore, this

part of the formula adds an AND condition so that the underlying table is filtered for 'Calendar'[CalendarYear] = 2003 *and* also for the condition 'Calendar'[Date] is *on or before* March 31, 2003. As you can deduce, this is all the dates year to date.

As you go to the next row in the matrix, the calendar year stays the same, but the month-end date moves to the end of the next month. So the number of days that are included increases as you work down the rows in the matrix.

Now it is important to point out that you could not use MIN() in line 7 as you could do in line 6; this time it has to be MAX(). If you used MIN(), you would get March 1, 2003, as the last date, and the year-to-date result would be out by almost a full month of sales. It is important to think about what your formulas need and make sure you provide the right formulas to achieve that outcome (of course).

Now let's go back to ALL('Calendar'). Line 5 of the formula refers to ALL('Calendar') instead of just the Calendar table (which you have used previously). ALL(), as discussed in Chapter 13, is the "remove filter" function. (If necessary, go back and refresh your memory about ALL() before moving on.)

It is important to use the ALL() function here because you know that the matrix reads the current filter context before doing the calculation. Probably the easiest way to explain why you need the ALL() function is to consider what would happen if you didn't use ALL().

Consider again the highlighted row in the matrix above. You know that the initial filter context for this row of the matrix is for all 31 days in the month of March 2003. You can "imagine" that the Calendar table is filtered behind the scenes so that only these 31 days of March 2003 are visible.

Now let's look at why the [Total Sales YTD] formula does not work without the ALL() function. You should write the following formula and add it to your matrix, as shown below. (Don't miss the opportunity to practice now!)

```
1  Total Sales YTD Doesn't Work =
2  CALCULATE (
3      [Total Sales],
4      FILTER (
5          Calendar,
6          'Calendar'[CalendarYear] = MAX ( 'Calendar'[CalendarYear] )
7              && 'Calendar'[Date] <= MAX ( 'Calendar'[Date] )
8      )
9  )
```

MonthName	Total Sales	Total Sales YTD Doesn't Work
January	$438,865	$438,865
February	$489,090	$489,090
March	$485,575	$485,575
April	$506,399	$506,399
May	$562,773	$562,773
June	$554,799	$554,799
July	$886,669	$886,669
Total	**$9,791,060**	**$9,791,060**

You can see in the matrix above (and in the one you have created yourself) that this formula is giving the sales *for the current month rather than YTD* in each row of the matrix. The reason it doesn't work is related to the initial filter context discussed earlier. For the row of March 2003, the initial filter context applied a filter so that only the 31 days of March 2003 were "visible" in the Calendar table (behind the scenes). So how can the formula possibly return sales for all days "year to date," including the sales from January and February? The dates in January and February were already filtered out by the matrix from the initial filter context, so you can't get the sales for these months to somehow reappear for the new formula if you write it this way.

If you want to include sales from January and February in the row next to the actual sales for March, you must first "remove the filter" created by the matrix. This is what `ALL()` does when it is wrapped around the `Calendar` table in line 5: It removes the filter context that comes from the matrix that is automatically applied to the `Calendar` table. You then reapply the filters you want to use in lines 6 and 7 so that you end up with all the dates YTD.

Note: Custom time intelligence always uses some form of `ALL('Calendar')` to remove the initial filter context. The `FILTER()` function therefore iterates through an unfiltered copy of the `Calendar` table. But the `MIN()` and `MAX()` functions operate in the initial filter context *before* the `ALL()` function removes it.

Tip: Go back and read this section again if necessary until you understand it well.

Now let me come back to that `ID` column I talked about earlier. A good `ID` column in a `Calendar` table starts at 1 and increments by 1 for each row in the table. So in the case of this `Calendar` table, each day of the year has an `ID` value that increments by 1. But the same applies to 445 calendars and weekly calendars. You should always have an `ID` column that increments by 1 for each row in the table (in chronological order, of course). This gives you a nice clean numeric column to move back and forward inside your formulas. To illustrate this point, the following formula will work for YTD:

```
Total Sales YTD Manual ID = CALCULATE([Total Sales],
    FILTER(ALL('Calendar'),
        'Calendar'[CalendarYear]=MAX('Calendar'[CalendarYear])
        && 'Calendar'[ID] <=MAX('Calendar'[ID])
    )
)
```

Notice that here you replace the `Date` column with the `ID` column. Using the `ID` column like this is very powerful and allows you to jump back and forward in time, using your knowledge of the `Calendar` table structure by just doing numeric addition and subtraction on the `ID` column.

For one more example using the `ID` column, write a measure that returns the total sales for the same period last year. You did this earlier, using the function `SAMEPERIODLASTYEAR()`, but recall that this inbuilt time intelligence function works only for a standard calendar. You can also write a custom time intelligence function that works with a custom calendar using `FILTER()`. Note in the matrix below that `[Total Sales LY]` works on both the month level and the year level.

CalendarYear	Total Sales	Total Sales LY ID
2002	**$6,530,344**	**$3,266,374**
January	$596,747	
February	$550,817	
March	$644,135	
April	$663,692	
May	$673,556	
June	$676,764	
July	$500,365	$473,388
August	$546,001	$506,192
September	$350,467	$473,943
October	$415,390	$513,329
November	$335,095	$543,993
December	$577,314	$755,528
2003	**$9,791,060**	**$6,530,344**
January	$438,865	$596,747
February	$489,090	$550,817
March	$485,575	$644,135
Total	**$29,358,677**	**$19,614,172**

Here is the formula you need to write for this:

```
Total Sales LY = CALCULATE([Total Sales],
    FILTER(ALL('Calendar'),
        'Calendar'[ID] >=MIN('Calendar'[ID]) -365 &&
        'Calendar'[ID] <=MAX('Calendar'[ID]) - 365
    )
)
```

Note that you can use the `ID` column to your advantage here to move back in time by 365 days. Also note how the first reference inside `FILTER()` is to `MIN('Calendar'[ID])`, and the second one is to `MAX('Calendar'[ID])`. It's time to think "whole of table" again. Let's take a look at two different areas of the following matrix.

CalendarYear	Total Sales	Total Sales LY ID
2003	**$9,791,060**	**$6,530,344**
January	$438,865	$596,747
February	$489,090	$550,817
March	$485,575	$644,135
April	$506,399	$663,692
May	$562,773	$673,556
June	$554,799	$676,764
July	$886,669	$500,365
August	$847,414	$546,001
September	$1,010,258	$350,467
October	$1,080,450	$415,390
November	$1,196,981	$335,095
December	$1,731,788	$577,314
2004	**$9,770,900**	**$9,817,454**
January	$1,340,245	$438,865
February	$1,462,480	$499,127
March	$1,480,905	$494,572
Total	**$29,358,677**	**$19,614,172**

In the matrix cell marked #1 above (October 2003), you need to be able to visualise the `Calendar` table as it is currently filtered. In the case of October 2003, there are 31 rows that remain unfiltered. The first (earliest) of these rows is October 1, 2003, and it has an ID of 823. The last unfiltered row is October 31, 2003, and it has an ID of 853. So "October this year" can be thought of as:

```
'Calendar'[ID] >=823 && 'Calendar'[ID] <=853
```

And October last year can be thought of as:

```
'Calendar'[ID] >=823 - 365 && 'Calendar'[ID] <=853 - 365
```

When you write it this way, it is obvious why you use >= `MIN` for the first filter line and <= `MAX` for the second one. And the really great thing is that this works regardless of the time period you are looking at. In this first example, you are looking at a month, but if you look at the #2 in the matrix above, this time the filter context is on an entire year. The formula therefore is filtering for all periods after the first date of the entire year (1/1/2003: `'Calendar'[ID] = 550`) and also for less than the last date of the calendar year (31/12/2003: `'Calendar'[ID] = 914`). Once you learn to trust this "whole of table" behaviour, you will be able to very quickly write custom time intelligence formulas by referencing the `ID` column alone.

What About Leap Years?

Astute readers will be crying foul about leap years by now. In fact, if you compare the measure `[Total Sales LY ID]` with the measure `[Total Sales LY]`, you will notice that there is a different answer for leap years. Well, as I said earlier, every business is different, and different businesses handle these things in different ways. It is beyond the scope of this book to provide solutions to this problem, but you can read about some possible approaches at http://www.daxpatterns.com/time-patterns/.

A Final Word on ID Columns

In the examples above, we have used an ID column on the day level of granularity—the same level of granularity as the `Calendar` table. I also like to load integer ID columns for the other important columns of data in a `Calendar` table. For example, I like to add a `MonthID` column to my `Calendar` tables. It starts with 1 for the first January in the calendar, 2 for the first February, . . . 12 for the first December. But then it would become 13 for the second January, 14 for the second February, etc. Having a `MonthID` column like this makes it easy to reach back in time and grab the same monthly period from any time in the past.

Practice Exercises: Time Intelligence, Cont.

It's time for some more practice. Write the following formulas. First set up an appropriate matrix so that you will get immediate feedback about whether your formula is correct. Find the solutions to these practice exercises in "Appendix A: Answers to Practice Exercises" on page 178.

68. [Total Sales Moving Annual Total]

With this DAX formula, you need to create a rolling 12-month total of sales. It will always show you 12 months' worth of sales, up to the end of the current month. Think about the problem using English words first and then convert that to DAX, using the techniques you have learnt here. I show my tips for writing this measure later in this chapter, but give it a go yourself first.

69. [Total Sales Rolling 90 Days]

This is the same as the formula for Practice Exercise 68, but instead of delivering a rolling 12-month total, you will instead deliver a rolling 90-day total. Try to do this one from scratch, without referencing Practice Exercise 68. This is good practice to help you think like the DAX engine.

Tips for Writing a Moving Annual Total

This section walks through how to create the formula in Practice Exercise 68. Start by setting up a new matrix with `Years` and `Months` on Rows and `[Total Sales]` on Values, as shown below.

CalendarYear	Total Sales
2001	**$3,266,374**
July	$473,388
August	$506,192
September	$473,943
October	$513,329
November	$543,993
December	$755,528
2002	**$6,530,344**
January	$596,747
February	$550,817
March	$644,135
April	$663,692
May	$673,556
June	$676,764
July	$500,365
August	$546,001
September	$350,467
Total	**$29,358,677**

Then write your formula:

```
Total Sales Moving Annual Total
    = CALCULATE([Total Sales],
        FILTER(ALL('Calendar'),
        'Calendar'[ID] > MAX('Calendar'[ID]) - 365
        && 'Calendar'[ID] <= MAX('Calendar'[ID])
    )
)
```

Note: This is not the only way to write this formula. Just as in Excel, there are often multiple ways to write a formula in Power BI. If you have something different and it works, that's great. Also note that this formula may not work with leap years, depending on how your business handles the extra day. (Some businesses ignore the extra day and actually have 6 × 364-day years followed by 1 × 371-day extraordinary year, so it depends.)

Now check your formulas against the matrix, as shown below. You can check the Moving Annual Total at the end of December 2002 (see #1 below) against the matrix calculated total (#2) to validate that the formula is working.

CalendarYear	Total Sales	Total Sales Moving Annual Total
2002	**$6,530,344**	**$6,530,344**
January	$596,747	$3,863,120
February	$550,817	$4,413,937
March	$644,135	$5,058,072
April	$663,692	$5,721,764
May	$673,556	$6,395,321
June	$676,764	$7,072,084
July	$500,365	$7,099,061
August	$546,001	$7,138,871
September	$350,467	$7,015,395
October	$415,390	$6,917,456
November	$335,095	$6,708,557
December	$577,314	$6,530,344
2003	**$9,791,060**	**$9,791,060**
January	$438,865	$6,372,462
February	$489,090	$6,310,736
Total	**$29,358,677**	**$9,744,506**

One thing to note is that the first `FILTER()` line in the formula says *greater than*, and the last `FILTER()` line says *less than or equal to*. It is easy to get these things wrong when writing formulas, but you should not worry about this because it is easy to check and verify. As long as you set up a matrix so that you can test the formulas you are writing, you can just take a guess and then change it if you need to (i.e., if you got it wrong). In this example, if you used *greater than or equal to*, you would end up with 366 days, which is incorrect.

But What About the First Year?

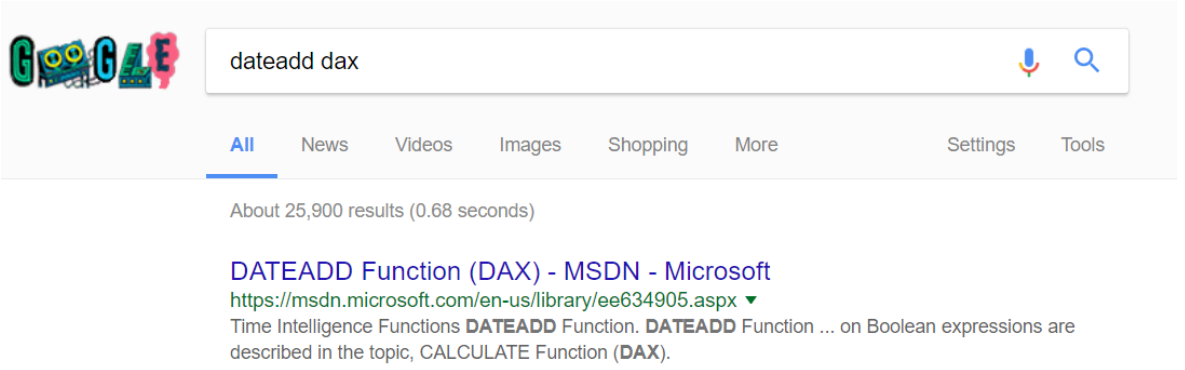
Now, if you want to get technical, the `[Total Sales Moving Annual Total]` result really doesn't make sense in the first 11 months of the sales data because you didn't have a full year of sales until the end of June 2002. There are many ways to solve this problem by using the `IF()` function. Here is one solution:

```
Total Sales MAT Improved = IF(MAX('Calendar'[ID])>=365,
    CALCULATE([Total Sales],
        FILTER(ALL('Calendar'),
        'Calendar'[ID] > MAX('Calendar'[ID]) - 365
        && 'Calendar'[ID] <= MAX('Calendar'[ID])
        )
    )
)
```

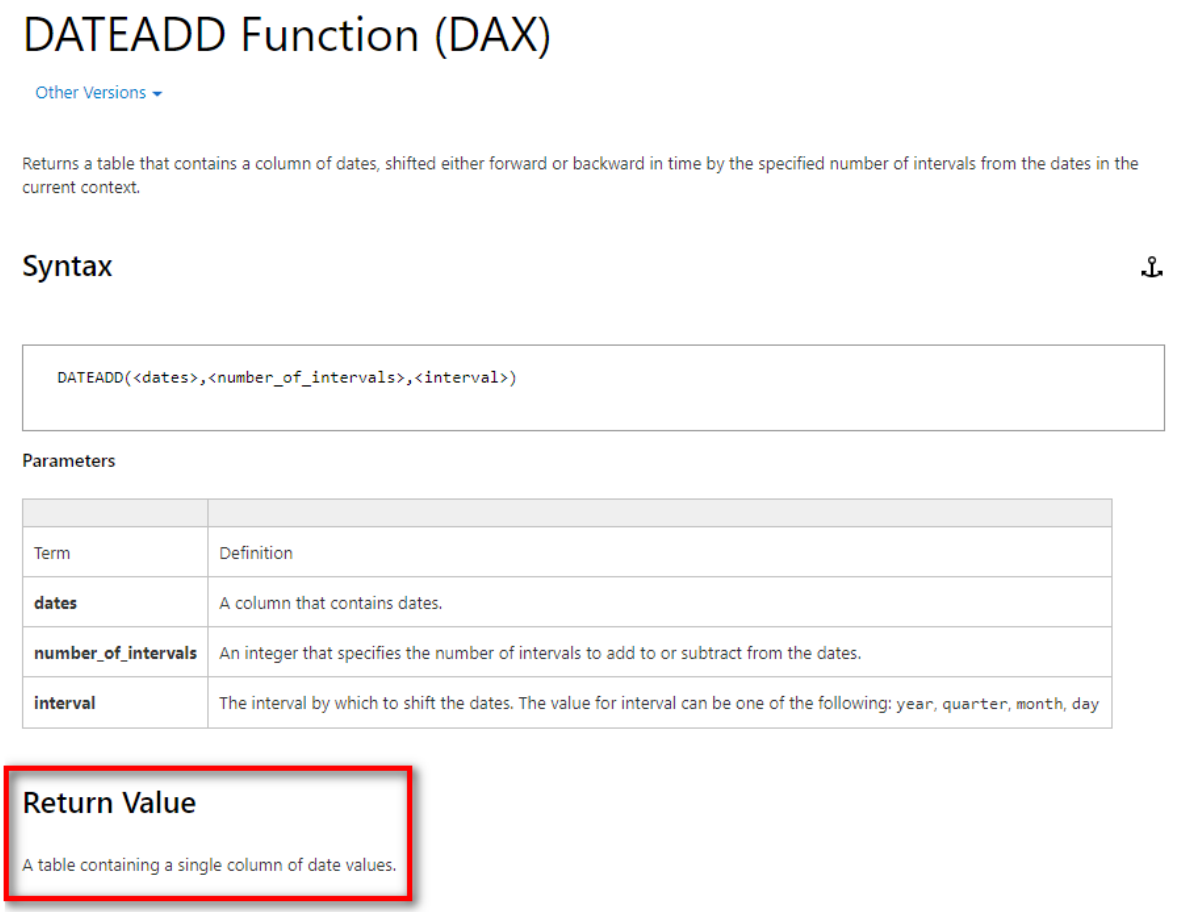
Tip: By now you may have realised that it is easiest to copy one formula and then edit the copied version for the new formula. Indeed, this is a good idea, but try to keep the copying and changing to a minimum while you are learning. It's a good idea to get as much DAX writing practice as you can. Once you know how to do it, using copy and paste is a great way to go faster.

Researching DAX Functions

There are a lot of other time intelligence functions that you can use to write time-based DAX formulas. A key piece of advice as you learn how to use these other time intelligence functions (indeed, all other DAX functions) is to do a quick online search and read the relevant information in the documentation. To do this, do a web search for the function name followed by the word *DAX*. In the example below, I have searched for "DATEADD DAX."



The first result returned is normally the official Microsoft documentation (MSDN) site. When you click on this MSDN link, you see something like the following.



DATEADD Function (DAX)

Other Versions

Returns a table that contains a column of dates, shifted either forward or backward in time by the specified number of intervals from the dates in the current context.

Syntax

```
DATEADD(<dates>,<number_of_intervals>,<interval>)
```

Parameters

Term	Definition
dates	A column that contains dates.
number_of_intervals	An integer that specifies the number of intervals to add to or subtract from the dates.
interval	The interval by which to shift the dates. The value for interval can be one of the following: `year`, `quarter`, `month`, `day`

Return Value

A table containing a single column of date values.

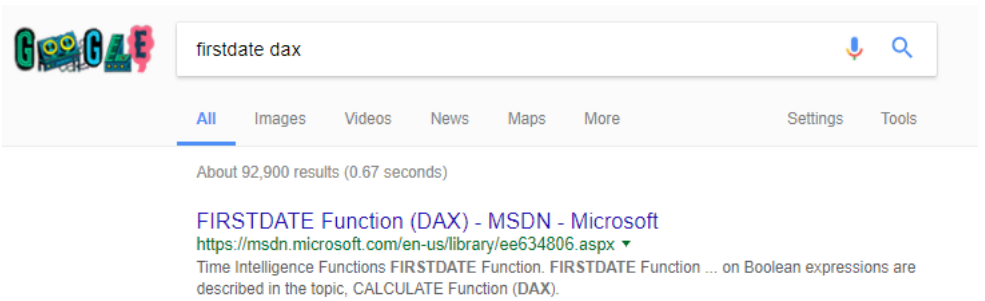
In many cases, the official documentation is not as useful as other websites. But there is some very important information that you can get from MSDN: the syntax, parameters, and return value. You can find the syntax and parameters by typing a formula directly into the formula bar, but sometimes the IntelliSense help doesn't clearly tell you the return value—and this is where doing a web search can help. The return value is a key piece of information that helps you understand how to use a function. In the case of `DATEADD()` above, the return value is *a table*, and hence you would use `DATEADD()` inside `CALCULATE()` to do a time shift. So you might write something like this:

```
Total Sales LY DATEADD = CALCULATE([Total Sales],
    DATEADD('Calendar'[Date],-1,Year)
)
```

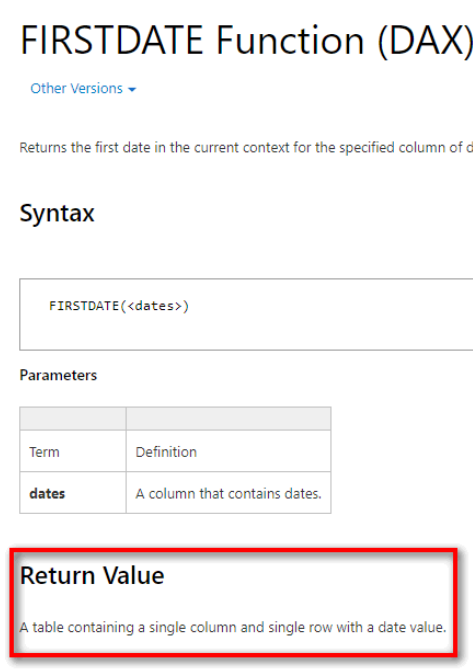
This formula works on various different time horizons, including quarters as well as years, as shown below. (You may realise that this is basically the same as the `SAMEPERIODLASTYEAR()` example shown earlier in this chapter.)

CalendarYear	Total Sales	Total Sales LY DATEADD
2001	**$3,266,374**	
3	$1,453,523	
4	$1,812,851	
2002	**$6,530,344**	**$3,266,374**
1	$1,791,698	
2	$2,014,012	
3	$1,396,834	$1,453,523
4	$1,327,799	$1,812,851
2003	**$9,791,060**	**$6,530,344**
1	$1,413,530	$1,791,698
2	$1,623,971	$2,014,012
3	$2,744,340	$1,396,834
4	$4,009,218	$1,327,799
2004	**$9,770,900**	**$9,791,060**
1	$4,283,630	$1,413,530
2	$5,436,429	$1,623,971
Total	**$29,358,677**	**$19,587,777**

As another example, when you do a quick search for `FIRSTDATE`, you find the MSDN site the first time again.



If you click through to the MSDN site, you can see that the returned value is a special table that has a single column and a single row, as shown below.



FIRSTDATE Function (DAX)

Other Versions

Returns the first date in the current context for the specified column of dates.

Syntax

```
FIRSTDATE(<dates>)
```

Parameters

Term	Definition
dates	A column that contains dates.

Return Value

A table containing a single column and single row with a date value.

`FIRSTDATE()` returns a single value in a table. This is a special type of table that can be placed directly into a cell in a matrix. (Normally you cannot do this.) So you could write a formula like this:

```
First Date = FIRSTDATE('Calendar'[Date])
```

CalendarYear	Total Sales	Total Sales LY DATEADD	First Date
2001	**$3,266,374**		**1/07/2001**
3	$1,453,523		1/07/2001
4	$1,812,851		1/10/2001
2002	**$6,530,344**	**$3,266,374**	**1/01/2002**
1	$1,791,698		1/01/2002
2	$2,014,012		1/04/2002
3	$1,396,834	$1,453,523	1/07/2002
4	$1,327,799	$1,812,851	1/10/2002
2003	**$9,791,060**	**$6,530,344**	**1/01/2003**
1	$1,413,530	$1,791,698	1/01/2003
2	$1,623,971	$2,014,012	1/04/2003
3	$2,744,340	$1,396,834	1/07/2003
4	$4,009,218	$1,327,799	1/10/2003
2004	**$9,770,900**	**$9,791,060**	**1/01/2004**
1	$4,283,630	$1,413,530	1/01/2004
2	$5,436,429	$1,623,971	1/04/2004
Total	**$29,358,677**	**$19,587,777**	**1/07/2001**

Other Time Intelligence Functions

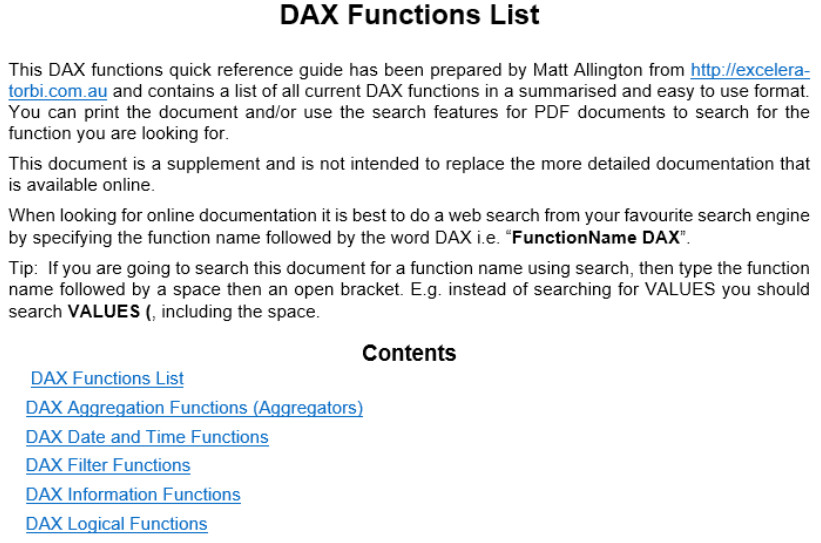
Here is a list of other time intelligence functions that you might want to explore:

```
DATESINPERIOD(date_column, start_date, number_of_intervals, intervals)
DATESBETWEEN(column, start_date, end_date)
DATEADD(date_column, number_of_intervals, interval)
FIRSTDATE (datecolumn)
LASTDATE (datecolumn)
LASTNONBLANKDATE (datecolumn, [expression])
STARTOFMONTH (date_column)
STARTOFQUARTER (date_column)
STARTOFYEAR(date_column [,YE_date])
ENDOFMONTH(date_column)
ENDOFQUARTER(date_column)
ENDOFYEAR(date_column)
PARALLELPERIOD(date_column)
PREVIOUSDAY(date_column)
PREVIOUSMONTH(date_column)
PREVIOUSQUARTER(date_column)
PREVIOUSYEAR(date_column)
NEXTDAY(date_column)
NEXTMONTH(date_column)
NEXTQUARTER (date_column)
NEXTYEAR(date_column [,YE_date])
DATESMTD(date_column)
DATESQTD (date_column)
DATESYTD (date_column [,YE_date])
TOTALMTD(expression, dates, filter)
TOTALQTD(expression, dates, filter)
```

A Free Quick Reference Guide

I have produced (and I maintain for new functions) a quick reference guide of all DAX functions in PDF format that you may like to download and use. The *DAX Reference Guide* PDF is not meant to replace the online documentation but to supplement it. As shown below, the PDF is fully indexed, and you can jump to the relevant sections by clicking on the hyperlinks in the table of contents.

You can download the *DAX Reference Guide* for free by visiting my online shop at http://xbi.com.au/shop and then navigating to the Books section.



DAX Functions List

This DAX functions quick reference guide has been prepared by Matt Allington from http://excelera-torbi.com.au and contains a list of all current DAX functions in a summarised and easy to use format. You can print the document and/or use the search features for PDF documents to search for the function you are looking for.

This document is a supplement and is not intended to replace the more detailed documentation that is available online.

When looking for online documentation it is best to do a web search from your favourite search engine by specifying the function name followed by the word DAX i.e. "**FunctionName DAX**".

Tip: If you are going to search this document for a function name using search, then type the function name followed by a space then an open bracket. E.g. instead of searching for VALUES you should search **VALUES (**, including the space.

Contents

DAX Functions List
DAX Aggregation Functions (Aggregators)
DAX Date and Time Functions
DAX Filter Functions
DAX Information Functions
DAX Logical Functions

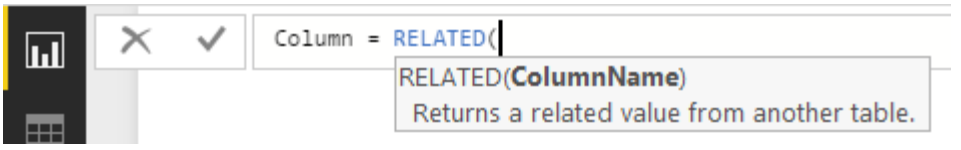
16: DAX Topic: RELATED() and RELATEDTABLE()

The functions `RELATED()` and `RELATEDTABLE()` are typically used in calculated columns to reference relevant records in other tables, although they can be used in measures, too. They are a bit like `VLOOKUP()` for tables that have a relationship. As mentioned briefly in Chapter 10, a row context does not follow a relationship. So even though there may be a relationship between two tables, a row context cannot use this relationship—unless you use one of these two functions that can. Basically, `RELATED()` and `RELATEDTABLE()` allow a row context to leverage an existing relationship so it can access columns in related tables.

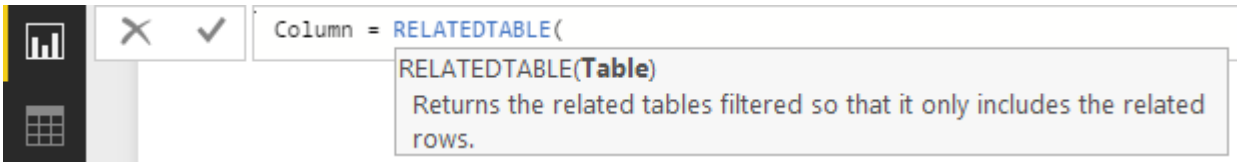
When to Use RELATED() vs. RELATEDTABLE()

To understand when to use the `RELATED()` and `RELATEDTABLE()` functions, you need to understand what each one returns. As you know, you can use IntelliSense in the formula bar to find out what each of these functions returns.

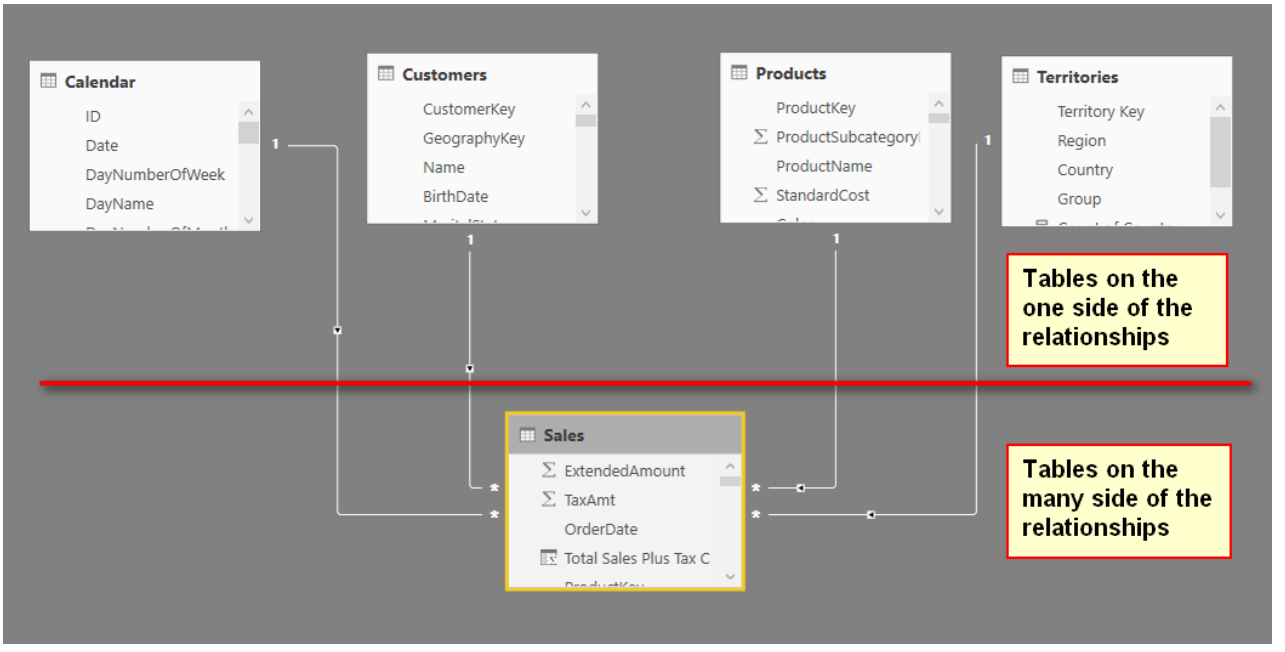
You can see below that `RELATED()` returns a single value from another table.



As shown below, `RELATEDTABLE()` returns a table.



Remember from Chapter 2 that relationships between tables in Power BI are normally of the type one-to-many. Also remember from Chapter 2 that best practice (especially for people coming from an Excel background) is to lay out tables in Relationships view with the lookup tables at the top (the "one" side of the relationship) and the data tables at the bottom (the "many" side of the relationship), as shown below.



The two `RELATED` functions allow you to refer to columns in another connected table. So when you think about it, if you want to add a custom column in a table on the "one" side of the relationship—i.e., add a new column in a lookup table (a table above the line in the image above)—then it is highly likely that there will be multiple rows on the "many" side of the relationship. So when writing a formula in a calculated column on a lookup table, you must use the `RELATEDTABLE()` function *because it will fetch a table of values*, including

all the matching values in the data table. Conversely, if you are writing a calculated column in a table on the "many" side of the relationship (i.e., a data table), then there will be only one matching row in the lookup table, and hence you use `RELATED()` to return that single value.

The RELATED() Function

This section provides an example of bringing a value from a column in a lookup table into a table on the "many" side of the relationship. For the sake of this example, assume that your business has a new management layer, and you want to add a new level of reporting to cover this new management layer. In effect, you need to enhance the `Territories` table to add a new geographic region. To achieve this, you could do the following:

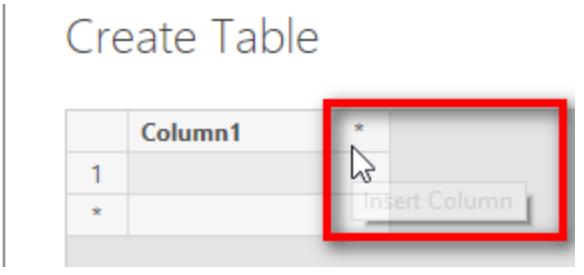
1. Create a new table that contains the logic of the new management layer.
2. Import the new table into the data model.
3. Join the new table to the existing Territories table (in this example).
4. Create a new calculated column in the Territories table (on the "many" side of the relationship) and bring in the new management layer from the new table into the Territories table as a new column.

This will all make more sense as you work through the following example, which also shows how you can manually add new tables of data in Power BI.

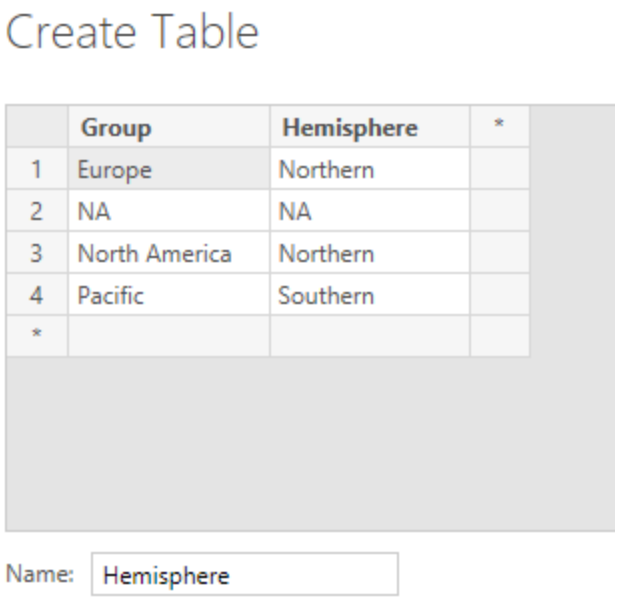
Here's How: Manually Adding Data to Power BI

This example shows how to add data directly into Power BI without having to use another tool like Excel:

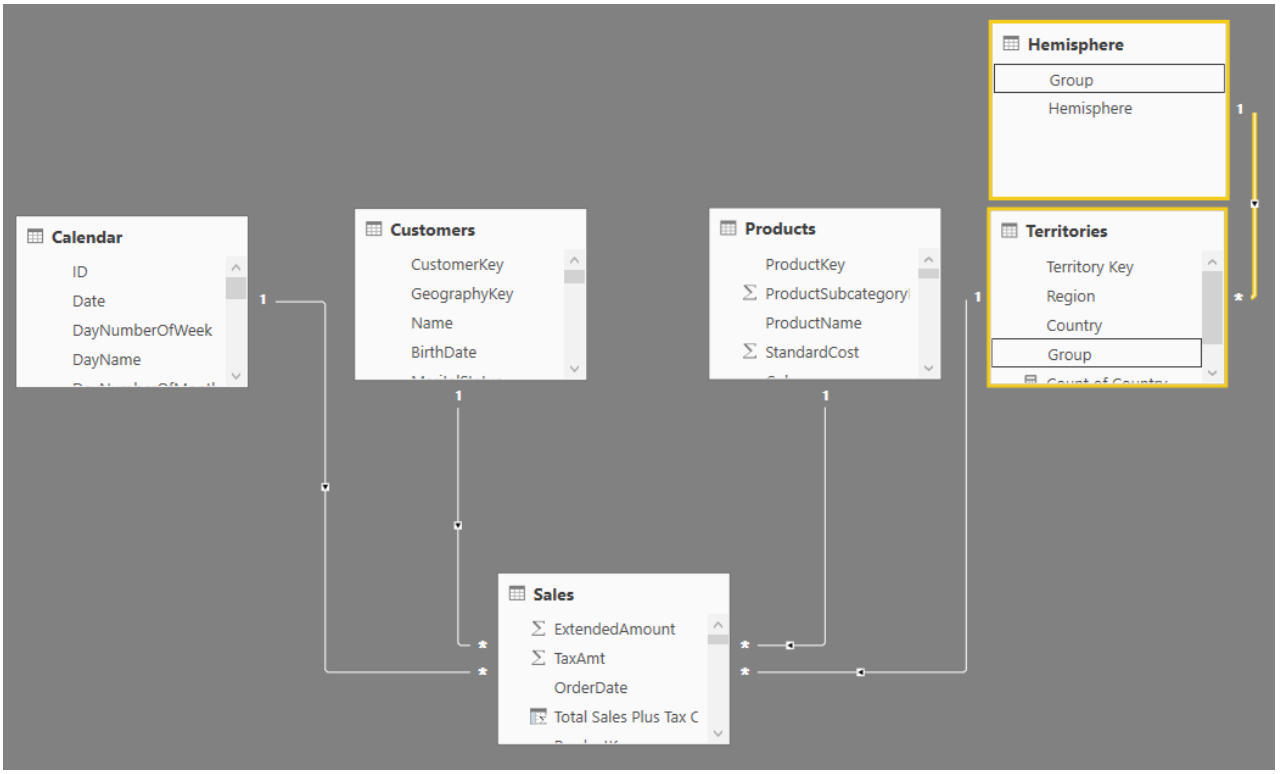
1. On the Home tab in Power BI Desktop, click Enter Data.
2. As shown below, click the * to add a new column.



3. Change the column names (double-click them and type a new name) and then enter the data as shown below. Rename the table `Hemisphere` and then click Load.



4. Switch to Relationships view and rearrange the new tables so that the new table is sitting above the current Territories table, as shown below. This new lookup table is a lookup table to another lookup table and will be on the "one" side of the new relationship. Power BI should automatically join the tables.



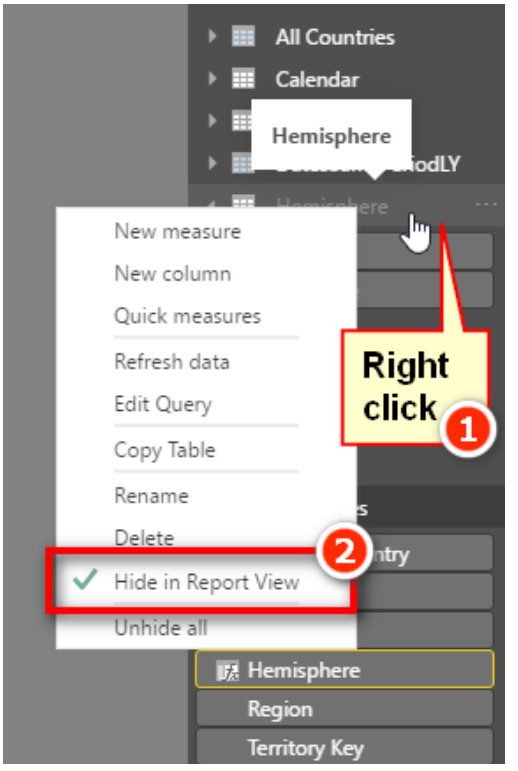
Note: The Territories table now has two roles. It is now acting as a lookup table to the Sales table and as a data table to the new Hemisphere table.

5. Bring the data that resides in the Hemisphere[Hemisphere] column into a new calculated column inside the Territories table. Switch to Data view, right-click on the Territories table, select New Column, and then type in the formula shown below. After you press Enter, you see all the values appear in the new calculated column. It is a lot like VLOOKUP()!

```
Hemisphere = RELATED(Hemisphere[Hemisphere])
```

Territory Key	Region	Country	Group	Hemisphere
1	Northwest	United States	North America	Northern
2	Northeast	United States	North America	Northern
3	Central	United States	North America	Northern
4	Southwest	United States	North America	Northern
5	Southeast	United States	North America	Northern
6	Canada	Canada	North America	Northern
7	France	France	Europe	Northern
8	Germany	Germany	Europe	Northern
9	Australia	Australia	Pacific	Southern
10	United Kingdom	United Kingdom	Europe	Northern
11	NA	NA	NA	NA

6. Finally, to hide the Hemisphere table from the client tools, in Data view, right-click the table (see #1 below) and select Hide in Report View (#2).



It is good practice to bring data from an add-on table like this `Hemisphere` table into the main `Territories` table as an additional column rather than use the data in an additional lookup table. It is possible to leave the `Territories` table untouched and use the columns from the `Hemisphere` table in your matrix. But the problem is that this can be confusing to users. It doesn't make business sense to have all the geographic information in the `Territories` table *except* for the hemisphere information, which is in the `Hemisphere` table. So for consistency and simplicity for the end user, it is better to bring all the "like data" into the same table.

> **Note:** Better practice is to do this when you load the data using Power Query, but you can also do it as shown here. And best practice is to change the `Territories` table back at the source to include the new `Hemisphere` column in the Territories table, but that is not always possible in a timely manner.

The RELATEDTABLE() Function

As discussed earlier, `RELATEDTABLE()` is used to reference a table on the "many" side of the relationship. A simple example is to add a new calculated column to count how many sales there have been for each product. Once again, I generally don't recommend that you do this (because you can do it in a measure), but there may be valid reasons to do it in some cases.

Go ahead now and add the following calculated column in the `Products` table:

```
= COUNTROWS(RELATEDTABLE(Sales))
```

As you know, `RELATEDTABLE()` returns a table, and `COUNTROWS()` counts the rows in that table. This calculated column in the `Products` table therefore takes the row context in the calculated column and leverages the relationship with the `Sales` table to count the rows in the `Sales` table for just the single product. As a result, you end up with a new column that indicates the number of items sold (over all time) for each product in the `Products` table. (The quantity for each line in the `Sales` table is always 1 in this sample data.)

> **Note:** You do not need to use CALCULATE() with RELATEDTABLE() to force context transition and convert the row context to a filter context. RELATEDTABLE() will work on its own.

One valid use case for using `RELATEDTABLE()` would be if you want to create a slicer to filter on slow-, moderate-, and fast-selling products. If you want to use a slicer, you must write your DAX as a calculated column. (You can't place measures in slicers.) You could first create a calculated column and then use the banding technique discussed in the next chapter to group products into slow-, moderate-, and fast-moving products. (Park this thought for now and come back after you have read Chapter 17 if you want to try out this technique.)

17: Concept: Disconnected Tables

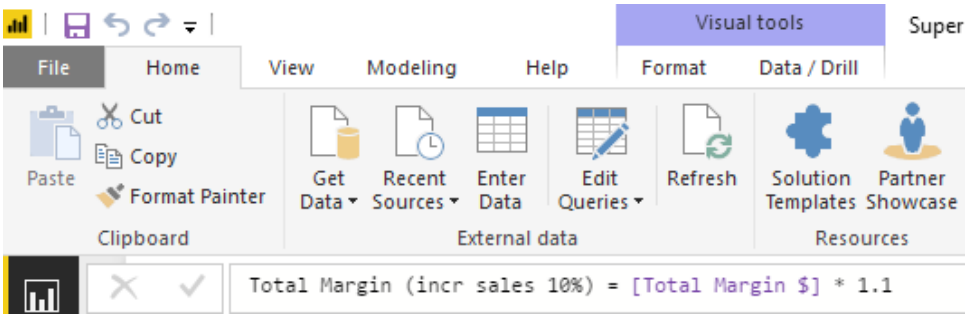
So far as you have worked through this book, you have always loaded tables into the data model and then connected them to other tables. This is a fundamental technique with Power BI that allows you to work across multiple tables without using `VLOOKUP()`. However, you are not required to join tables together in the data model, and indeed there are some instances when it doesn't make sense to do so. This chapter discusses two techniques that do not involve connecting tables:

- Using What-If analysis
- Using banding

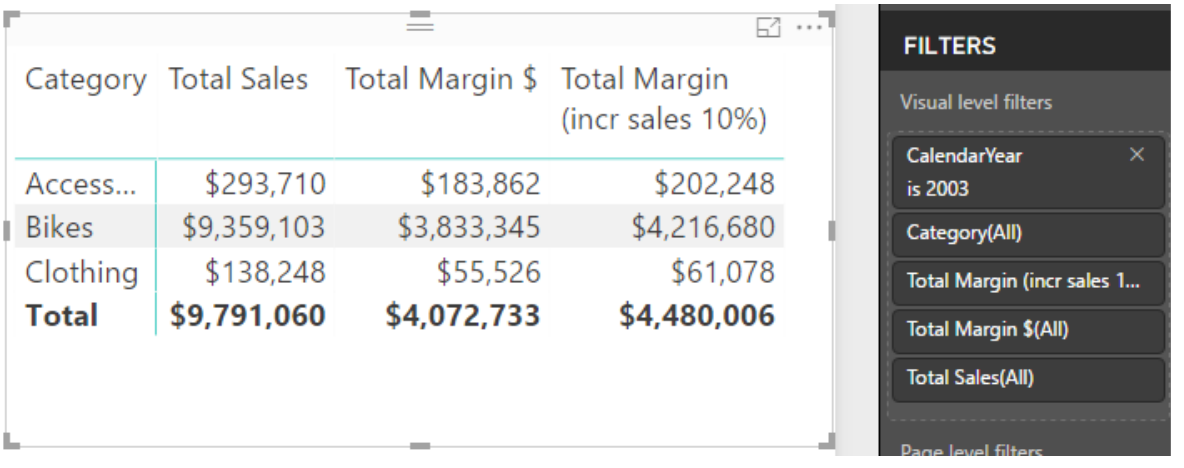
Using What-If Analysis

I first learnt to do What-If analysis by manually creating a table of values and then writing a special measure that would "harvest" the value selected by the user so it could be used inside a formula. In Aug 2017 Microsoft released a new feature called What-If that replaces the need to complete this process manually. With this new feature, it is easier than ever to do your own What-If analysis. Let's look at an example to demonstrate how to use the What-If capability.

Imagine that in your data, the sales result is directly proportional to the profit result. You have sales data and want to see what impact an increase in sales will have on your total profit. You could write a new measure hard-coded at a 10% increase, as shown below.



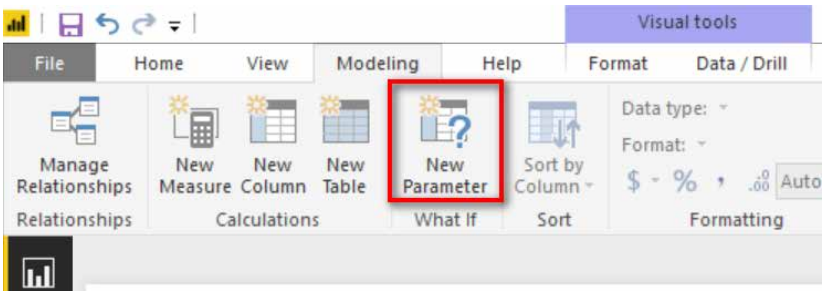
And it would look as shown below in a matrix.



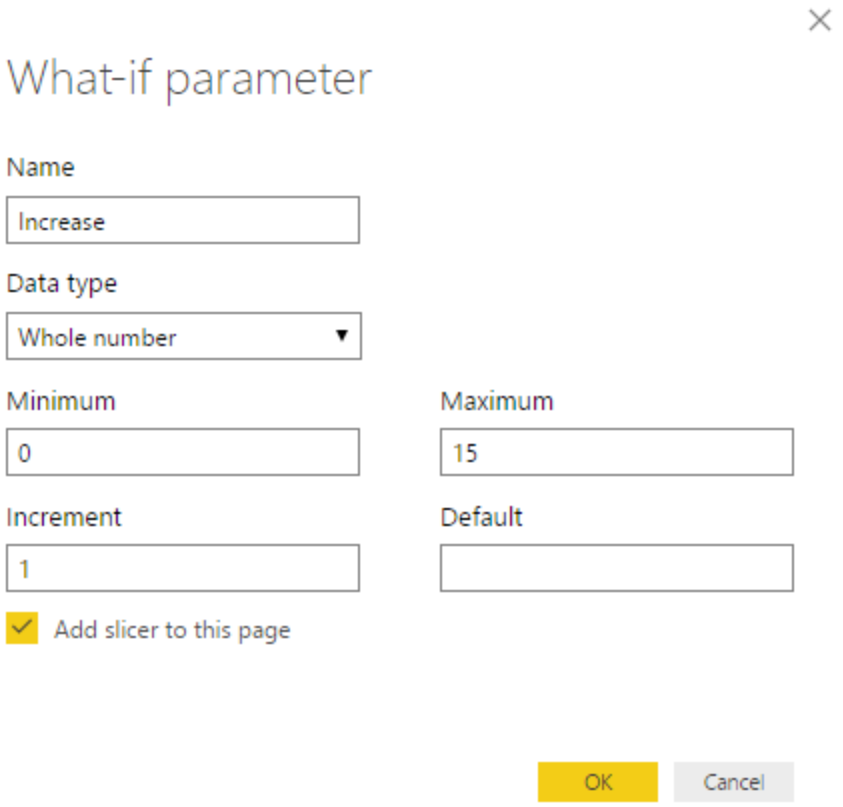
But what if you wanted to see what it looks like for a 5% increase in sales, or 15%, or some other percentage? It would not be efficient to create lots of new measures, one for each value. A better approach is to use the What-If analysis capability provided by Power BI.

Here's How: Using What-If

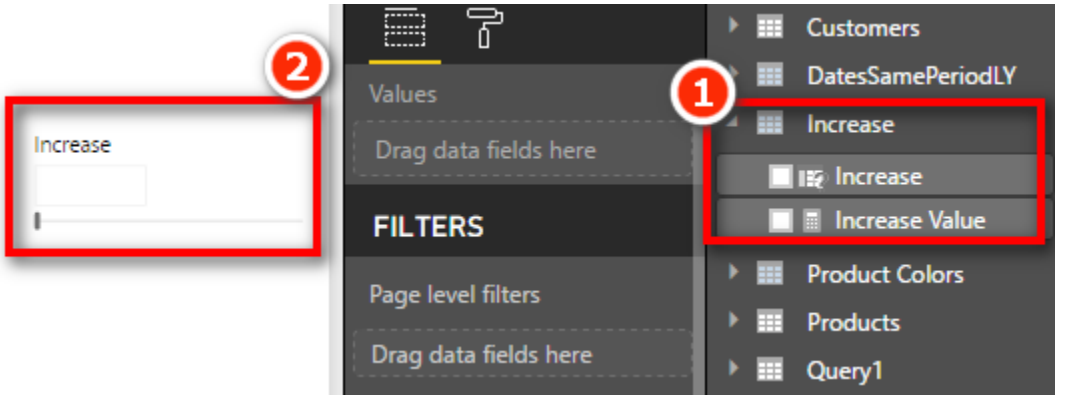
1. Create a new blank page in your Power BI workbook.
2. Navigate to the Modeling tab and click on New Parameter as shown below.



3. Enter a minimum and maximum value that will be used inside the What-If analysis, along with the increment. As you can see below, I have entered a range of integers from 0 to 15. I have also given my parameter a name "Increase".



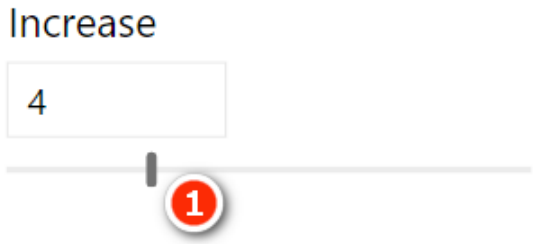
4. Click OK.
5. Notice that there is a new Table (including a new measure `[Increase Value]` in Fields list (#1) and also a new Slicer (#2) on the Report.



6. Write the following new measure that utilises the `[Increase Value]` measure shown above.

```
Total Margin with Selected Increase =
    [Total Margin $] * (100 + [Increase Value])/100
```

7. Add the above measure in the matrix from before, as shown below. Note I still have a visual level filter on CalendarYear = 2003. You can now use the Slicer (#1) to vary the sales increase and see what impact that increase has on `[Total Margin with Selected Increase]`.



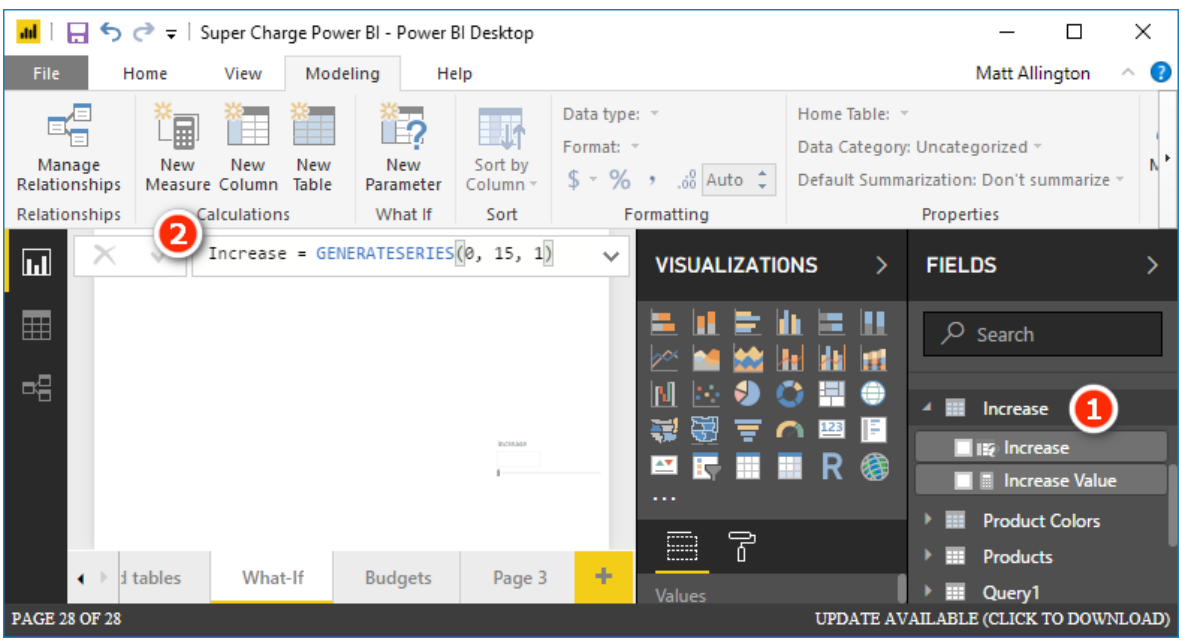
Category	Total Sales	Total Margin $	Total Margin with Selected Increase
Accessories	$293,710	$183,862	$191,216
Bikes	$9,359,103	$3,833,345	$3,986,679
Clothing	$138,248	$55,526	$57,747
Total	**$9,791,060**	**$4,072,733**	**$4,235,642**

How It Works

When you click the New Parameter button in Power BI, Power BI automatically creates three things. A new table of values, a new measure and a new slicer (optional but created by default).

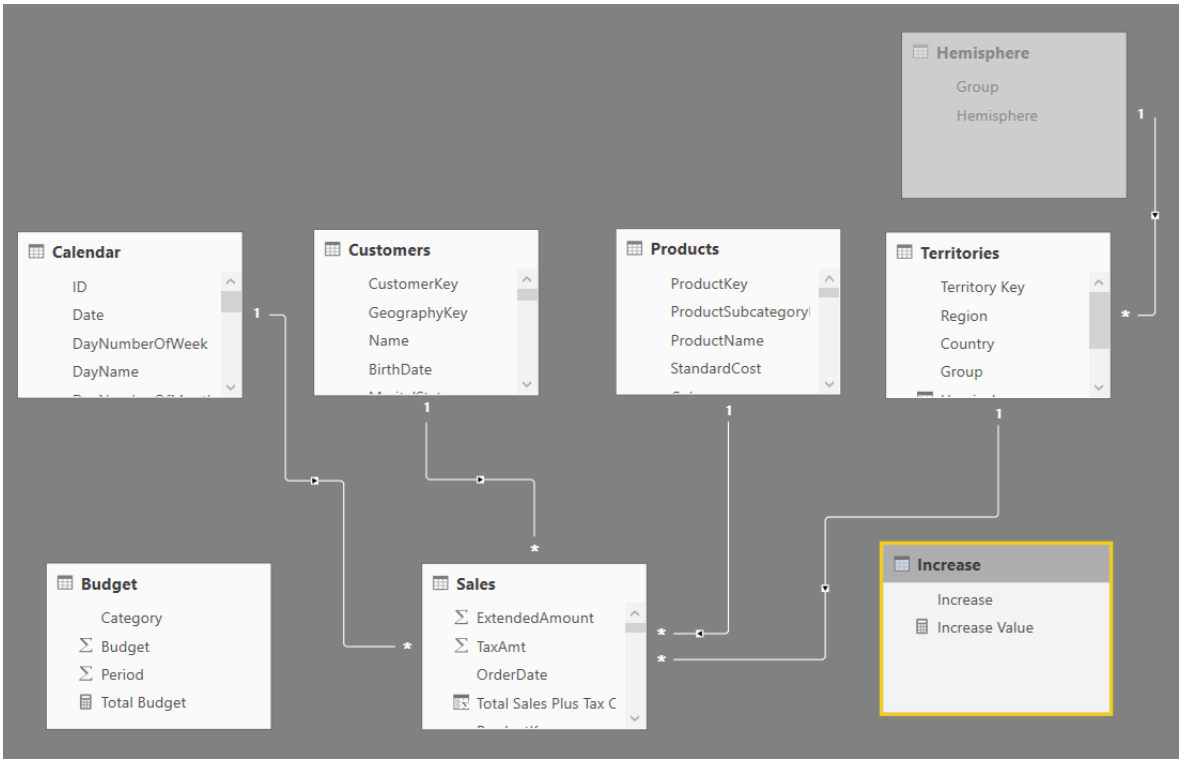
The New Table

Select the new table created by the New Parameter button in the Fields list on the right (#1 below). You will see that this new `Increase` table is a calculated table that uses the formula shown as #2 below.



It is pretty easy to work out the syntax of the function `GENERATESERIES()` just by looking at the formula. `GENERATESERIES()` is a new function in Power BI desktop that you can use anytime you want to create a new table of values with a constant increment between values.

If you switch to Relationships view, you will see the new table. (You may need to select Fit to Screen to be able to see it.) This time the table is not joined to any other table – it is a disconnected table. Just position it somewhere so it is easy to see on the screen, as shown below.



The New Slicer

Switch back to Report view. The new slicer that was automatically created uses the only column in the table `Increase[Increase]` as the input field. The slicer then allows the user to select a single value to represent the increase in sales required for analysis.

The New Measure

If you now click on the `[Increase Value]` measure, you will see that the measure formula is as follow:

```
Increase Value = SELECTEDVALUE(Increase[Increase])
```

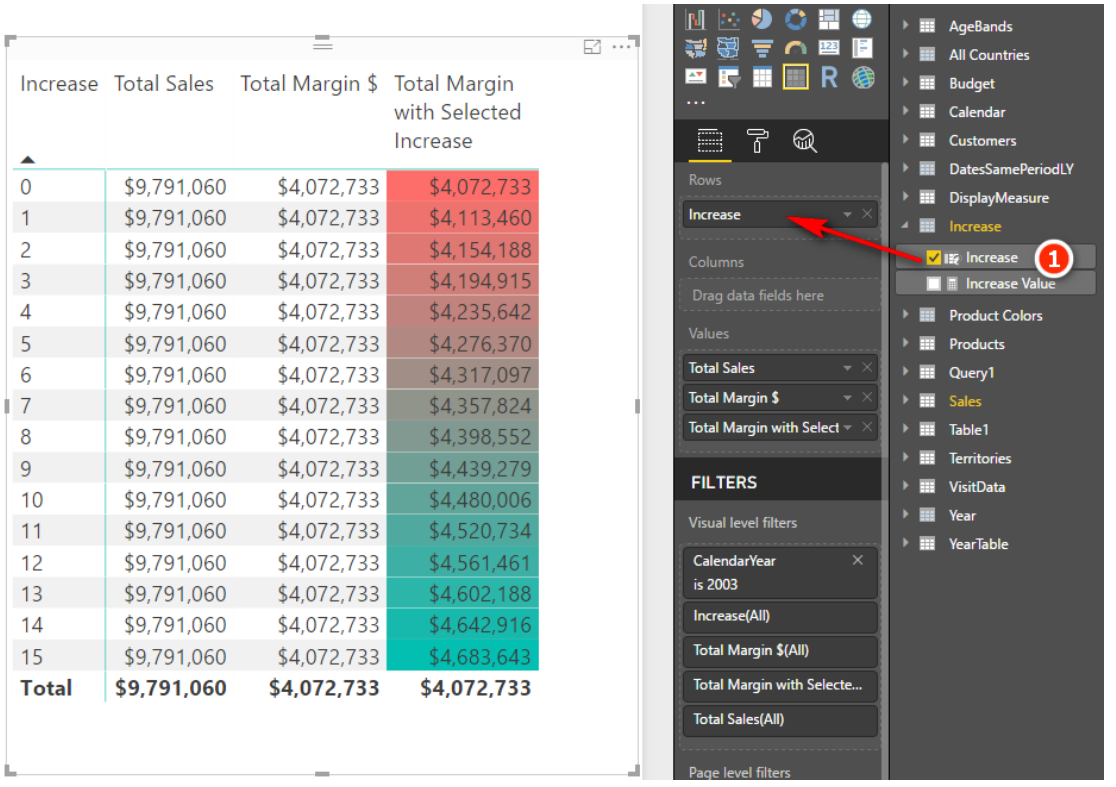
This measure uses the new function `SELECTEDVALUE()`. You may recall from Chapter 12 that this function returns the single value selected in the filter context, or blank if there is more than 1 value selected. This is the secret sauce of this What-If parameter. The `SELECTEDVALUE()` function "harvests" the selection from the user in the slicer and then passes that selected value to your formula. By default, if the user hasn't selected a single value, it returns `BLANK()`. (However, this can be changed via an optional final parameter.)

> **Note:** The behaviour of slicers tends to change as Microsoft is improving and developing Power BI. At this writing, there are a number of different configuration choices for slicers. Click the drop-down arrow in the top-right corner of the slicer to change the way the slicer is displayed.

Seeing All What-If Variants at Once

As well as using the slicer to see one of the What-If numbers at a time, it is also possible to use the `Increase[Increase]` Column on Rows in a matrix to see all the single values at once. I have modified the matrix from earlier as shown below. I have removed `Product[Category]` from Rows and added `Increase[Increase]` to Rows in its place (#1 below). I also cleared the selection from the slicer on the

page to see the full list of values. Lastly, I added some conditional formatting to make the changes in Margin measure more obvious.



Increase	Total Sales	Total Margin $	Total Margin with Selected Increase
0	$9,791,060	$4,072,733	$4,072,733
1	$9,791,060	$4,072,733	$4,113,460
2	$9,791,060	$4,072,733	$4,154,188
3	$9,791,060	$4,072,733	$4,194,915
4	$9,791,060	$4,072,733	$4,235,642
5	$9,791,060	$4,072,733	$4,276,370
6	$9,791,060	$4,072,733	$4,317,097
7	$9,791,060	$4,072,733	$4,357,824
8	$9,791,060	$4,072,733	$4,398,552
9	$9,791,060	$4,072,733	$4,439,279
10	$9,791,060	$4,072,733	$4,480,006
11	$9,791,060	$4,072,733	$4,520,734
12	$9,791,060	$4,072,733	$4,561,461
13	$9,791,060	$4,072,733	$4,602,188
14	$9,791,060	$4,072,733	$4,642,916
15	$9,791,060	$4,072,733	$4,683,643
Total	**$9,791,060**	**$4,072,733**	**$4,072,733**

Practice Exercise: Harvester Measures

In Chapter 9, you created the following DAX formula:

```
Total Customers Born Before 1950 =
    CALCULATE([Total number of Customers],
    Customers[BirthDate] <DATE(1950,1,1))
```

In the next practice exercise, you will write a measure that allows you to change the “before year” using the What-If feature.

Write the following new DAX formula. Find the solution to this practice exercise in "Appendix A: Answers to Practice Exercises" on page 178.

70. [Total Customers Born Before Selected Year]

Using the technique described above, create a new matrix (on a new page) that allows the user to select from a list of years in a slicer using a new What-If parameter. Change the above measure from being hard coded to 1950 and instead make the year selectable from the slicer.

This is quite a difficult problem and you will have to think back on what you have learnt in previous chapters to make it work. You should try to do it yourself, and if you get stuck, read the start of the worked through solution below, then try to solve the problem again.

Here’s How: Solving Practice Exercise 70

There is a trick to this practice exercise. The original measure you created used a “simple filter” in `CALCULATE()`. If you replace the “year value” from the first formula with the What-If measure `[Year Value]`, you get the error message shown below.

```
Total Customers Born Before Selected Year Error =
CALCULATE ([Total number of Customers],Customers[BirthDate] < DATE ( [Year Value], 1, 1 ))
```

A function 'CALCULATE' has been used in a True/False expression that is used as a table filter expression. This is not allowed.

The problem is that you cannot use measures in a "simple" `CALCULATE()` formula. If you want to use measures (as you do in this case), you must use the `FILTER()` function inside `CALCULATE()`. So instead of writing this:

```
Customers[BirthDate] < DATE ( [Year Value], 1, 1 )
```

you need to write a `FILTER()` function that filters the `Customers` table to replace the line above.

Go back and give it a go: See if you can write the correct formula by using the `FILTER()` function. If you still need more help, read on to see the correct formula.

Here is the worked-through solution for Practice Exercise 70:

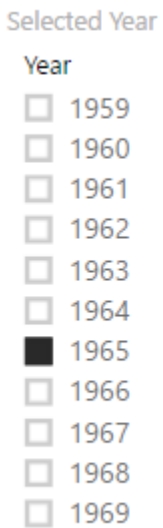
1. Create a new What-If parameter using values (say 1900 through 2000). Give it a name such as `Year`. Note the new measure created called `[Year Value]`.
2. Write the following measure and then add it to your matrix:

```
Total Customers Born Before Selected Year
    = CALCULATE (
    [Total number of Customers],
    FILTER (
        Customers,
        Customers[BirthDate] < DATE ( [Year Value], 1, 1 )
    )
)
```

You should end up with something that looks as shown below.

Note: I have changed the slicer layout do be a list of values. You can change a slicer from the drop-down arrow in the top-right corner of the slicer.

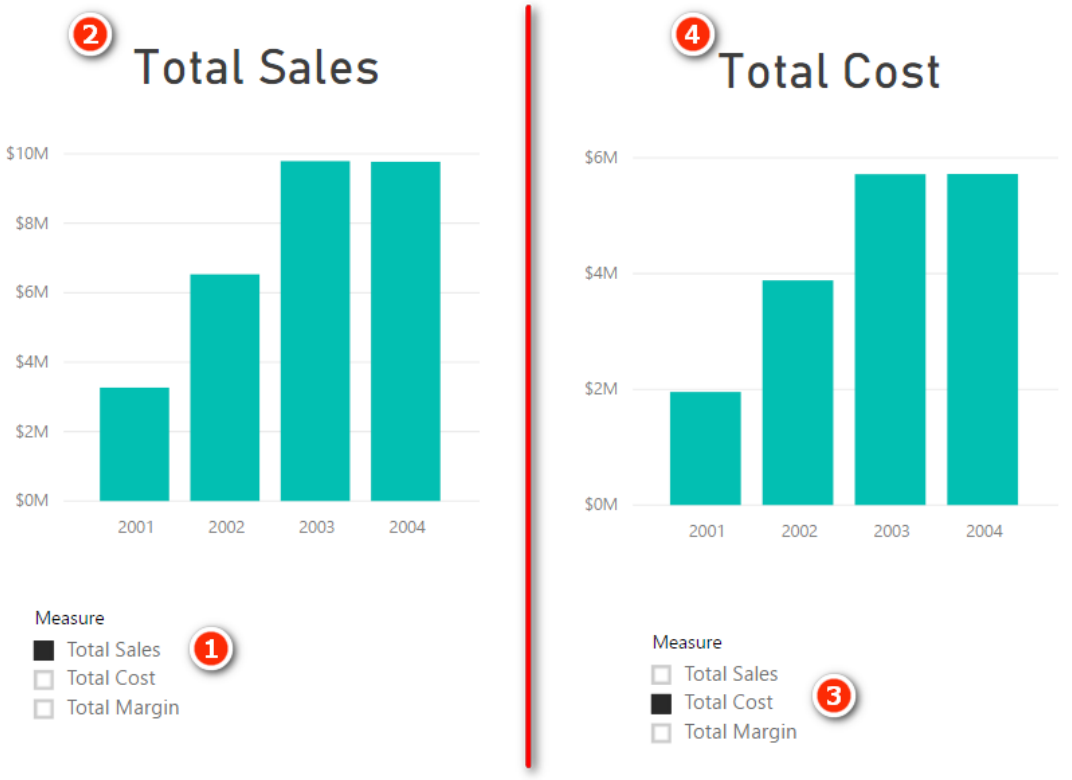
When you click on a year in the slicer, the `[Year Value]` measure updates, and the results for `[Total Customers Born before Selected Year]` update to show the value for year you have selected in the slicer.



Occupation	Total Customers Born Before Selected Year
Clerical	1,567
Management	2,229
Manual	860
Professional	3,382
Skilled Manual	2,071
Total	**10,109**

The SWITCH() Function Revisited

In Chapter 11, I introduced you to the `SWITCH()` function. One really cool feature of `SWITCH()` is that you can create a switch measure that allows you to toggle between multiple other measures. Take a look at the image below.



The image on the left has Total Sales selected in the slicer (#1). When Total Sales is selected, the chart updates to show Total Sales (#2). When the user selects a different value in the slicer such as Total Cost (#3), the chart changes to show Total Cost (#4). This toggle effect is really engaging for the report user and can be used to create very complex and useful interactive reports.

Here's How: Creating a Morphing Switch Measure

You need to create a disconnected table and a harvester measure to be able to complete this technique:

1. On the Home menu, click Enter Data.
2. Enter 3 rows of data as shown below. Call the table `DisplayMeasure`, then click Load.

Create Table

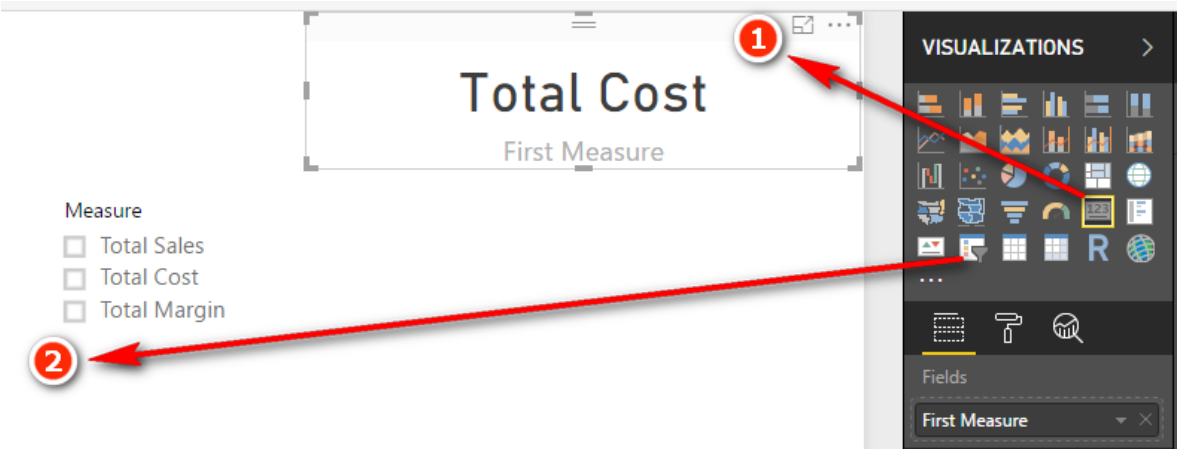
	Measure ID	Measure	*
1	1	Total Sales	
2	2	Total Cost	
3	3	Total Margin	
*			

3. Navigate to Data view, click the new `DisplayMeasure` table, go to the Modeling menu and change the sort order of the `Measure` column so it sorts by `Measure ID` column instead. Do you remember how? You did it in Chapter 12.
4. Right click on the DisplayMeasure table and add a new measure as follows.

```
Selected Measure = SELECTEDVALUE(DisplayMeasure[Measure ID])
```

5. This measure is called a Harvester Measure and it uses the same technique used with What-If earlier in the chapter. It checks to see if there is a single value selected for `DisplayMeasure[Measure ID]` and if so it returns that value, otherwise it returns `BLANK`. It "harvests" the selection from the user when used with a slicer.

6. Go back to Report view and create a new page. Place a Card (#1) on the report and add the `DisplayMeasure[Measure]` column to the Card Fields. Then place a Slicer (#2) on the report and place the `DisplayMeasure[Measure]` on the Slicer as well.



7. Now when you click on the slicer, the Card will update to show you which measure you have selected. This Card will be the title for the chart that will be added next.
8. Click on the Card, go to the Format pane, and turn off Category Label.
9. Right click on the Sales table and write the following measure.

```
Measure to Display = SWITCH([Selected Value],
1,[Total Sales],
2,[Total Cost],
3,[Total Margin $]
)
```

10. Add a new Column Chart to the report (clustered or stacked column chart, both will work). Place `Calendar[Year]` on the Axis and the new `[Measure to Display]` measure as the Value as shown below.

11. Go to the Format pane for the chart and turn off the default Title. Also expand the X-Axis section and change the Axis type to Categorical.

When done, you should have an interactive chart that will display the measure selected by the user in the slicer.

Using Banding

Another disconnected table technique is banding; I learnt this technique from Marco Russo and Alberto Ferrari at http://sqlbi.com.

To understand banding, think about the earlier example, in which you created a slicer based on the year the customer was born. A more common and practical need is to be able to analyse customers based on their age group rather than their actual age, like this:

- Under 20
- 20 but less than 30
- 30 to less than 40
- 40 to less than 50
- 50 to less than 60
- 60 and over

It is possible to write a calculated column in the `Customers` table that creates these age group bands. But it would be a very complex formula, and it would be hard to edit.

> **Note:** For the sake of the exercise, I use January 1, 2003, as the "current date" from which to work out the age of each customer. Of course, in reality, each customer's age band will change over time, but I have ignored that fact for this example so that the results you see onscreen will be the same as my results shown below. If I used `TODAY()` in this exercise, my results would be different to yours.

A hard-coded calculated column formula for age group might look like this:

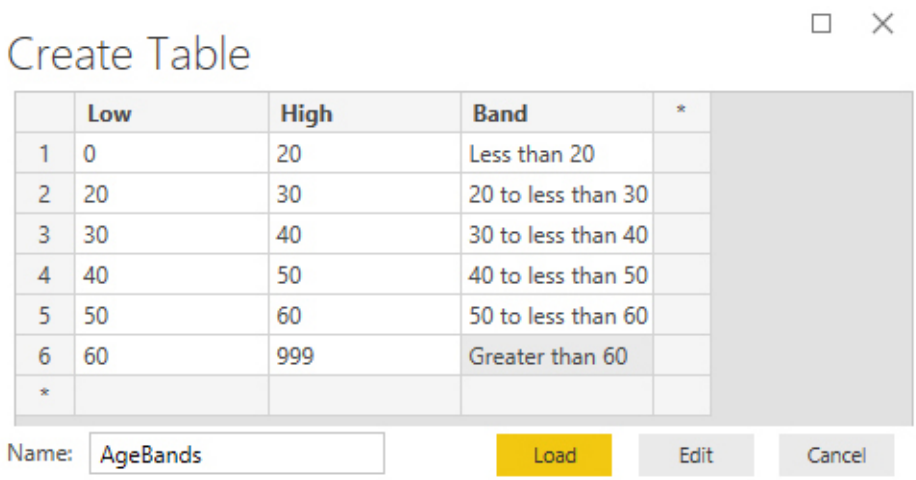
```
=IF(((date(2003,1,1) - Customers[BirthDate])/365)<20,"Less than 20",IF
(((date(2003,1,1) - Customers[BirthDate])/365)<30,"20 to less than
30",IF(((date(2003,1,1) - Customers[BirthDate])/365)<40,"30 to less
than 40",IF(((date(2003,1,1) - Customers[BirthDate])/365)<50,"40 to
less than 50",IF(((date(2003,1,1) - Customers[BirthDate])/365)<60,"50
to less than 60","Greater than 60")))))
```

This DAX works, but it is not very user friendly, it is hard to write, and it is even harder to read and maintain. A better approach is to use banding.

Here's How: Applying Banding

The first step in banding is to create a table of data that contains the upper and lower values for each band, as well as a text description. You can type these values directly into Power BI. Follow these steps:

1. Click Enter Data in Power BI.
2. Enter the data into the form as shown below. Make sure you give the table a name and then click Load.



Note: It is important to set up the banding table so there is no crossover of ages between the Low and High ranges. The table above covers all possible ages between 0 and 999, without any duplication. Of course, the 999 value is any arbitrarily large value to catch everyone.

Note: There is no need to join this table to any other table in the data model. In fact, there is no workable way you can do that anyway. Even if there were an age column in the Customers table, you still couldn't join this table to the age column. This banding table doesn't contain all the possible ages for customers; it just has the age bands. So if you first create a customer age column and then join the Low column to this new column, the data will only match for customers who are 20, 30, 40, etc. There will be no match for customers with ages that don't end in a zero (e.g., 21, 22, 23, etc.). So that is not going to work. This table is not joined; hence, it is called a disconnected table.

3. Right-click the Customers table in Data view, select New Column, and enter the following formula:

```
Age =(DATE(2003,1,1) - Customers[BirthDate])/365
```

Note: Although it is not required to make this banding technique work, you could enhance this formula with some rounding, as follows:

```
= ROUNDDOWN((date(2003,1,1) - Customers[BirthDate])/365,0)
```

4. Now you have a new calculated column, as shown above, and you can write some DAX to create the banding column.
5. Right-click the Customers table, select New Column, and enter the following formula:

```
Age Group = CALCULATE(VALUES(AgeBands[Band]),
    FILTER(AgeBands,
        Customers[Age] >= AgeBands[Low]
     && Customers[Age]
        < AgeBands[High]
    )
)
```

6. The key to this formula is the `FILTER()` function. This function iterates over the `AgeBands` table and checks each customer's age against the low and high values for each band. There is only ever one single row in the `AgeBands` table that matches the age of the customer. The `FILTER()` function inside `CALCULATE()` first filters the `AgeBands` table so that only the one row that matches the age band is left visible. Then `CALCULATE()` evaluates the expression `VALUES(AgeBands[Band])`, and because there is only one row visible, `VALUES()` returns the name of the band as a text value into the column.

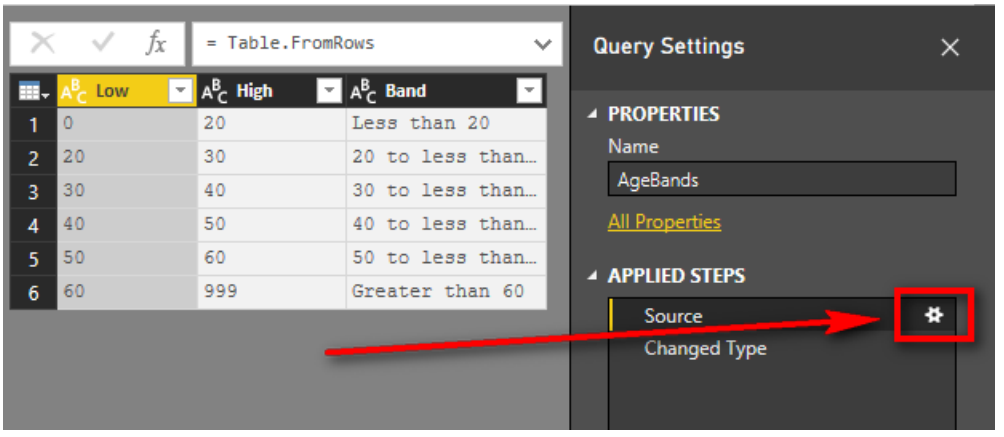
Note: There are two main benefits of taking this approach to banding:

- The DAX formula is easier to read and understand. Once you get used to the concept, it is easier to write, too.
- It is easy to make changes in the future. For example, if you want to add another age band to your analysis (e.g., a new "Greater than 70" age band), all you need to do is add another row to your AgeBands table and then click Refresh.

Here's How: Editing a Table Previously Created with Enter Data

Add a new row to the table, like this:

1. Right-click the AgeBands table in the fields list and select Edit Query.
2. In the Query Editor (shown below), click on the cog next to the Source step.



3. Edit the data in the dialog box to add the new bands of data. Your table should look like the one shown below.

Create Table

	Low	High	Band
1	0	20	Less than 20
2	20	30	20 to less than 30
3	30	40	30 to less than 40
4	40	50	40 to less than 50
5	50	60	50 to less than 60
6	60	70	60 to less than 70
7	70	999	Greater than 70
*			

4. Click OK.
5. Click Close & Apply on the Query Editor Ribbon and then save the pbix workbook.

Maintaining a banding table like this is much easier than editing a complex nested `IF` statement.

It's time to use this new calculated column in a visual. Create a new matrix on a new sheet. Put `Customers[Age Group]` on Rows and then add a couple of the measures you wrote earlier (in Practice Exercise 15):

```
[Total Customers That Have Purchased][Total Sales]
```

You can also add some conditional formatting so that the matrix is easier to read. You should end up with a matrix something like the one shown below.

Age Group	Customers That Have Purchased	Total Sales
20 to less than 30	3,319	$4,356,580
30 to less than 40	6,301	$11,537,347
40 to less than 50	4,937	$8,585,476
50 to less than 60	2,727	$3,685,270
60 to less than 70	1,076	$1,117,530
Greater than 70	124	$76,475
Total	**18,484**	**$29,358,677**

It is easy to see the power of banding. It is unlikely that you will ever want to analyse a business based on sales to customers who are 20, 21, 22, etc. Grouping customers into age brackets is more practical, and this disconnected table banding technique makes it a snap.

Interim Calculated Columns

In the banding example, you first created an `Age` calculated column and then created an `Age Group` calculated column. Breaking the problem into parts like this makes the DAX easier to read, write, and debug. However, you should be aware that it is generally not considered good practice to leave interim calculated columns in a data model as they inefficiently take up extra space (unless you want to use the interim column in your data model as well, of course). What you really should do after you get the final calculated column working as expected is combine all the unwanted interim columns into a single final calculated column, and then delete the unwanted interim columns. This will save space in your workbook which will improve efficiency. Making this change could also make the formula harder to read. To solve this problem, I am going to introduce you to the concept of variables in DAX. Let me first explain the variables syntax and then I will show you how to remove the interim column.

Variables Syntax

`Two keywords in DAX allow you to create and refer to variables in your DAX formulas.` The first keyword is `VAR` (which stands for Variable).

> **Note** In reality `VAR` is more like a constant than a variable as its value cannot change during evaluation.

`VAR` is always accompanied by a second keyword, `RETURN`.

Here is the syntax for `VAR`:

```
My Column (or Measure) =
VAR FirstVariableName = <valid DAX expression>
VAR SecondVariableName = <another DAX expression>
Return
<another DAX expression that can reference the variables>
```

The above generic syntax can be a bit confusing, so let me show you a real example using the formula from above.

```
1  Age Group =
2  VAR Age =
3      ROUNDDOWN ( ( DATE ( 2003, 1, 1 ) - Customers[BirthDate] ) / 365, 0 )
4  RETURN
5      CALCULATE (
6          VALUES ( AgeBands[Band] ),
7          FILTER ( AgeBands, Age >= AgeBands[Low] && Age < AgeBands[High] )
8      )
```

Note how lines 2 and 3 in the formula above set the value of the variable Age to be the value that was previously stored in the original Age calculated column from earlier in this exercise. Once the variable has been set, it is referred to again (twice) in line 7.

Some Important Things to Note

A variable can refer to another variable as shown in line 4 below

```
1  Age Group =
2  VAR AgeInDays =
3      ROUNDDOWN ( ( DATE ( 2003, 1, 1 ) - Customers[BirthDate] ), 0 )
4  VAR Age = AgeInDays / 365
5  RETURN
6      CALCULATE (
7          VALUES ( AgeBands[Band] ),
8          FILTER ( AgeBands, Age >= AgeBands[Low] && Age < AgeBands[High] )
9      )
```

A variable can contain a table as well as a value as shown in row 5 below.

```
1  Age Group =
2  VAR AgeInDays =
3      ROUNDDOWN ( ( DATE ( 2003, 1, 1 ) - Customers[BirthDate] ), 0 )
4  VAR Age = AgeInDays / 365
5  VAR BandsTable =
6      FILTER ( AgeBands, Age >= AgeBands[Low] && Age < AgeBands[High] )
7  RETURN
8      CALCULATE ( VALUES ( AgeBands[Band] ), BandsTable )
```

Variables are set in the initial filter and row context. It doesn't matter if the filter and/or row context changes after the RETURN keyword, the variables have already been assigned and hence they will not change as a result of any changing filter or row context.

Now you know how the VAR syntax works, let me show you how to remove the interim calculated column and move everything into the final banding column.

Here's How: Deleting Interim Calculated Columns

Follow these steps to combine the interim column into the final banding calculated column, and then delete the interim column:

1. Navigate to the interim calculated column in the table (Age in this example).
2. Highlight the formula, then Ctrl+C to copy the entire formula from the interim column as shown below.

```
Age = ROUNDDOWN((date(2003,1,1) - Customers[BirthDate])/365,0)
```

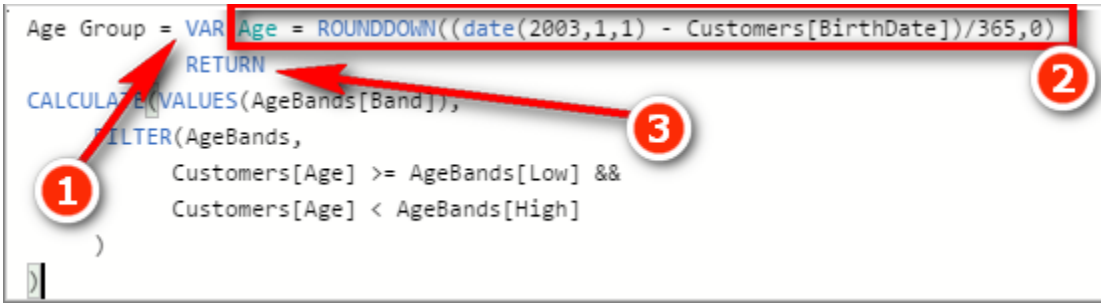
3. Navigate to the final banding calculated column (Age Group in this example). You can enlarge the formula bar by clicking the drop-down arrow in the top right if needed.

4. Create two new blank lines after the = in the formula (press Shift+Enter). You should now have as shown below

```
Age Group =


CALCULATE(VALUES(AgeBands[Band]),
    FILTER(AgeBands,
        Customers[Age] >= AgeBands[Low] &&
        Customers[Age] < AgeBands[High]
    )
)
```

5. Type the keyword `VAR` (#1 below), paste the `Age` column code (#2), and then type the `RETURN` keyword (#3) as shown below.



6. Replace the two instances of the original column names `Customers[Age]` with the reference to the variable `Age` as shown below.

```
Age Group = VAR Age = ROUNDDOWN((date(2003,1,1) - Customers[BirthDate])/365,0)
            RETURN
CALCULATE(VALUES(AgeBands[Band]),
    FILTER(AgeBands,
        Age >= AgeBands[Low] &&
        Age < AgeBands[High]
    )
)
```

7. Delete the interim column `Customers[Age]`.

Note: Of course, if you need the interim column in your table, you should keep it. But if you don't need it, you should remove it by using the process shown above. There is deeper coverage of the use of variables at my blog: https://exceleratorbi.com.au/using-variables-dax/.

18: Concept: Multiple Data Tables

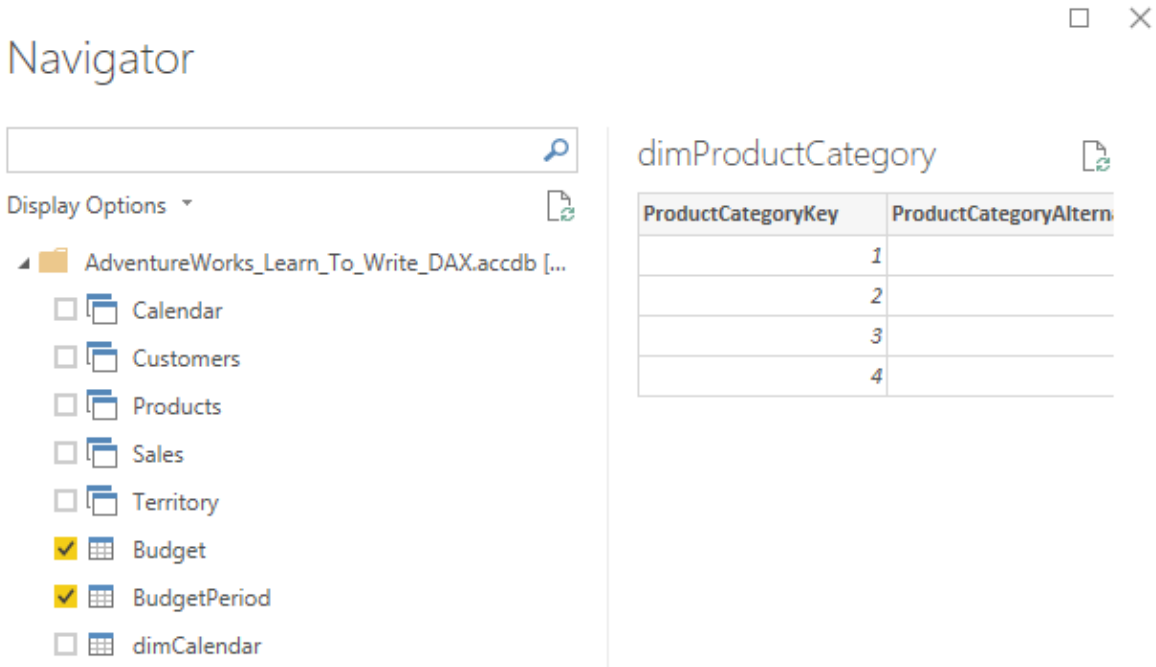
So far in this book, we have used only a single data table, the `Sales` table. It is quite likely that you will want or need to use multiple data tables in your data models. When you bring a second data table into Power BI, it is common for people to think that they should join the new data table to the original data table, but this is incorrect. The correct way to join a second data table to a data model is to treat the new data table exactly the same as the first data table.

To help you understand how to do this, let's look at a common business scenario in which a business wants to load a budget table as well as a sales table. One of the challenges of this scenario is that the budget is often at a different level of granularity than actual sales. For example, sales may be captured and reported every day for every individual product, but budgets may be set only for each month and for each product category.

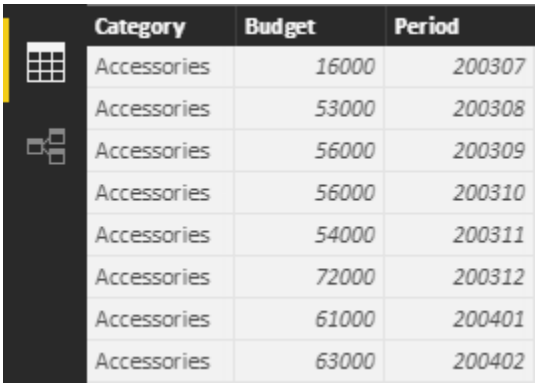
Here's How: Adding a Budget Table

The following steps walk you through the process of importing a `Budget` table, creating a new `BudgetPeriod` table, and then creating a measure for the budget:

1. In Power BI, click Get Data, More, Access Database and navigate to the Access database you used in Chapter 2.
2. Select the `Budget` and `BudgetPeriod` tables from the list, as shown below.



3. Click Load. You now see that the Budget table has a monthly sales budget for each category. The Period column is in the format YYYYMM for year and month, as shown below.



Category	Budget	Period
Accessories	16000	200307
Accessories	53000	200308
Accessories	56000	200309
Accessories	56000	200310
Accessories	54000	200311
Accessories	72000	200312
Accessories	61000	200401
Accessories	63000	200402

4. You can also see that the BudgetPeriod table is a type of calendar table and is different from what you have used so far. Like the Budget table, it contains a Period column in the format YYYYMM, as shown below.

CalendarYear	MonthName	Month Number	Period
2003	July	7	200307
2003	August	8	200308
2003	September	9	200309
2003	October	10	200310
2003	November	11	200311
2003	December	12	200312
2004	January	1	200401
2004	February	2	200402

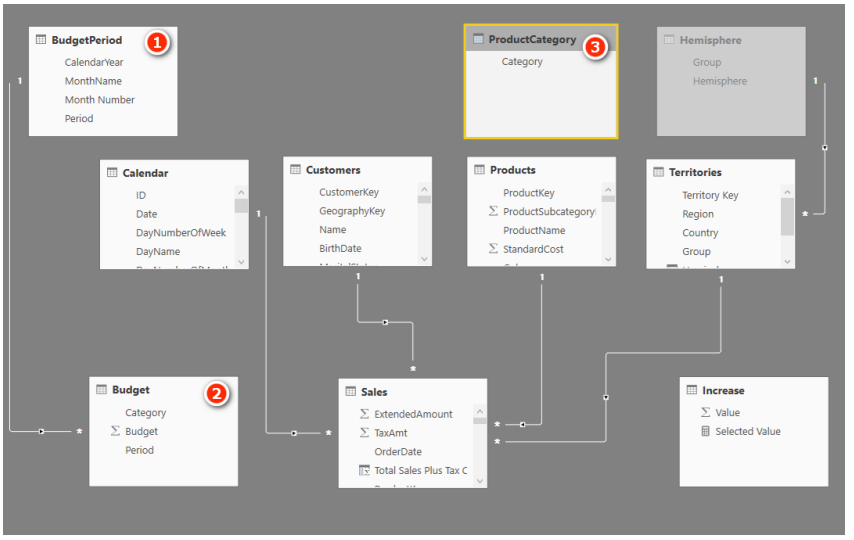
5. You need a `ProductCategory` table, so click the New Table button and type the formula shown below.

6. The new ProductCategory table has a list of the four possible product categories.

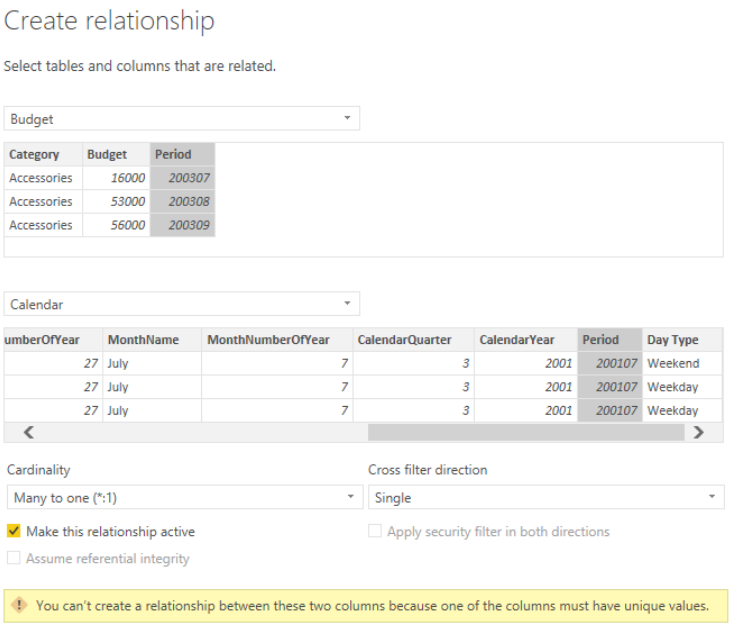
Note: The reason you need all these new tables will make sense shortly.

7. Switch to Relationships view.
8. Rearrange your tables as shown below. Place the BudgetPeriod table (see #1 below) above the Calendar table and place the Budget table (#2) next to the Sales table. Put the ProductCategory table (#3) above the Products table, as shown.

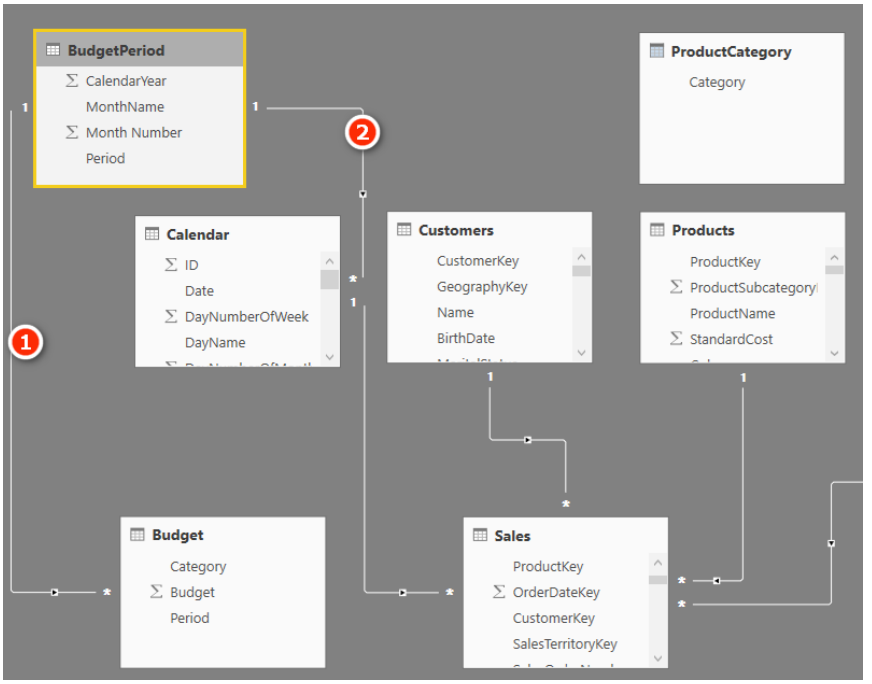
Note: The relationship between `BudgetPeriod` and `Budget` is a physical relationship automatically created by Power BI when the New `BudgetPeriod` table was created. Don't confuse this with *lineage*, which is an inherited relationship for a virtual table that exists only during formula evaluation.



9. Now let me explain why you need the BudgetPeriod table. Go ahead and try to join the Budget table to the Calendar table. It is a bit hard to drag and drop the `Period` column, so instead go to the Home tab, click Manage Relationships, and try to create a new relationship between the `Budget` table and the `Calendar` table, as shown below. Note the error message.

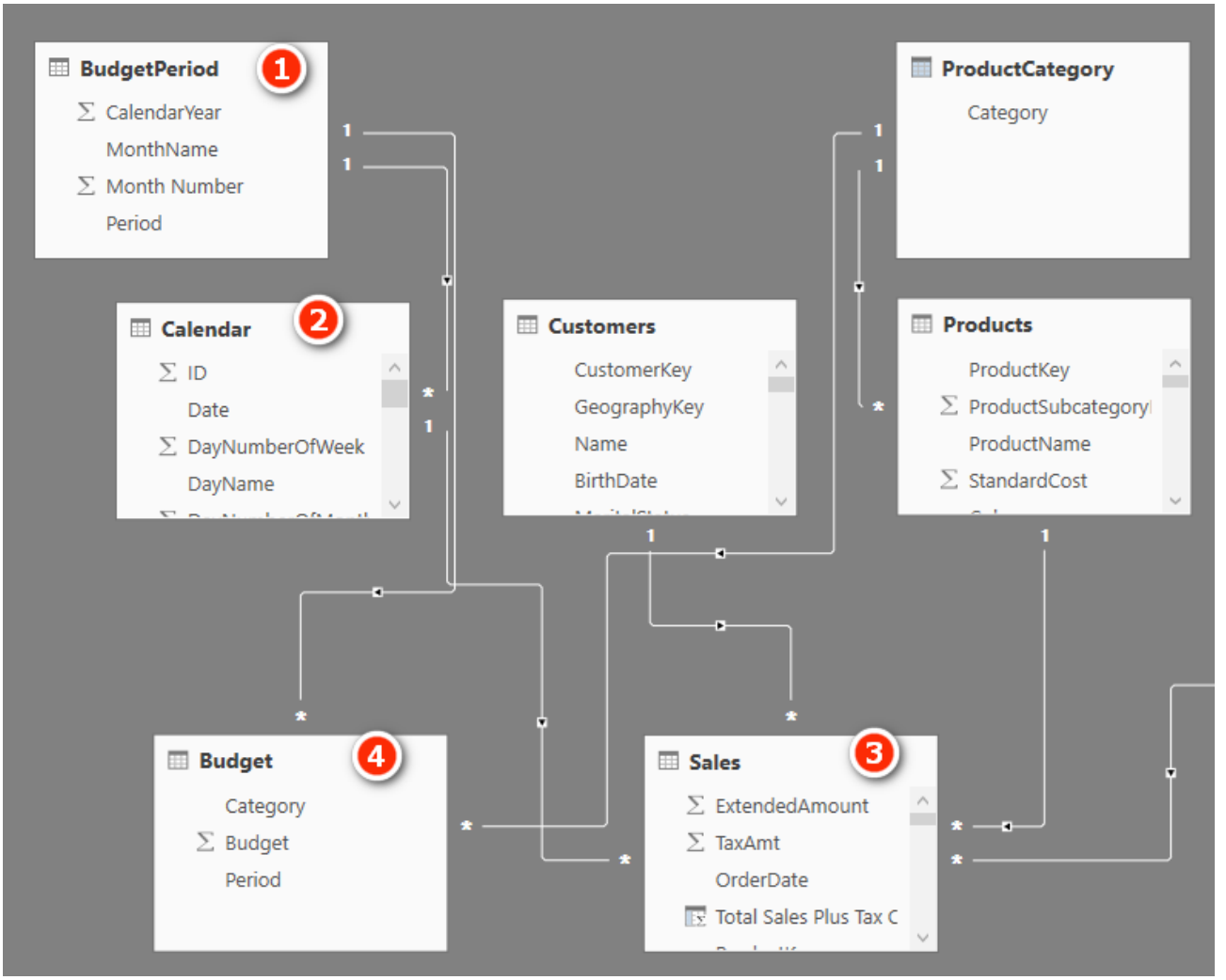


10. Click Cancel and close the Create Relationship window. Do you see the issue? The Calendar table is a daily calendar, but the Budget table is a monthly budget (a very common business scenario). The Period column in the Calendar table has between 28 and 31 entries for each month in the budget table. *But Power BI supports only one-to-many relationships.* The lookup table (Calendar) at the top simply must have a single value for Period if you are to make the join, so this is not going to work. You therefore need the BudgetPeriod table. There is only one value for each period in the Budget-Period table, and hence you are able to join the Budget table to the BudgetPeriod table. In fact, this relationship (see #1 below) was auto-created when the data was loaded.
11. Join the Calendar table to the BudgetPeriod table by dragging the Period column from the Calendar table to the Period column in the BudgetPeriod table (see #2 below).



12. Now you need to join the ProductCategory table to the Budget table. If you try to join the Budget[Category] column to the Products table, you get the same error as before.
13. To join the Budget table to the ProductCategory table, click and drag the column `Budget[Category]` to `ProductCategory[Category]`.
14. To join the Products table to the ProductCategory table, click and drag the column `Products[Category]` to the `ProductCategory[Category]` column.

When you are finished, you should have something like the layout shown below. Notice that it becomes difficult to keep track of all the relationships when you have lots of tables in a data model. This is one reason I recommend arranging the tables using the Collie layout methodology, as shown below.



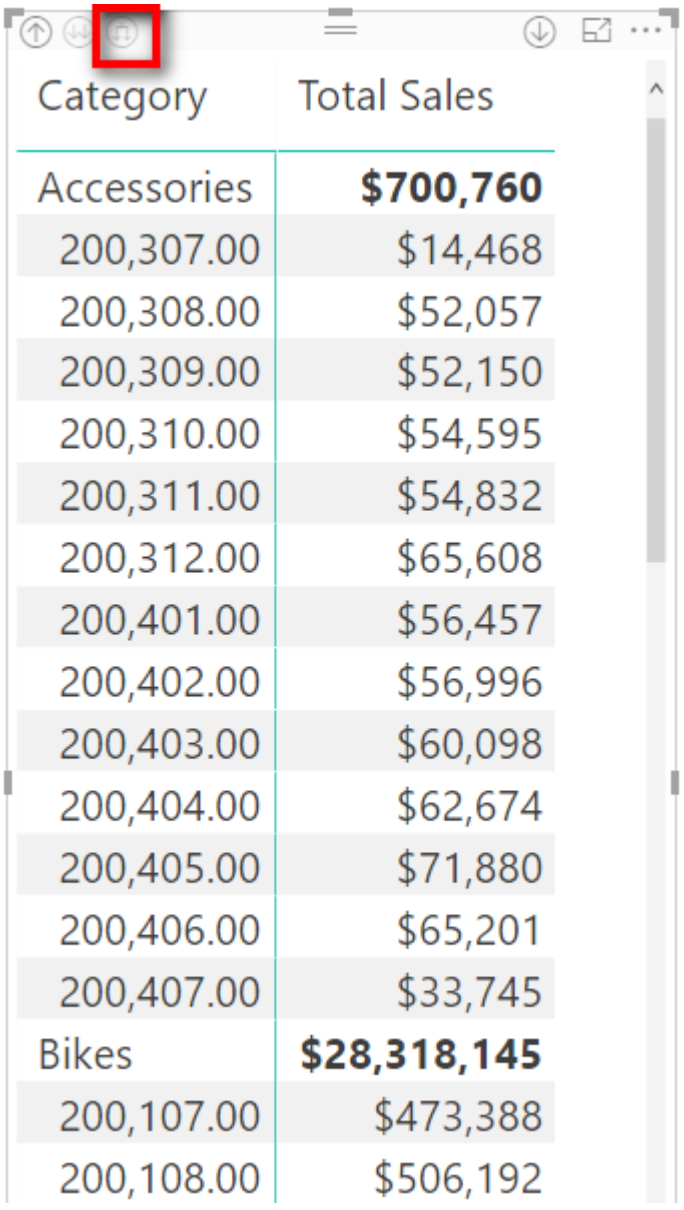
As you can see in the image above, the tables on the "many" side of the relationship should be down below, and the tables on the "one" side of the relationship should be up high. The filters always flow downhill, and this layout makes it much easier to understand how the filters flow. So if you filter on the `BudgetPeriod` table (see #1 above), this table directly filters the `Budget` table (#4) via the direct relationship. In addition, the `BudgetPeriod` table (see #1) directly filters the `Calendar` table (#2), and the `Calendar` table (#2) filters the `Sales` table (#3). So the net result is that any filter you apply to the `BudgetPeriod` table (#1) filters both the `Sales` table (#3) and the `Budget` table (#4). The same concept applies with the `ProductCategory` table.

When working with data tables of differing granularities, as in this case, it is important to use the correct tables and columns in your matrix filters. So when working with both the `Sales` table and the `Budget` table, you must use the columns from the `BudgetPeriod` table in your matrixes; *columns from the* `Calendar` *table will not work*.

Practice Exercises: Multiple Data Tables

It's time to get some practice writing new DAX formulas across the two data tables: `Budget` and `Sales`. First, create a new matrix. Then put `ProductCategory[Category]` on Rows, `BudgetPeriod[Period]` on Rows, and `[Total Sales]` on Values. Make sure you select the correct columns from the two new tables (`ProductCategory` and `BudgetPeriod`).

Once your matrix is set up, click on the Expand All Down icon, as shown below, to expand all levels in the matrix.



Category	Total Sales
Accessories	**$700,760**
200,307.00	$14,468
200,308.00	$52,057
200,309.00	$52,150
200,310.00	$54,595
200,311.00	$54,832
200,312.00	$65,608
200,401.00	$56,457
200,402.00	$56,996
200,403.00	$60,098
200,404.00	$62,674
200,405.00	$71,880
200,406.00	$65,201
200,407.00	$33,745
Bikes	**$28,318,145**
200,107.00	$473,388
200,108.00	$506,192

Note that the periods shown on Rows in the image above are not formatted properly. The periods should be in the format YYYYMM, but they are displayed with commas and decimal points. This is an easy fix. Just select the `BudgetPeriod[Period]` column in the fields list and change the data type to Whole Number.

Now it is time to create some measures. Right-click the `Budget` table, select New Measure, and then write the following new measures. Find the solutions to these practice exercises in"Appendix A: Answers to Practice Exercises" on page 178.

71. [Total Budget]

72. [Change in Sales vs. Budget]

73. [% Change in Sales vs. Budget]

The image below shows what the matrix looks like with these formulas and the addition of conditional formatting.

Category	Total Sales	Total Budget	Change in Sales vs Budget	% Change in Sales vs Budget
Accessories	**$700,760**	**$739,000**	**-$38,240**	**-5.2%**
200307	$14,468	$16,000	-$1,532	-9.6%
200308	$52,057	$53,000	-$943	-1.8%
200309	$52,150	$56,000	-$3,850	-6.9%
200310	$54,595	$56,000	-$1,405	-2.5%
200311	$54,832	$54,000	$832	1.5%
200312	$65,608	$72,000	-$6,392	-8.9%
200401	$56,457	$61,000	-$4,543	-7.4%
200402	$56,996	$63,000	-$6,004	-9.5%
200403	$60,098	$63,000	-$2,902	-4.6%
200404	$62,674	$68,000	-$5,326	-7.8%
200405	$71,880	$78,000	-$6,120	-7.8%
200406	$65,201	$63,000	$2,201	3.5%
200407	$33,745	$36,000	-$2,255	-6.3%
Bikes	**$28,318,145**	**$28,750,000**	**-$431,855**	**-1.5%**
200107	$473,388	$483,000	-$9,612	-2.0%
200108	$506,192	$516,000	-$9,808	-1.9%
200109	$473,943	$502,000	-$28,057	-5.6%

19: Concept: Using Analyze in Excel and Cube Formulas

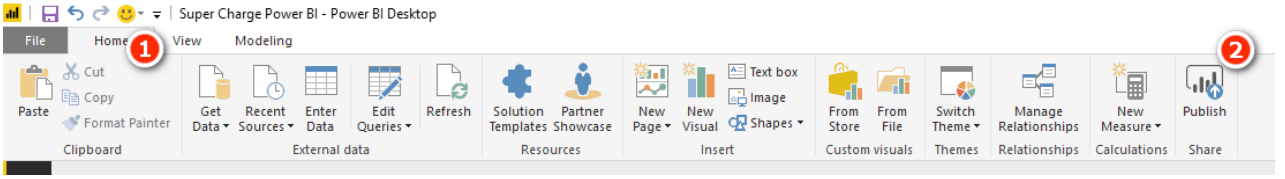
So far in this book, you have consumed and visualised the information from data models in reports inside Power BI. But after you have invested so much time and effort building your Power BI data models, you may want to get to the data by using good old traditional Excel. Fortunately, there is an easy way to do this from PowerBI.com, as long as you have a Power BI Pro licence: You can use Analyze in Excel. Before you can use Analyze in Excel, however, you must first publish your Power BI Desktop file to PowerBI.com.

Note: If for some reason you are not able to publish to PowerBI.com, you can still complete the cube formulas exercises later in this chapter. Simply go to my website, http://xbi.com.au/localhost, and follow the instructions on how to connect Excel to a local instance of Power BI Desktop running on your PC.

Here's How: Publishing a Report to PowerBI.com

Follow these steps to publish a report to PowerBI.com:

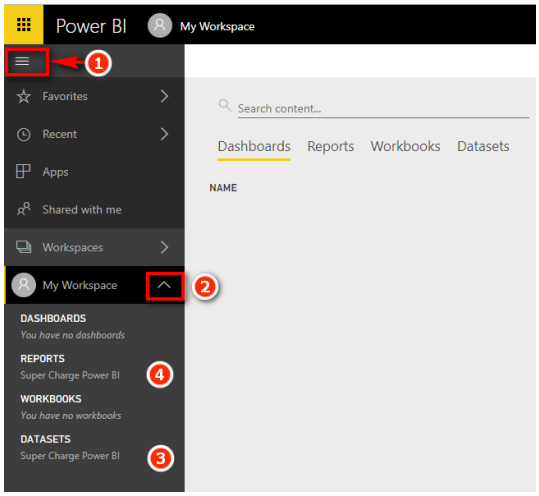
1. Save the Power BI workbook.
2. On the Home tab (see #1 below), click the Publish button (#2).



3. If this is the first time you are using PowerBI.com, you will be prompted to create an account. Simply follow the instructions to create yourself a new account. If you already have an account, just sign in with your credentials. You get a success message when the file has been loaded to PowerBI.com. If you are creating an account for the first time, you need to activate the 60-day trial in order to use Analyze in Excel.

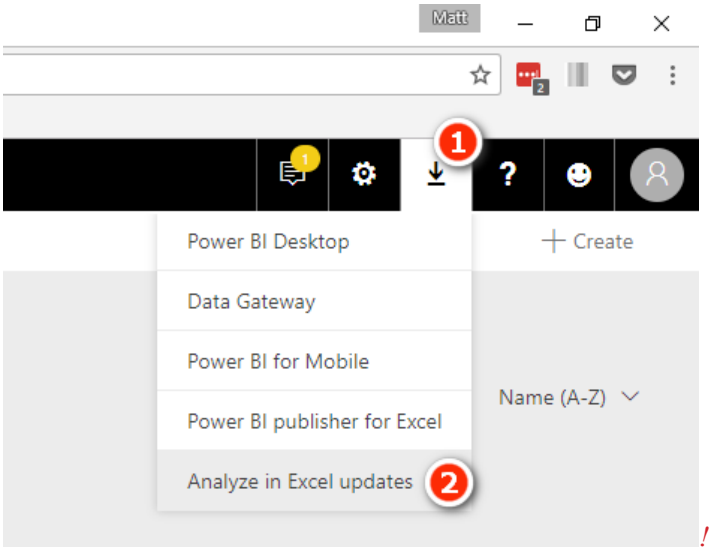
Note: Your access to PowerBI.com may be controlled by your IT department. If you cannot create an account as described above, you may need to contact your IT department for support. Also, as at this writing, it is not possible to sign up to PowerBI.com using a generic email address such as @gmail.com or @hotmail.com

4. In a browser, navigate to http://powerbi.com and click the sign-in link in the top-right corner of the website, using the same credentials you specified in step 3.
5. Once you are logged in, expand the menu on the left-hand side (see #1 here) and open the workspace where you uploaded your workbook, such as My Workspace (#2).
6. Find your data model under Datasets (see #3 here) and all your reports in the Reports section (#4).



Before you can use Analyze in Excel the first time, you need to install the Analyze in Excel updates. Follow these steps:

1. Click on the download arrow in the top-right corner of the browser (see #1 below) and select Analyze in Excel Updates (#2).
2. After the update is downloaded, run the downloaded file on your PC. You need administration rights on your computer to be able to complete the installation.

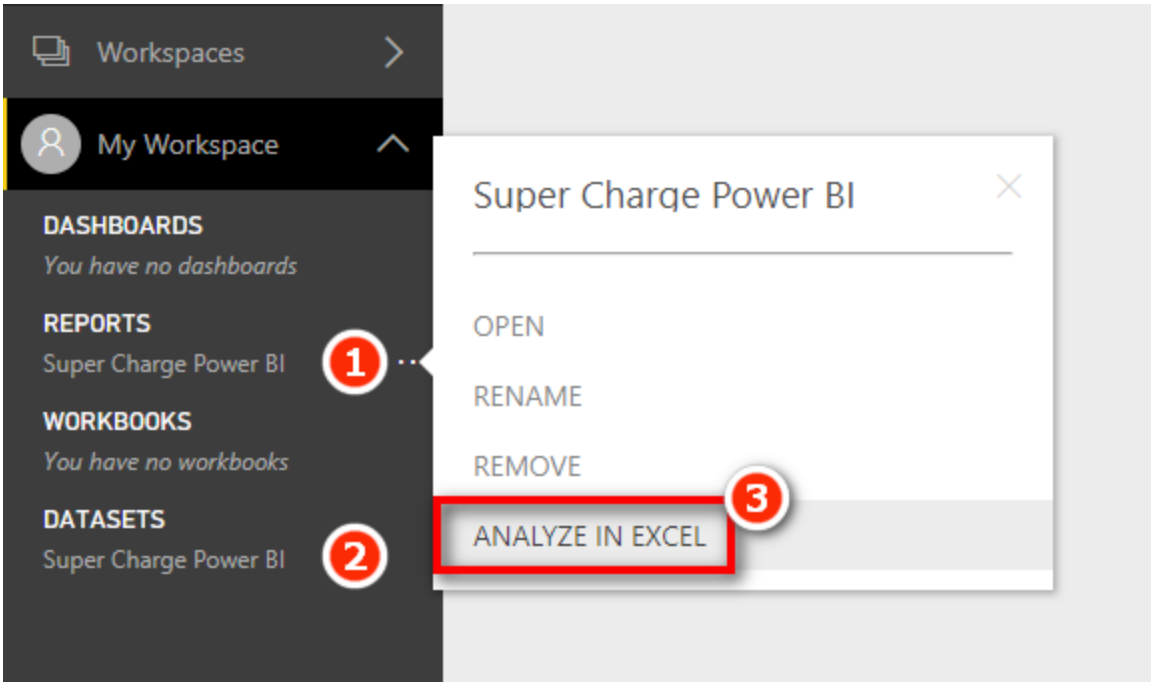


Here's How: Using Analyze in Excel

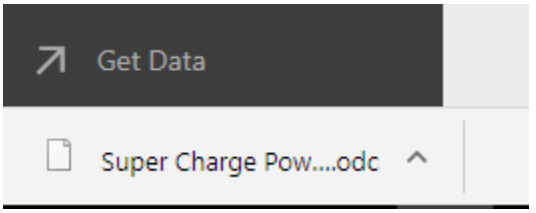
Using Analyze in Excel could not be easier. Simply follow these instructions:

1. Navigate to either Reports (see #1 below) or Datasets (#2) on the left-hand side and right-click the ellipsis to open the menu and then select Analyze in Excel (#3). An ODC file is then downloaded to your PC.

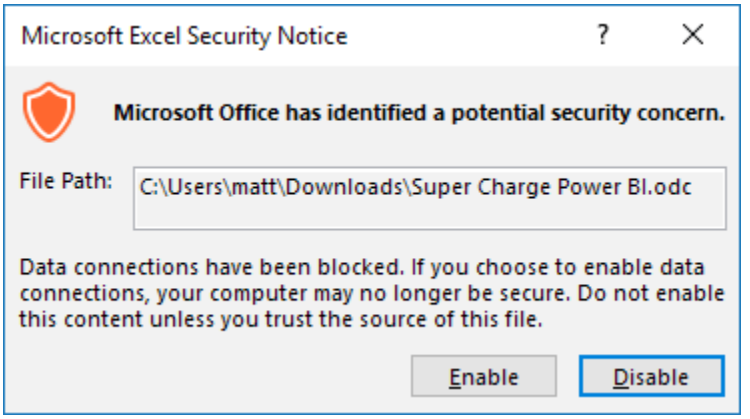
Note: An ODC file is a small text file that contains a set of instructions to tell Excel how to connect directly to Power BI. You can open an ODC file with a text editor and see what it contains if you are interested.



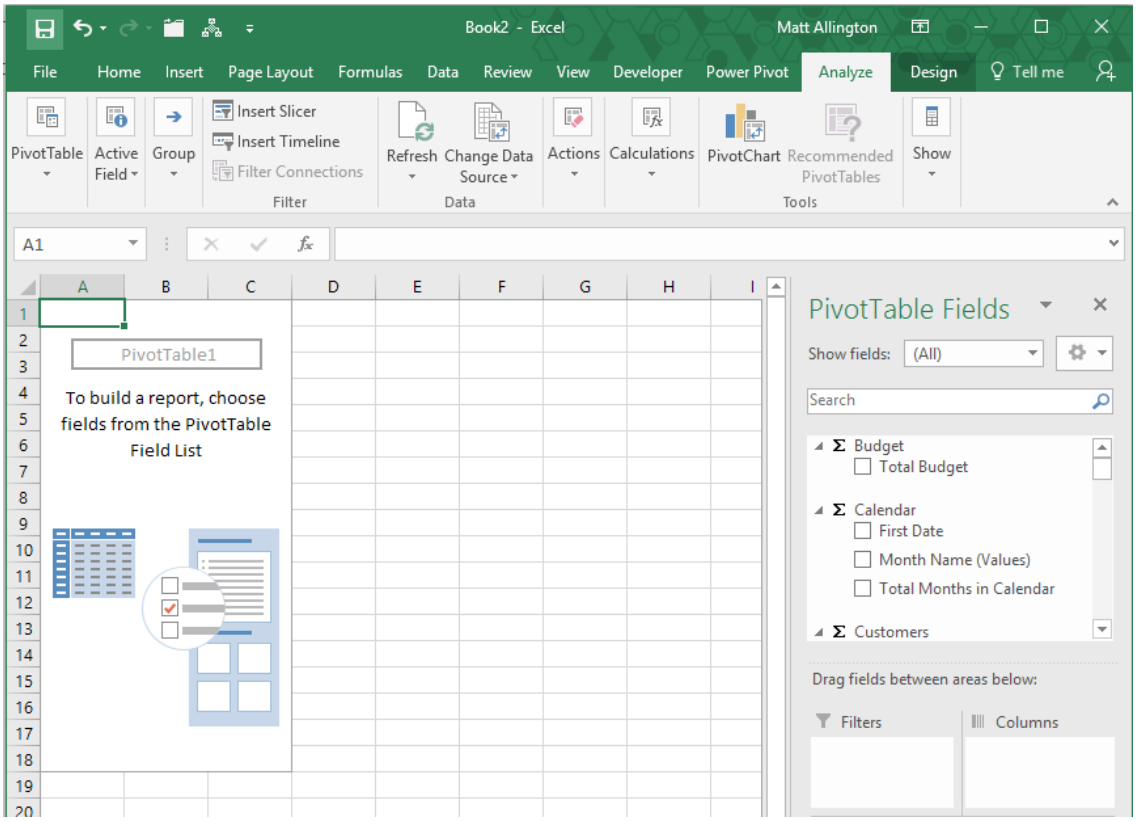
2. Find the ODC file that was downloaded (probably to your downloads folder, depending on your browser) and click to open it. The figure below shows what this might look like using Google Chrome (bottom left hand corner of the Chrome screen).



3. If you're prompted with a security warning when Excel launches, click Enable.

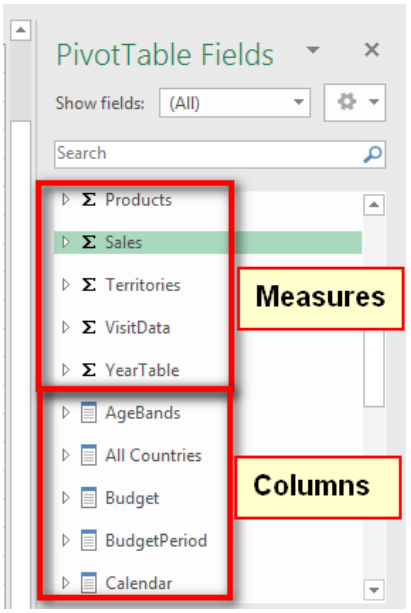


If everything has gone well, you should now have a new pivot table in a new Excel workbook that looks like the one below. You will have to log in again to your account if prompted.

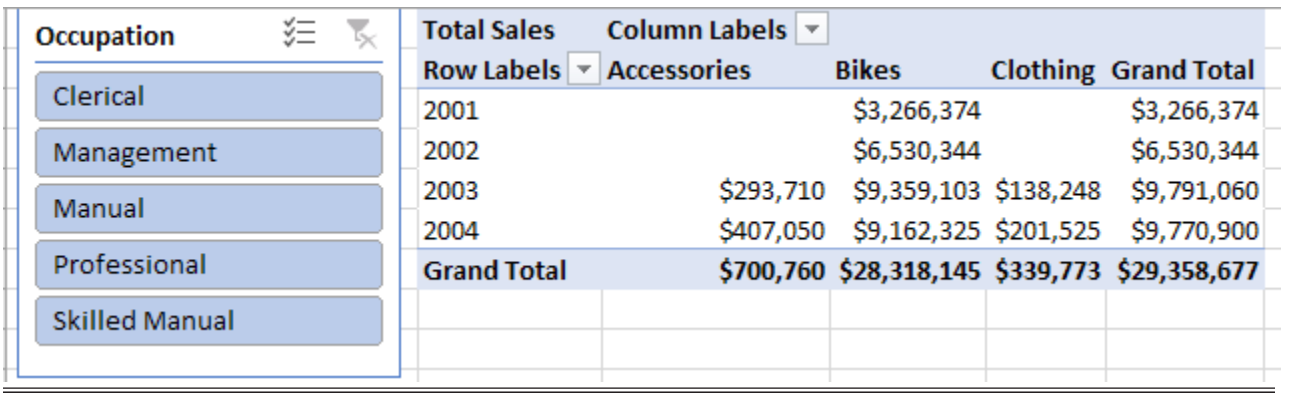


The really cool thing about this Excel workbook/pivot table is that it has a direct connection to PowerBI.com. The data model remains at PowerBI.com, and only the data needed to display the pivot table is stored in the Excel workbook. The data model could be as big as 10 GB on PowerBI.com, but the Excel workbook could be as small as 20 KB. We call these "thin workbooks."

The PivotTable Fields list on the right-hand side of Excel looks slightly different to this list for a regular pivot table. There are measure tables and also column tables, differentiated by the two different icons shown below.



You should be able to build a pivot table similar to one of the matrixes that you have already built in Power BI Desktop. (See the example below.)



Using Cube Formulas

The final concept topic in this book is one of my favourites: cube formulas. Cube formulas have been around for many years. But before Power BI was launched, the main way you could use cube formulas was to connect to a SQL Server Analysis Services (SSAS) multidimensional cube. Some large companies have SSAS set up. Some of those companies may connect directly to SSAS from Excel, and some of the ones that do may have discovered cube formulas. But given how rare this scenario is, most people have not come across cube formulas prior to discovering Power BI.

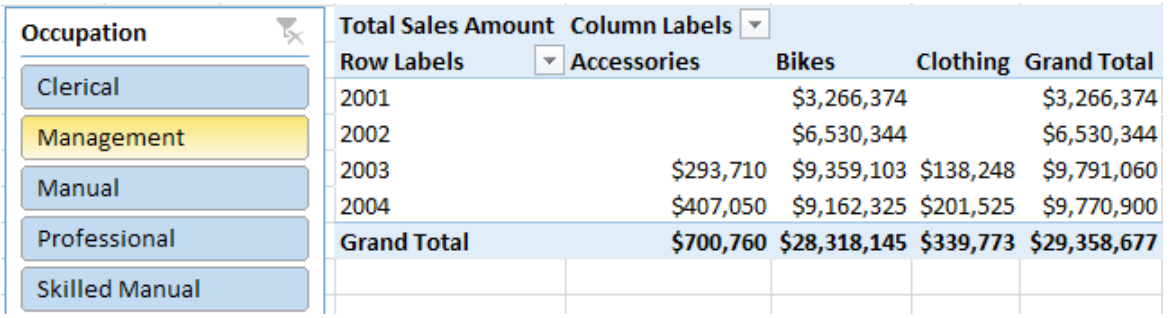
Pivot tables in Excel are great, and I use them all the time, but they do have some limitations. The biggest limitation is that a pivot table locks you into a particular format. But what if you want to put a single value in a single cell in a workbook? In that case, you could create a pivot table and then point the cell in question to the pivot table, but that involves a lot of overhead. In addition, if the pivot table changes shape at any time (e.g., on refresh), then chances are the cell positions will change, and your formula may point to the wrong cell. The best-case scenario is that you realise there is a problem. The worst-case scenario is that your formula points to another similar cell in the pivot table, and you don't even notice!

"What about `GETPIVOTDATA()`?" I hear some of you say. Well, yes, you can use `GETPIVOTDATA()`, but you still have the overhead of the pivot table, and the bottom line is that cube formulas are much better. The easiest way to get started with cube formulas is to convert an existing pivot table to cube formulas. The following pages walk you through how to do that.

Here's How: Converting a Pivot Table to Cube Formulas

Follow these steps to convert a pivot table to cube formulas:

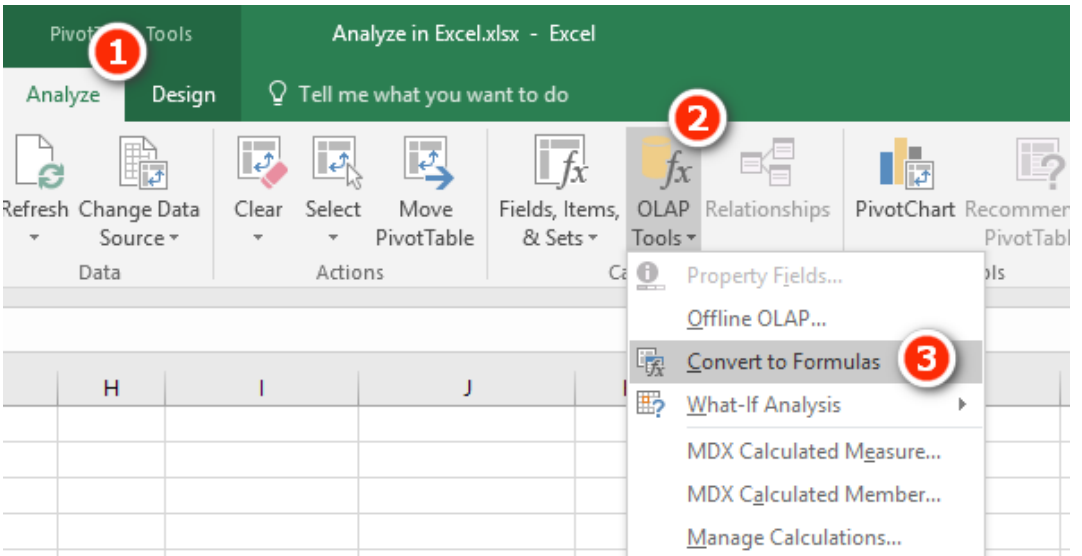
1. Create a new blank sheet in your Excel workbook and insert a pivot table like the one shown below, which is the same one created in Chapter 18.



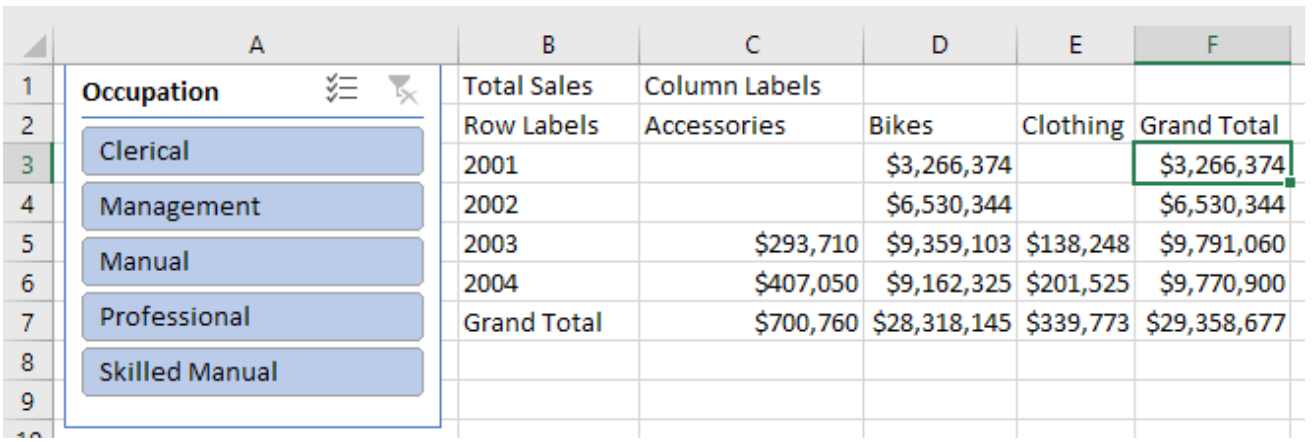
Occupation: Clerical, Management, Manual, Professional, Skilled Manual

Total Sales Amount	Column Labels			
Row Labels	Accessories	Bikes	Clothing	Grand Total
2001		$3,266,374		$3,266,374
2002		$6,530,344		$6,530,344
2003	$293,710	$9,359,103	$138,248	$9,791,060
2004	$407,050	$9,162,325	$201,525	$9,770,900
Grand Total	**$700,760**	**$28,318,145**	**$339,773**	**$29,358,677**

2. Put 'Calendar'[CalendarYear] on Rows, Products[Category] on Columns, and [Total Sales] on Values. Also add a slicer for Customers[Occupation]. Click on the slicer and make sure it works before proceeding.
3. To convert the pivot table to cube formulas, click inside the pivot table and then select the Analyze tab (see #1 below), click OLAP Tools (#2), and select Convert to Formulas (#3).



Bam! Your pivot table is converted to a stack of standalone formulas that you can move around as you want on the spreadsheet.

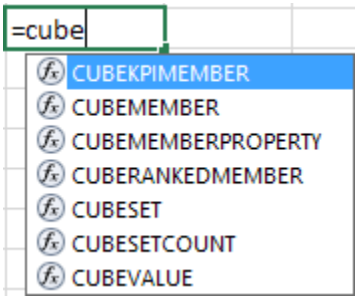


Occupation: Clerical, Management, Manual, Professional, Skilled Manual

	A	B	C	D	E	F
1	Occupation	Total Sales	Column Labels			
2		Row Labels	Accessories	Bikes	Clothing	Grand Total
3		2001		$3,266,374		$3,266,374
4		2002		$6,530,344		$6,530,344
5		2003	$293,710	$9,359,103	$138,248	$9,791,060
6		2004	$407,050	$9,162,325	$201,525	$9,770,900
7		Grand Total	$700,760	$28,318,145	$339,773	$29,358,677
8						
9						

What's more, the slicer still works! Go ahead and drag the formulas around to new locations in your spreadsheet and then click on the slicer to verify that it works.

Writing Your Own Cube Formulas

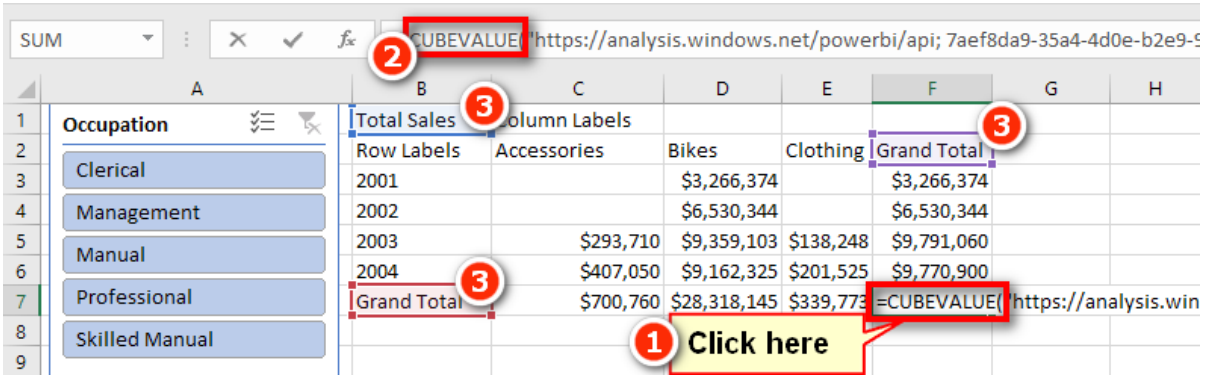


There are seven cube formulas in total in the family, and they all start with the word `CUBE`. You can see the list by typing `=CUBE` into a cell in your workbook, as shown below.

This book covers the two most-used formulas, `CUBEVALUE()` and `CUBEMEMBER()`. Once you have mastered these two formulas, you can do some research to learn about the other five.

CUBEVALUE() vs. CUBEMEMBER()

Go back to the pivot table that you just converted and double-click inside the grand total cell (see #1 below) so that Excel is in Edit mode. Notice in the formula bar (#2) that this grand total cell is a `CUBEVALUE()` formula, and it points to a number of other cells (#3). The formulas inside each of these other cells (labelled #3) are `CUBEMEMBER()` formulas.



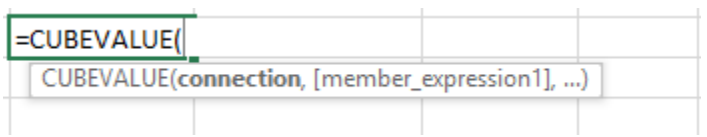
`CUBEVALUE()` is used to extract the value of a measure from the data model, and `CUBEMEMBER()` is used to extract a value from a column/lookup table. When they are used together, `CUBEMEMBER()` filters the data model before calculating the `CUBEVALUE()` expression.

Now that you know about cube formulas, you can build a pivot table that contains the cube formulas you want in your spreadsheet and then simply select Analyze, OLAP Tools, Convert to Formulas. Once you have done this, you can copy and paste the resulting formulas wherever you want. But it actually isn't very hard to write cube formulas from scratch, so let's do that together now.

Here's How: Writing CUBEVALUE() from Scratch

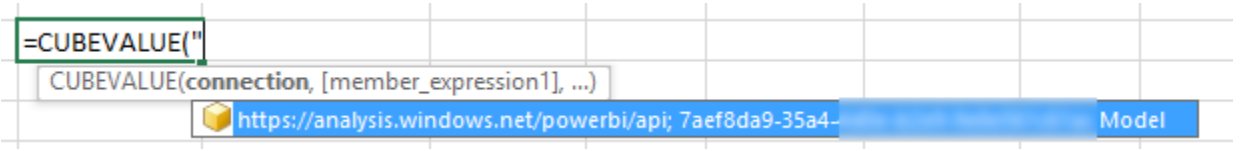
The important keyboard keys when writing cube formulas are the double quote, the square brackets, and the full stop (period in the USA). This information will make sense as you work through these steps. Be sure to follow these steps exactly:

1. Click in an empty cell in your workbook and type `=CUBEVALUE(`. Notice the tooltip that pops up, asking for a connection and one or more member expressions. The member expressions can be either measures or table columns from your data model.



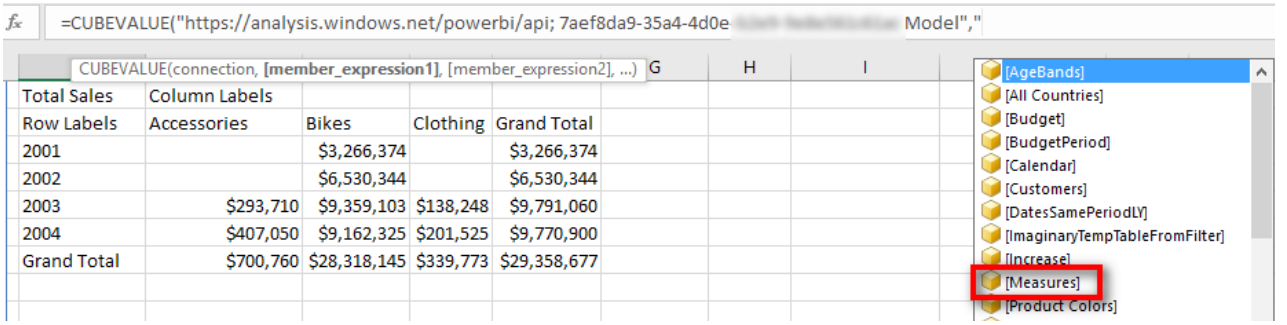
2. Type " (a double quote). You are presented with a list of connections available to the workbook. Given that this is a thin workbook created using Analyze in Excel, you should have a connection

string that looks something like the one shown below. Note that the long number (GUID) is unique to your instance of Power BI. Anyone who can access your Power BI data set (e.g., someone else in your organisation) will also be able to interact with this thin workbook when it is done.

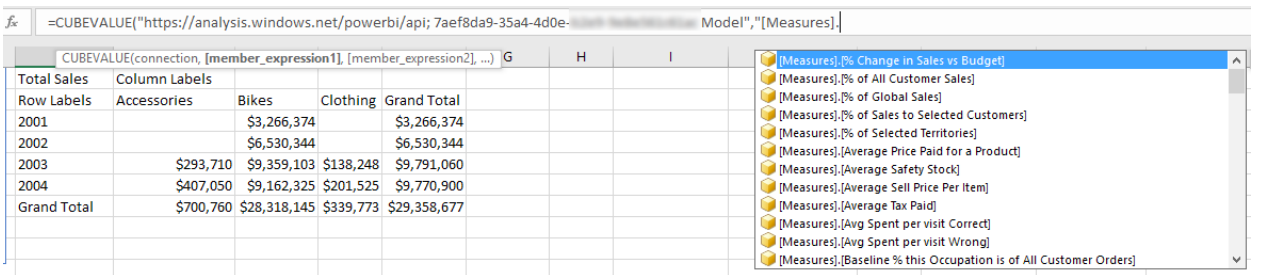


Note: If you are writing a cube formula using my localhost workbook http://xbi.com.au/localhost, then you will get a different connection string experience than shown above.

3. Press the Tab key to select the connection and then type " again.
4. Type , (a comma).
5. Type " (a double quote) again to start the next parameter. This time notice that the tooltip shows you a list of all the tables in the data model (see below). There is also one additional item in the list, [Measures]. All of your DAX formulas are stored in [Measures].



6. Type [and then M and press Tab to select [Measures].
7. Type . (a full stop/period), and you see a list of all the measures that exist in the data model. From here you can either keep typing [followed by the name of the measure or use the up and down arrow keys on the keyboard to navigate to the measure you want to select.

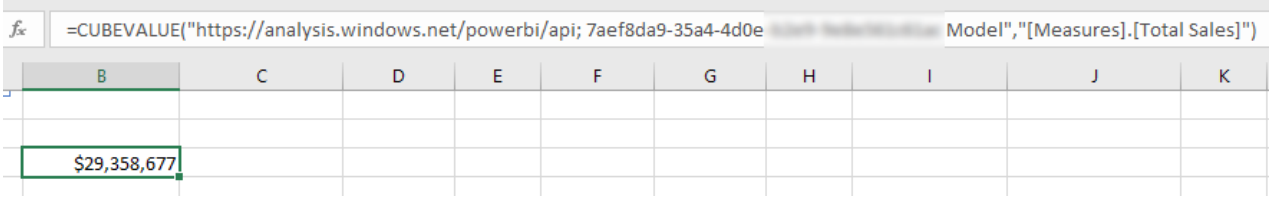


8. Type [and then type Total S. This brings the [Total Sales] measure to the top.
9. Press Tab, type "), and press Enter.

If you follow these instructions exactly, you end up with a value in a cell, as shown below. This is your first handwritten cube formula:

```
= CUBEVALUE(<your connection string>,"[Measures].[Total Sales]")
```

Note: In the formula above, I have used <your connection string> in place of the actual connection string because what you have on your screen will be different.



You probably noticed that the value you end up with after writing this cube formula is the grand total for all the data in the data model. It should therefore be clear that the data model is completely unfiltered. It is possible to filter this formula just as in a pivot table by adding some `CUBEMEMBER()` functions into the formula (sort of like adding a column to Rows in a pivot table).

Note: Before moving on, you should rewrite the formula above a couple of times for practice. Remember that the most important keys on your keyboard in this process are double quotes, square brackets, and the full stop/period, along with Tab to select the highlighted selection. Practice the rhythm of writing these formulas using these keys on the keyboard.

Here's How: Applying Filters to Cube Formulas

To filter an existing formula, follow these steps:

1. Select one of the formulas you have already written and start to edit it.
2. Delete the last `)` and then type `,` (a comma). The tooltip asks for member_expression2.
3. Type `"[`.
4. Use the down arrow key to select [Calendar] and then press Tab.
5. Type `.` (full stop/period) and use the down arrow key to select [CalendarYear]. Then press Tab.
6. Type `.` (full stop/period) and notice that the tooltip offers only a single choice, [All]. Select [All] and then press Tab.
7. Type `.` (full stop/period) again and notice that you now have a list of the possible years to select from. Select `[2003]`.
8. Finish the formula by typing `")` and pressing Enter.

This is the final formula:

```
= CUBEVALUE(<your connection string>,"[Measures].[Total Sales]", "[Calendar].[CalendarYear].[All].[2003]")
```

Go back into this formula again and delete the closing bracket `)`, add another `,` (a comma), and then follow the same process as above to add another cubemember, this time for `Products[Category]` = `"Clothing"`. This is the formula you need:

```
= CUBEVALUE(<your connection string>,
    "[Measures].[Total Sales]",
    "[Calendar].[CalendarYear].[All].[2003]",
    "[Products].[Category].[All].[Clothing]"
        )
```

You can add any measure from your data model into your spreadsheet by writing a cube formula like this. You can further filter the measure in your cube formula by adding additional `CUBEMEMBER()` expressions inside the cube formula you are writing.

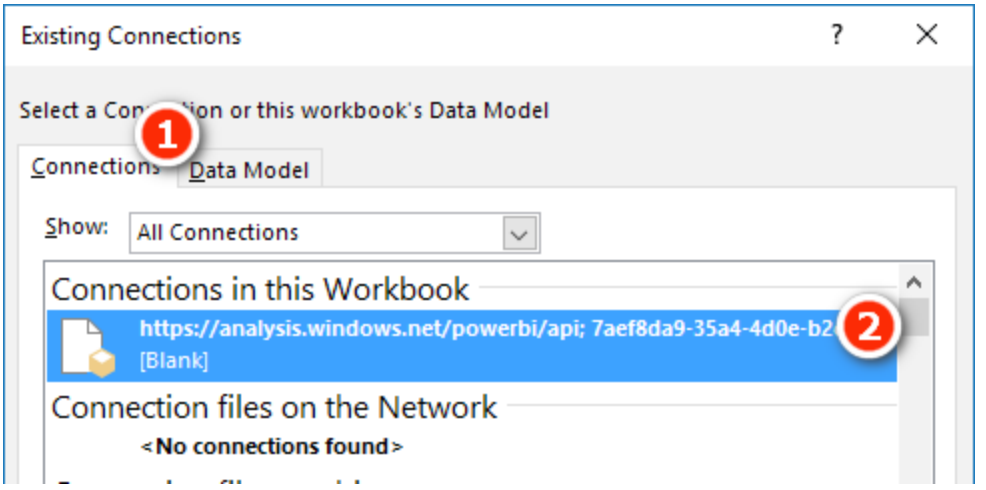
Here's How: Adding a Slicer Without a Pivot Table

Connecting your formulas to slicers is easy. You should have a slicer for `Customers[Occupation]` on the sheet. If you don't have this slicer, then go ahead and add it now. Here are the steps to add a slicer when there is no pivot table:

1. Select Insert, Slicer.

Note: In this case, you can't right-click on a column in the PivotTable Fields list because there is no pivot table.

2. In the Existing Connections dialog that appears, select the Connections tab (see #1 below), select Connections in This Workbook (#2), and then click Open.

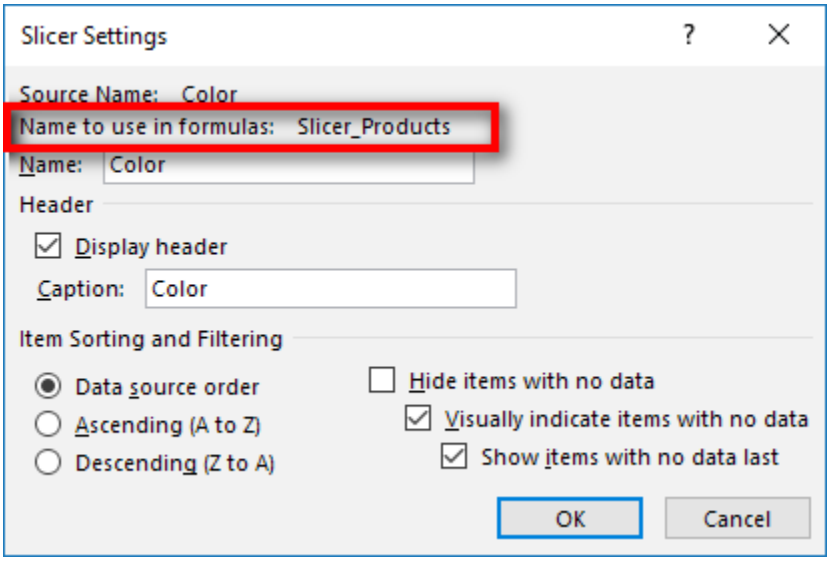


3. Find the Products[color] slicer in the list, select the correct checkbox, and click OK. You now have a slicer on your sheet, but it is not connected to your formula.

Here's How: Connecting a Slicer to a Cube Formula

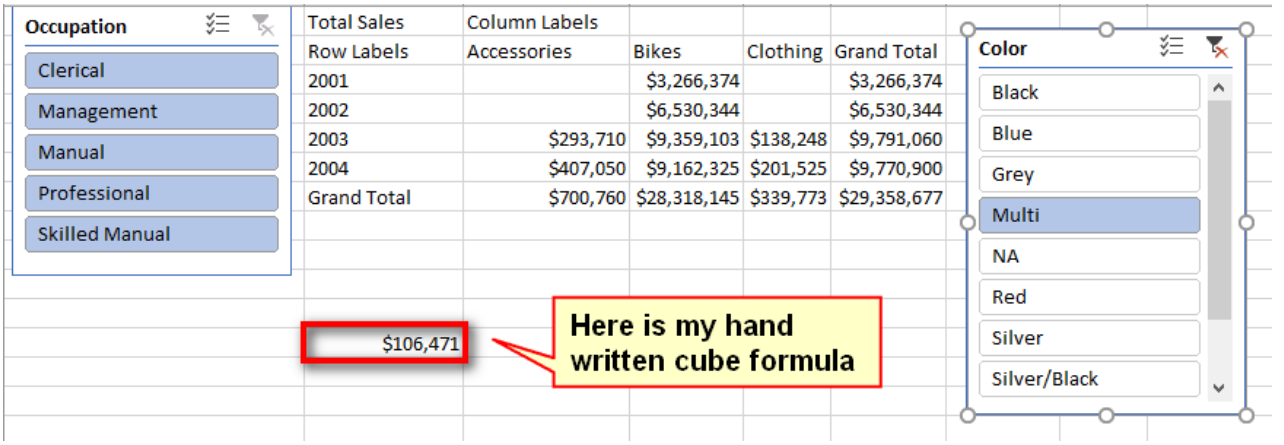
Follow these steps to connect a slicer to a cube formula:

1. Check the unique name for the slicer you want to connect by right-clicking on the slicer and selecting Slicer Settings.
2. In the Slicer Settings dialog that appears, note and memorise the value that appears in the second line in the dialog box, Name to Use in Formulas. You will need the name of the slicer in the next step. In the example shown here, it is called Slicer_Products. In your case, it may be called something different. Once you've noted the slicer name, click Cancel.



3. Write a new version of the Total Sales cube formula and this time add the slicer to this formula. Simply add a comma after `[Total Sales]`, followed by the slicer name from step 2, and then type `)`. Your formula should now look something like this (though your slicer may have a slightly different name):

```
= CUBEVALUE(<Your Connection String>,
    "[Measures].[Total Sales]",
    Slicer_Products
)
```



Note: You do not use double quotes around slicer names. This is an unfortunate inconsistency, but it is just how it works.

4. Now test it out: Click on your slicer and watch your cube formula update.

Take a deep breath and be amazed. How cool are cube formulas?!

Writing CUBEMEMBER() Formulas

In addition to referencing a column name inside a `CUBEVALUE()` formula, it is possible to write a `CUBEMEMBER()` formula directly in a cell in a workbook. Here is an example of a `CUBEMEMBER()` formula:

```
= CUBEMEMBER(<Your Connection String>,
    "[Customers].[Occupation].[All].[Manual]"
)
```

You can see a lot more of these formulas if you go back to the original pivot table that you converted and click in the column and row headings. If you write a `CUBEMEMBER()` formula as a standalone formula in a cell, you can reference that cell from within your `CUBEVALUE()` formula by using cell references. Once again, you can see this by examining the formula in your converted pivot table.

20: Transferring Your Skills to Excel

Power BI is a relatively new product from Microsoft. It first became generally available in July 2015, and the pace of change over the few years since its release has been phenomenal. The fact that you have purchased this book and have now arrived at this point probably means that you already know this. But the truth is that both Power Pivot and Power Query are technologies that were first built (and continue to evolve) for Microsoft Excel. Microsoft didn't (and still doesn't) do a really great job at marketing the existence of these products as part of Excel, and as a result, many (most?) people who could benefit from the technologies inside Excel don't know they exist. The good news, however, is that the skills you have learnt in this book are completely transferable to Power Pivot for Microsoft Excel. This chapter is here to help you transfer those skills with a minimum of pain.

Differences Between Power BI and Power Pivot for Excel

There are a couple of differences between Power BI and the various versions of Power Pivot for Excel that you should be aware of as explained below.

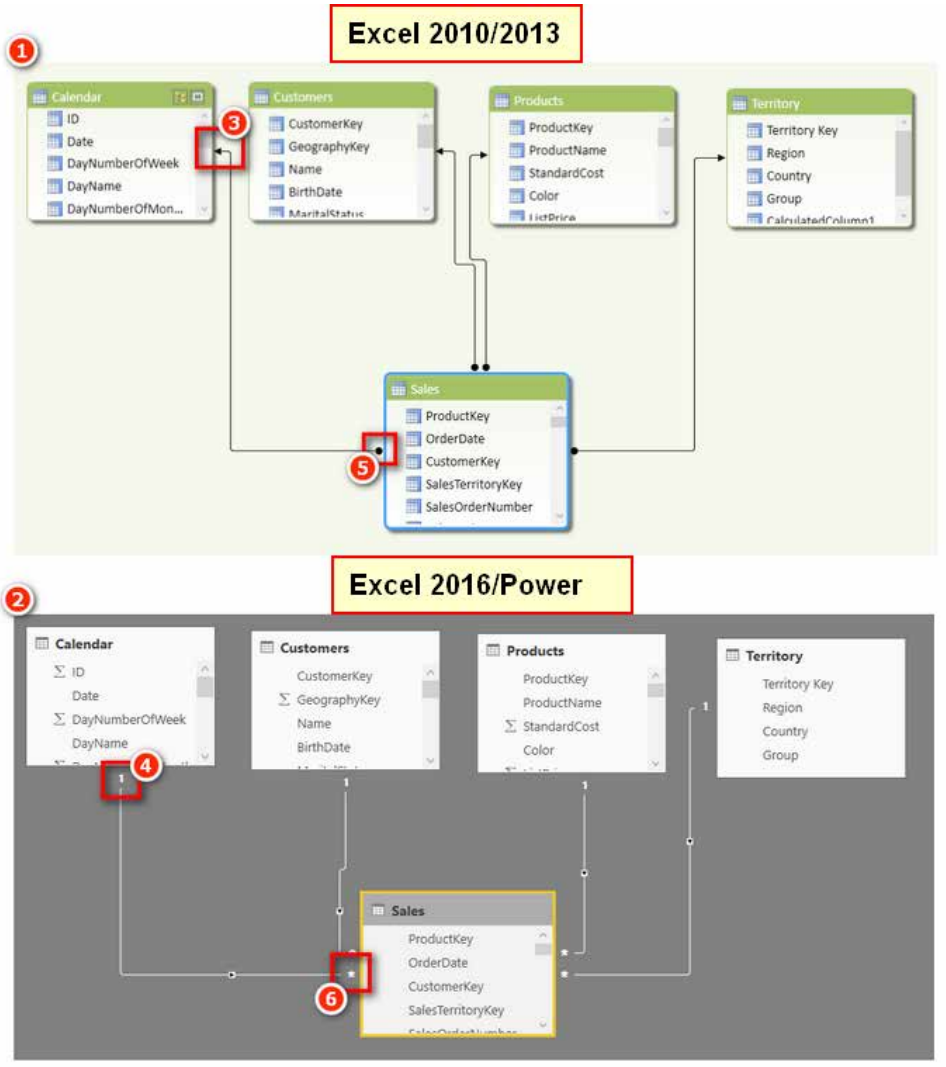
Note: I use the following conventions below:

- *Power BI* means Power BI Desktop
- *Power Pivot* is the data modelling engine that exists in both Power Pivot for Excel and Power BI Desktop.
- *Power Query* is the data acquisition tool that exists in both Excel and Power BI Desktop.

All Versions of Excel

- Each version of Excel has a unique version of Power Pivot, and not all functions are available in all versions. Power BI contains the latest and most up-to-date version of Power Pivot. I keep an up-to-date quick guide to all the functions in Power Pivot that you can download from http://xbi.com.au/quickguide.
- All Excel versions of Power Pivot support relationships of the type one-to-many; they do not support one-to-one like in Power BI. This is not a major issue as it is not a common relationship type.
- There is no bidirectional cross-filtering available in any of the Excel versions of Power Pivot.
- You can convert the data models produced in Excel 2010 to Excel 2013 or 2016 data models (upgrade), but you can't go back the other way (downgrade).
- Excel 2013/2016 data models can be opened in both 2013 and 2016.

- Excel 2010 and 2013 have a different user experience in Diagram view compared to Excel 2016 and Power BI Desktop. You can see the differences between the different UIs in the image below. The earlier versions of Excel (see #1 below) have an arrow pointing to the "one" side of the relationship (#3) and a black dot on the "many" side (#5). Excel 2016 and Power BI have the new, improved UI (#2) with a 1 on the "one" side of the relationship (#4) and an * on the "many" side (#6).

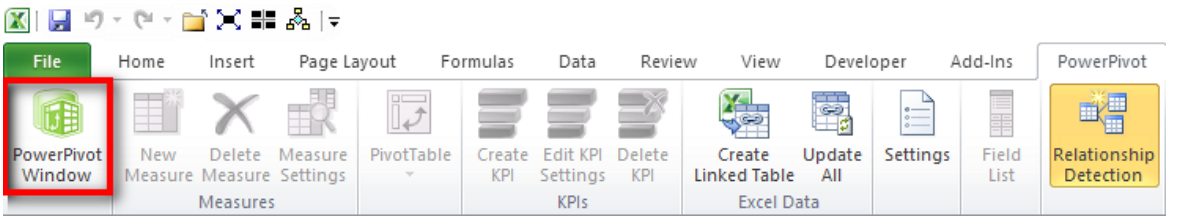


Excel 2013

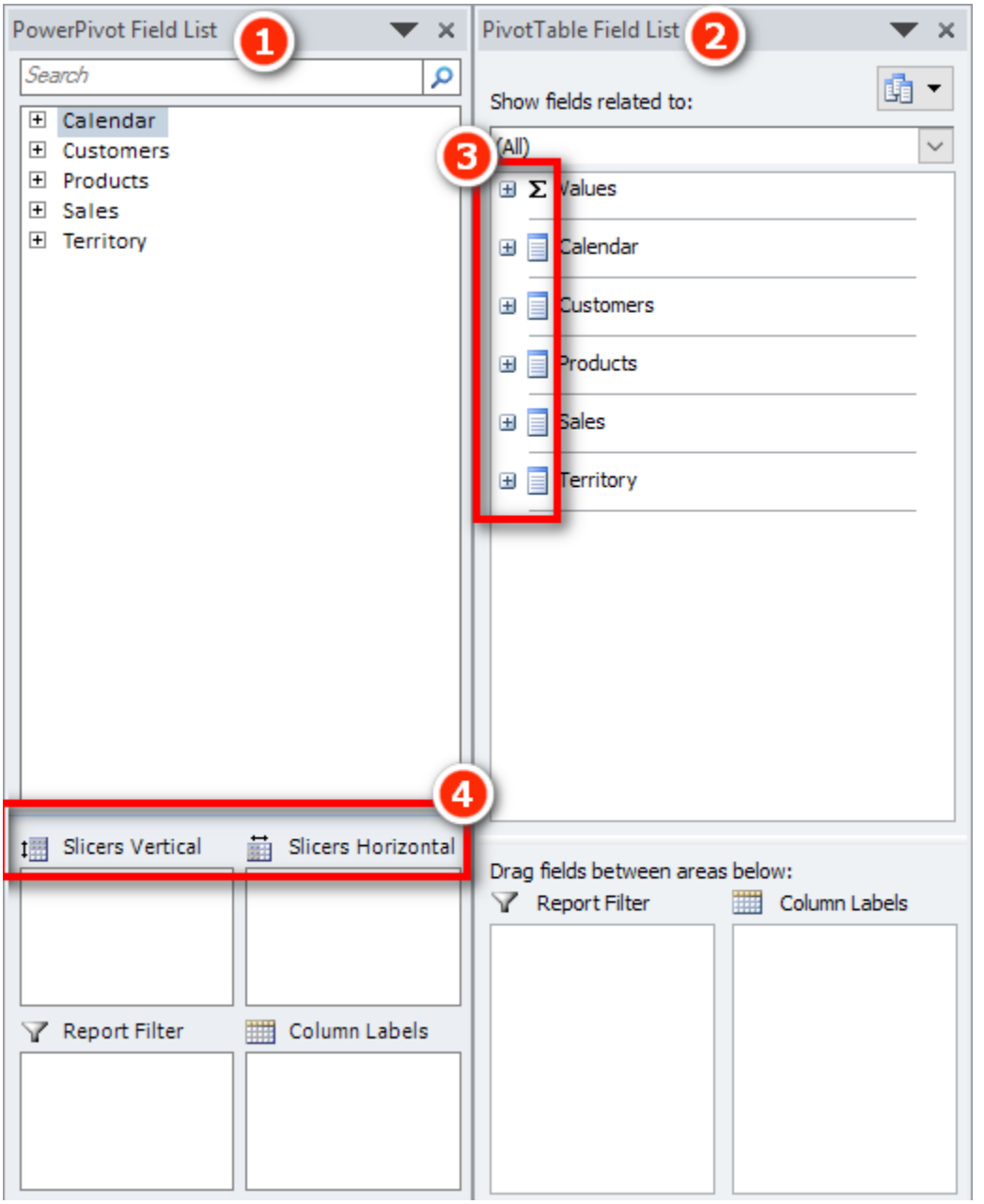
- In Excel 2013, the term *calculated fields* is used instead of *measures*. This was an unfortunate change made in Excel 2013 that lasted for this one version of Excel only, fortunately.

Excel 2010

- The ribbon in Excel 2010 is slightly different in that it has a Power Pivot Window option (as shown below) instead of Manage in the later versions of Excel. The Power Pivot icon is the same in all Excel versions.



- Excel 2010 is the only version that uses a completely separate add-in to deploy Power Pivot, and this means there are two completely different field lists for pivot tables. The first time you stumble on the traditional PivotTable field list, you may be confused about what is happening, so being forewarned is forearmed. In the image below, note the different titles in the two different field lists. The one on the left (see #1 below) is the newer PowerPivot Field list; this is the one you should be using when working with Power Pivot. The one on the right (#2) is the original field list, and you don't use it with Power Pivot pivot tables. But it is possible to have both open at the same time, and what is more confusing is that you can accidently open the wrong one without realising it. The key visual clues as to which one you have open are easy to spot: The titles are different, the PivotTable field list has special icons (#3), and the PowerPivot field list has a slicer drop zone (#4).



You Need Office Professional Plus

Not all versions of Excel come with Power Pivot as part of the offering. If you have the Home Edition or one of the many other lower-priced versions, you may be out of luck. And if you don't have Power Pivot with your version of Excel, it is not something you can just upgrade to; you need to purchase a different version of Excel. If you have an Office 365 subscription, this will not be a major issue, but if you have a one-time purchase product, you will face a new expense to get a new version of Excel with Power Pivot.

Excel 2010

For Excel 2010 you can download the free Power Pivot add-in from the Microsoft website. You can search using a browser for the download. Just make sure you find and install Service Pack 2. You should search for `Power Pivot for Excel 2010 SP2.`

Excel 2013/2016

For these newer versions of Excel, you need to purchase Microsoft Office ProPlus to be able to get the Power Pivot add-in. ProPlus is used by most large organisations that purchase an E3 licence. You can see a full list of the versions that have Power Pivot (and the ones that don't) at http://xbi.com.au/versions.

Power Query

Excel 2010/2013

Power Query is a free add-in that you can download from Microsoft. Just search for Power Query in a web browser. After installing you will have a new tab called Power Query.

Excel 2016

Power Query comes bundled with Excel 2016. You can find it on the Data tab, Get and Transform.

Migrating Data from Power BI to Excel

If you read this heading and got excited, then I am sorry to tell you that you cannot migrate a Power BI Desktop data model into Power Pivot for Excel. It is possible to migrate the other way, though (from Excel to Power BI), as shown below.

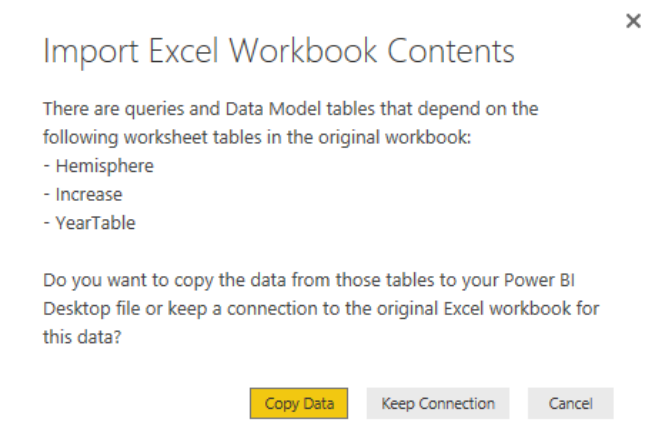
Don't forget that it is possible to use Analyze in Excel to create a new Excel workbook that points to a Power BI workbook loaded to PowerBI.com. This feature does require a Power BI Pro licence, however.

Here's How: Importing Excel Power Pivot Workbooks to Power BI Desktop

It is possible to import a Power Pivot data model from an Excel workbook into Power BI Desktop, along with all the data connections, relationships, and measures. Unfortunately, any reports you have created in Excel will not be migrated and will need to be re-created in Power BI.

Follow these steps to import a Power Pivot workbook from Excel into Power BI Desktop:

1. In Power BI Desktop, select File, New. This will open a new blank Power BI Desktop file.
2. Select File, Import, Excel Workbook Contents.
3. Navigate to the Excel workbook you want to migrate, select it, click OK, and then select Import.
4. When you get the choice to copy the data from your queries or keep the connection (shown below), click Keep Connection.



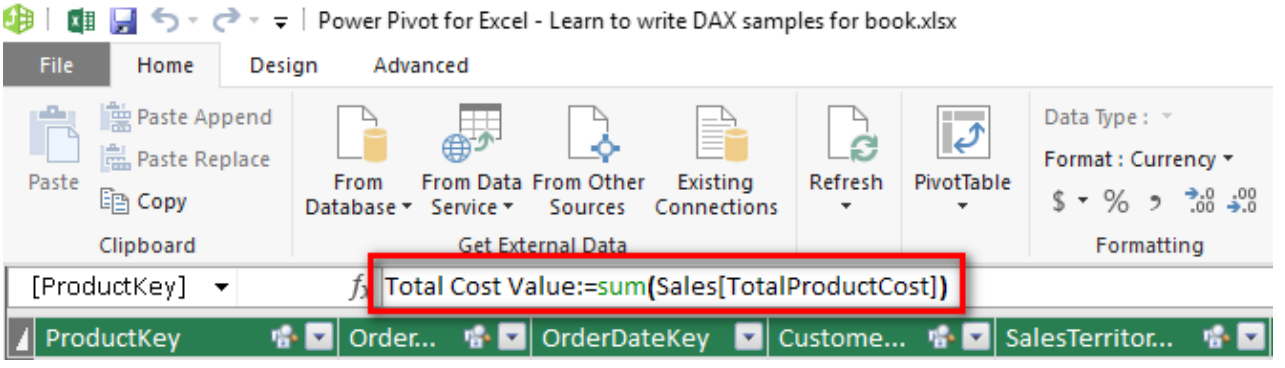
Note: Power BI Desktop doesn't include the concept of linked tables. When you import an Excel workbook that contains linked tables into Power BI Desktop, you can either bring the data in as a one-off migration or retain the link to the original linked table in the original Excel workbook.

The Excel workbook data model is then imported into Power BI Desktop. From there you can proceed to use Power BI Desktop instead of Excel and build your own visualisations on top of the Power Pivot data model.

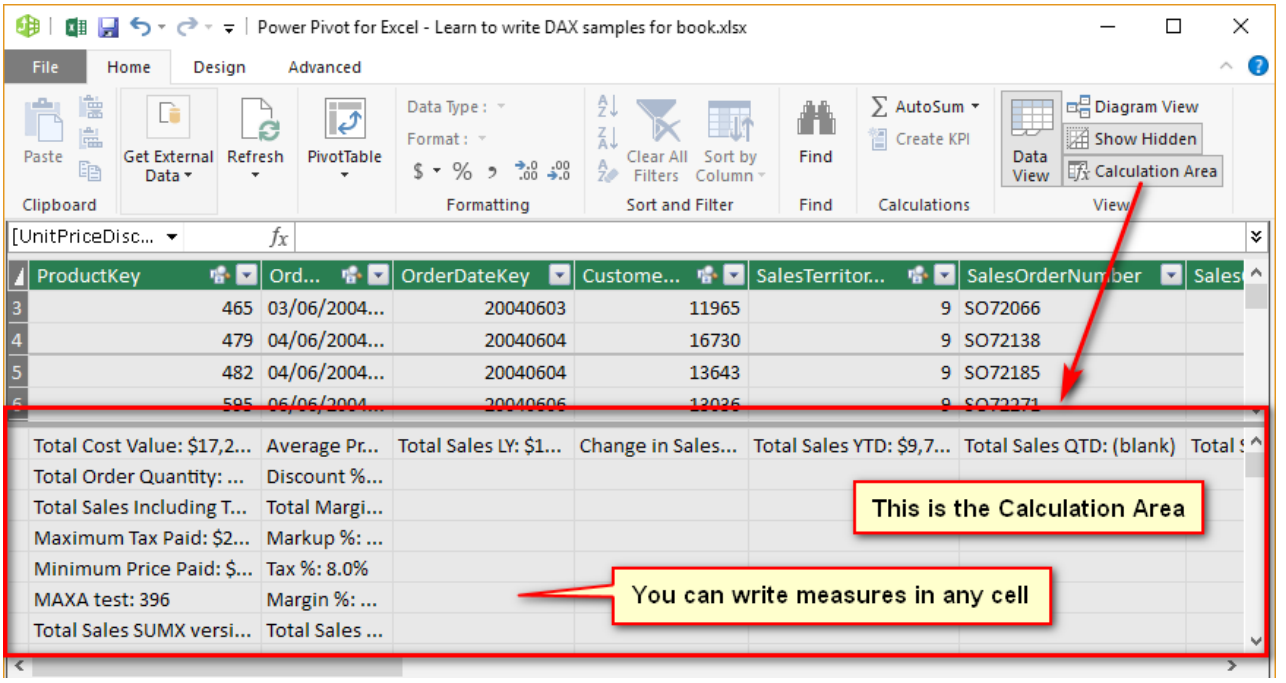
Writing DAX Measures in Excel

There are three places you can write DAX measures in Excel:

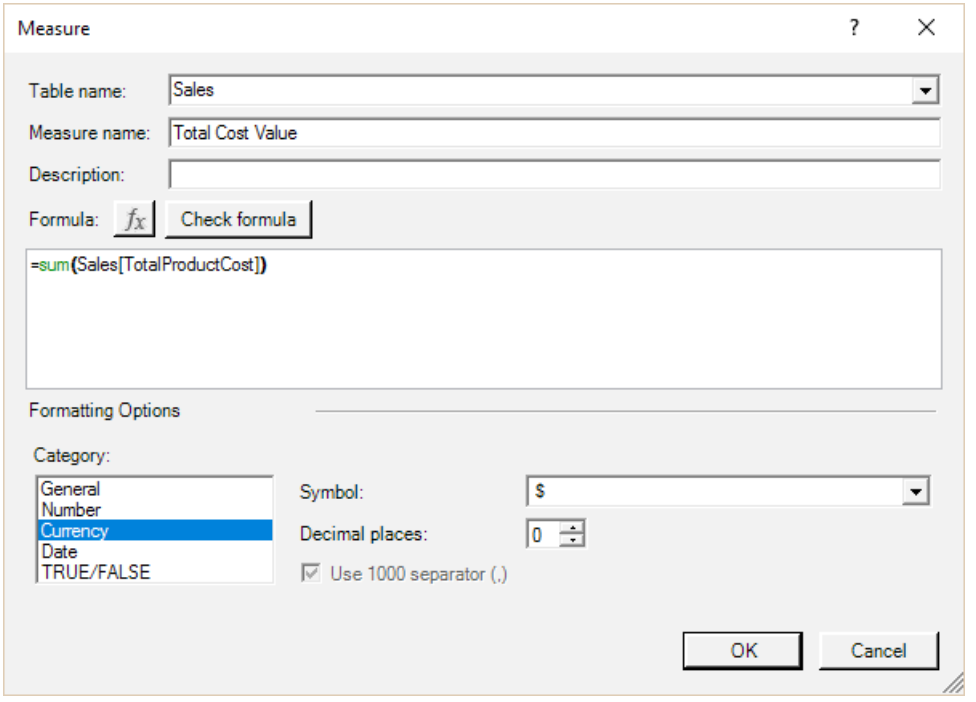
- You can write a measure in the formula bar in the Power Pivot window, as shown below. If you use this method, you must specify the measure name followed by a colon and then the formula.



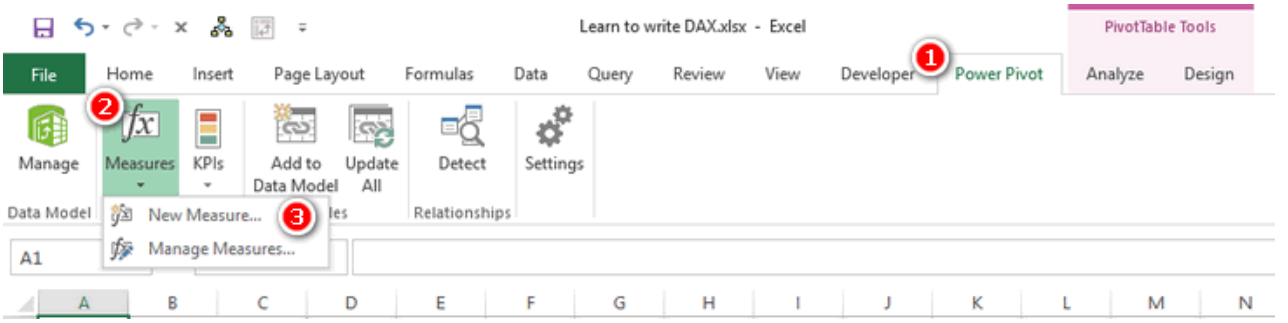
- You can write and edit measures in any empty cell in the calculation area at the bottom of the Power Pivot window, as shown below. You also need to add a colon when writing a measure here.



You can write measures in the Measure dialog in Excel, as shown below.



- You open this dialog in Excel by navigating to the Power Pivot tab (see #1 below) and selecting Measures (#2), New Measure (#3) or by right-clicking the Table name in the Pivot Table Fields list and choosing Add Measure….



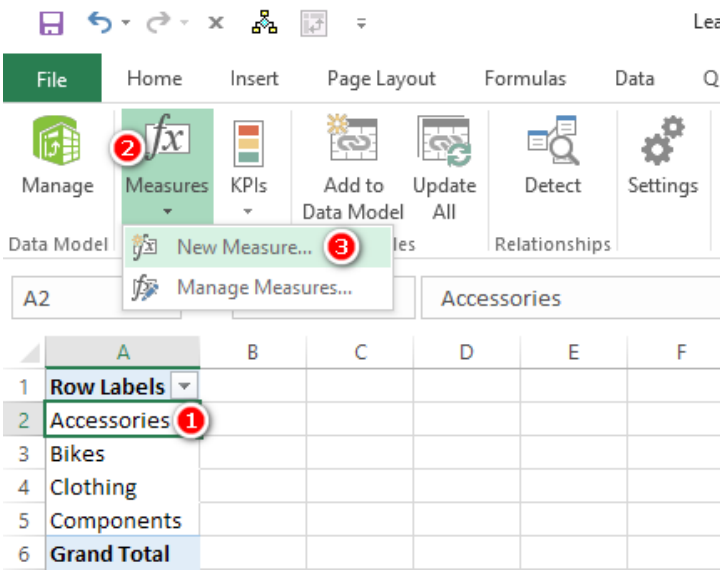
Tip: In general, I recommend that Excel users write DAX in the Measure dialog box in Excel. I also recommend to first create a pivot table that provides some context for the measure you are about to write. If you do it this way, you will immediately see the measure appear in the pivot table once you click OK, and this will give you immediate feedback about whether the formula looks correct.

Note: At this writing, some versions of Excel 2016 do not automatically add a new measure to a pivot table as described in the tip above. However, Microsoft plans to reverse this change in a future update. Some older versions of Excel 2016 may therefore not automatically add the measure.

Here's How: Writing Measures

To create a new measure in Excel, follow these steps:

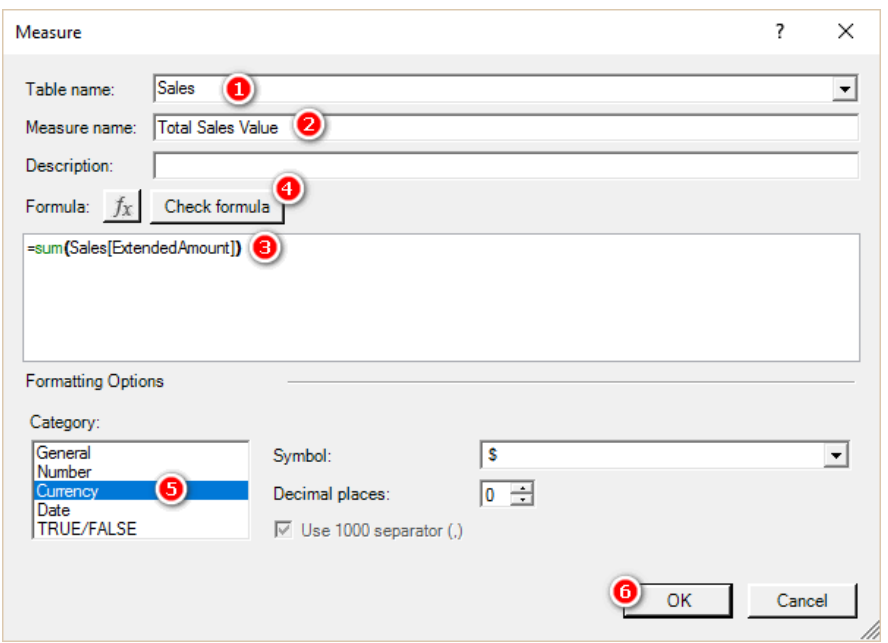
1. Create a new blank pivot table connected to your data model (or use an existing one if you already have something appropriate).
2. Add some relevant data to the rows in your pivot table (see #1 below).
3. Click inside the pivot table, navigate to the Power Pivot tab, click the Measures drop-down arrow (#2), and then select New Measure (#3). The Measure dialog appears.



Tip: You should use the Measure dialog shown below as a process flow/guide. If you don't do this, you risk missing one or more of the steps. Missing a step will end up costing you time and causing rework. Get in the good habit of following the process steps I describe here, using the dialog to remind you of all the steps. Always follow the order outlined here.

4. In the Table Name drop-down (see #1 below), select the table where your measure will be stored.
5. In the Measure Name text box (#2), give the new measure a name.
6. In the Formula box (#3) write the DAX formula.
7. Click Check Formula (#4) to check whether the formula you wrote is syntactically correct. Fix any errors if you need to.
8. Select an appropriate formatting option from the Category list (#5), including a suitable symbol and decimal places in the area to the right of the Category list.
9. Click OK (#6) to save your measure.

Note: I generally don't enter anything in the Description box, but it is there for you to use if you like. It's for reference only and doesn't impact the behaviour of the formulas.



21: Next Steps on Your DAX Journey

You have almost finished reading all the chapters of this book. Now what? First of all, let me assure you that this is just the start, not the end, of your journey to learning how to supercharge Power BI (and now Excel) by learning to write DAX. As I have been saying all through the book, the most important thing is to practice, practice, practice. Start using your new skills at work and at play so that you build your depth of skill and knowledge. It will take you many months of using your new skills before you become an expert, but you are well on your way already. Now that you have a basic understanding of Power BI and DAX, you can incrementally learn and improve over time. But there are some things that will help you learn more and faster.

Three Person Teaching/Learning

I am a big believer in "three-person teaching/learning." I first heard of this concept from Stephen Covey, at one of his seminars. The basic idea is that you learn more when you learn with the intent to teach others, and you learn more from the process of teaching others. For this reason, I really believe in the benefits of participating in user forums. As I mentioned at the start of the book, I have set up a forum at http://powerpivotforum.com.au, and it is free for anyone and everyone to ask questions and also to help others. If you want to really cement your new skills and knowledge, then sign up and ask for help, and, more importantly, answer questions and teach others on the forum. When you teach others, you cement your knowledge and become better and stronger with your DAX. There is also an excellent Power BI community and forum at http://community.powerbi.com. You can participate in the forums and maybe also join a local Power BI user group in your region.

Blogs

There are a number of Power Pivot blogs that I recommend you subscribe to. Reading blogs is a great way to keep in touch with the latest thinking from people who spend their life working with Power Pivot. Here are some that I think are especially useful:

- My blog: http://xbi.com.au/blog
- Rob Collie's blog: http://powerpivotpro.com
- Marco Russo and Alberto Ferrari's blog: http://sqlbi.com
- Reza Rad's blog: http://radacad.com/blog
- Gilbert Quevauvilliers' blog: https://www.fourmoo.com

Books

There are a few really good DAX books that I recommend (and have mentioned previously). I keep a list of books I recommend on my website and update it over time. You can always find an updated list at http://xbi.com.au/books.

Online Training

I offer online training for Power Pivot and also Power Query. You can find out more from these links:

- Supercharge Power BI: http://xbi.com.au/scpbitraining
- Power Query: http://xbi.com.au/powerquerytraining

Live Training

Some people learn best in a classroom environment. If you're one of them, you might want to attend a live training event in a location suitable for you. I personally deliver live training courses in Australia. For details about upcoming events, see http://xbi.com.au/training. I also deliver customised in-house training for companies that have larger groups of users and are looking for a more personalised experience.

If you are in the United States, I recommend that you take a look at Rob Collie's live offerings. For information, see http://powerpivotpro.com.

There are many great things about these training courses, but one super benefit is that both Rob and I teach Power Pivot using the same techniques I have used in this book. By attending one of our live training courses, you will continue to learn using the same methodology you have used in this book.

Power Query

Power Query is a desktop ETL (extract, transform, and load) tool for Excel users. It is the same technology that comes bundled with Power BI. Power Query allows you to connect to data from anywhere, change the shape of that data, and then load it into your workbooks. Once the data is loaded with Power Query, you can easily refresh the link at any time and bring in the latest updated data.

I often blog about Power Query at http://xbi.com.au/blog, and you can also get great information from these websites and books:

- Ken Puls' blog: http://www.excelguru.ca/blog/
- Chris Webb's blog: http://blog.crossjoin.co.uk/
- Gil Raviv's blog: https://datachant.com
- Chris Webb's book *Power Query for Power BI and Excel*: http://xbi.com.au/ChrisWebbBook
- Ken Pul's book Master Your Data with Excel and Power BI: Leveraging Power Query to Get & Transform Your Task Flow: http://xbi.com.au/masteryourdata

That's All, Folks

I hope you have enjoyed this book and that it has successfully started you on your journey to becoming a DAX superstar. If you liked this book, please tell your Power BI and Excel friends and colleagues so they, too, can become DAX superstars.

Appendix A: Answers to Practice Exercises

This appendix provides the answers to the practice exercises scattered throughout the book. The answers are in the same order the exercises appear in the book and are numbered so you can easily match up the exercises and the answers.

SUM() These practice exercises appear in Chapter 4. As you compare your answers to the ones shown here, consider the following questions: Did you remember to put your measures in the correct table? Did you put the measure in the table where the data comes from? Did you format with an appropriate number format?

```
1. Total Sales
= SUM(Sales[ExtendedAmount])
Or:
Total Sales
= SUM(Sales[SalesAmount])
2. Total Cost
= SUM(Sales[TotalProductCost])
Or:
Total Cost
= SUM(Sales[ProductStandardCost])
3. Total Margin $
= [Total Sales] – [Total Cost]
4. Total Margin % = [Total Margin $] / [Total Sales]
Or:
Total Margin % = DIVIDE([Total Margin $] , [Total Sales])
5. Total Sales Tax Paid
= SUM(Sales[TaxAmt])
6. Total Sales Including Tax
= [Total Sales] + [Total Sales Tax Paid]
7. Total Order Quantity
= SUM(Sales[OrderQuantity])
```

COUNT() These practice exercises appear in Chapter 4.

```
8. Total Number of Products
= COUNT(Products[ProductKey])
9. Total Number of Customers
= COUNT(Customers[CustomerKey])
```

Note: Counting the "key" columns is generally pretty safe because, by definition, each one must have a value. Technically, you can count any column that has a numeric value in each cell, and you will get the same answer. Just be careful if you are counting a numeric column that may have blank values because COUNT()will not count blanks.

COUNTROWS() These practice exercises appear in Chapter 4.

Note: Remember that `COUNTROWS()` takes a table, not a column, as input.

```
10. Total Number of Products COUNTROWS Version
= COUNTROWS(Products)
11. Total Number of Customers COUNTROWS Version
= COUNTROWS(Customers)
```

DISTINCTCOUNT() These practice exercises appear in Chapter 4.

```
12. Total Customers in Database DISTINCTCOUNT Version
= DISTINCTCOUNT(Customers[CustomerKey])
13. Count of Occupation
= DISTINCTCOUNT(Customers[Occupation])
14. Count of Country
```

```
= DISTINCTCOUNT(Territories[Country])
15. Total Customers That Have Purchased
= DISTINCTCOUNT(Sales[CustomerKey])
```

MAX(), MIN(), and AVERAGE() These practice exercises appear in Chapter 4.

```
16. Maximum Tax Paid on a Product
= MAX(Sales[TaxAmt])
17. Minimum Price Paid for a Product
= MIN(Sales[ExtendedAmount])
18. Average Price Paid for a Product
= AVERAGE(Sales[ExtendedAmount])
```

COUNTBLANK() These practice exercises appear in Chapter 4.

```
19. Customers Without Address Line 2
= COUNTBLANK(Customers[AddressLine2])
20. Products Without Weight Values
= COUNTBLANK(Products[Weight])
```

DIVIDE() These practice exercises appear in Chapter 4.

```
21. Margin %
= DIVIDE([Total Margin $] , [Total Sales])
22. Markup %
= DIVIDE([Total Margin $] , [Total Cost])
23. Tax %
= DIVIDE([Total Sales Tax Paid] , [Total Sales])
```

SUMX() These practice exercises appear in Chapter 7.

```
24. Total Sales SUMX Version
= SUMX(Sales, Sales[OrderQuantity] * Sales[UnitPrice])
```

Note: In this sample database, the order quantity is always 1.

```
25. Total Sales Including Tax SUMX Version
= SUMX(Sales,Sales[ExtendedAmount] + Sales[TaxAmt])
26. Total Sales Including Freight
= SUMX(Sales,Sales[ExtendedAmount] + Sales[Freight])
27. Dealer Margin
= SUMX (Products,Products[ListPrice] - Products[DealerPrice])
```

AVERAGEX() These practice exercises appear in Chapter 7.

```
28. Average Sell Price per Item
= AVERAGEX(Sales, Sales[UnitPrice])
```

Note: The expression can be a single column. It doesn't have to be an equation using multiple columns.

```
Or: Average Sell Price per Item Weighted
= AVERAGEX(Sales, Sales[OrderQuantity] * Sales[UnitPrice])
```

In fact, this sample database always has `Sales[OrderQuantity] = 1`, so the answer will be the same as the previous formula.

```
29. Average Tax Paid = AVERAGEX(Sales, Sales[TaxAmt])
30. Average Safety Stock
= AVERAGEX(Products, Products[SafetyStockLevel])
```

Calculated Columns This practice exercise appears in Chapter 8.

```
31. = IF(
     OR('Calendar'[CalendarQuarter]=1,
       'Calendar'[CalendarQuarter]=2
     ),
   "H1","H2"
```

```
    )
```

Note: There are a number of ways to write this calculated column. If yours is different from this but works, then all is well and good.

CALCULATE() with a Single Table These practice exercises appear in Chapter 9.

```
32. Total Male Customers
= CALCULATE([Total Number of Customers],
     Customers[Gender] = "M")
33. Total Customers Born Before 1950
= CALCULATE([Total Number of Customers],
     Customers[BirthDate] < DATE(1950,1,1))
34. Total Customers Born in January
= CALCULATE([Total Number of Customers],
     MONTH(Customers[BirthDate])=1)
35. Customers Earning at Least $100,000 per Year
= CALCULATE([Total Number of Customers],
     Customers[YearlyIncome]>=100000)
```

CALCULATE() with Multiple Tables These practice exercises appear in Chapter 9.

```
36. Total Sales of Clothing
= CALCULATE([Total Sales],
     Products[Category]="Clothing")
37. Sales to Female Customers
= CALCULATE([Total Sales],
      Customers[Gender]="F")
38. Sales of Bikes to Married Men
= CALCULATE([Total Sales],
     Customers[MaritalStatus]="M",
     Customers[Gender]="M",
     Products[Category]="Bikes"
  )
```

VALUES() These practice exercises appear in Chapter 12.

```
39. Number of Color Variants
= COUNTROWS(VALUES(Products[Color]))
40. Number of Sub Categories
= COUNTROWS(VALUES(Products[SubCategory]))
41. Number of Size Ranges
= COUNTROWS(VALUES(Products[SizeRange]))
42. Product Category (Values)
= IF(HASONEVALUE(Products[Category]),
    VALUES(Products[Category])
  )
Or:
= SELECTEDVALUE(Products[Category])
43. Product Subcategory (Values)
= IF(HASONEVALUE(Products[SubCategory]),
     VALUES(Products[SubCategory])
  )
Or: = SELECTEDVALUE(Products[SubCategory])
44. Product Color (Values)
= IF(HASONEVALUE(Products[color]),
    VALUES(Products[color])
  )
Or: = SELECTEDVALUE(Products[color])
```

```
45. Product Subcategory (Values) edited
= IF(HASONEVALUE(Products[SubCategory]),
     VALUES(Products[SubCategory]),
     "More than 1 Sub Cat"
  )
Or: = SELECTEDVALUE(Products[SubCategory]),"More than 1 Sub Cat")
46. Product Color (Values) edited
= IF(HASONEVALUE(Products[color]),
     VALUES(Products[color]),
     "More than 1 Color"
  )
Or: = SELECTEDVALUE(VALUES(Products[color]),"More than 1 Color")
```

ALL(), ALLEXCEPT(), and ALLSELECTED() These practice exercises appear in Chapter 13.

```
47. Total Sales to All Customers
= CALCULATE([Total Sales] , All(Customers))
```

Note: This measure belongs in the Sales table, not the Customers table.

```
48. % of All Customer Sales
= DIVIDE([Total Sales] , [Total Sales to All Customers])
49. Total Sales to Selected Customers
= CALCULATE([Total Sales] , ALLSELECTED(Customers))
50. % of Sales to Selected Customers
= DIVIDE([Total Sales] , [Total Sales to Selected Customers])
51. Total Sales for All Days Selected Dates
= CALCULATE([Total Sales] , ALLSELECTED('Calendar'))
```

Note: Did you know to use ALLSELECTED() and not ALLEXCEPT()?

```
52. % Sales for All Days Selected Dates
  = DIVIDE([Total Sales] ,[Total Sales for All Days Selected Dates])
53. Total Orders All Customers
= CALCULATE([Total Order Quantity] , ALL(Customers))
54. Baseline Orders for All Customers with This Occupation
= CALCULATE([Total Order Quantity] ,
    ALLEXCEPT(Customers, Customers[Occupation])
  )
55. Baseline % This Occupation of All Customer Orders
= DIVIDE(
   [Baseline Orders for All customers with this Occupation] ,
   [Total Orders All Customers]
  )
56. Total Orders Selected Customers
= CALCULATE([Total Order Quantity] , ALLSELECTED(Customers])
57. Occupation % of Selected Customers
= DIVIDE(
    [Total Order Quantity] ,
    [Total Orders Selected Customers]
  )
58. Percentage Point Variation to Baseline
= [Occupation % of Selected Customers] -
[Baseline % this Occupation is of All Customer Orders]
```

FILTER() These practice exercises appear in Chapter 14.

```
59. Total Sales of Products That Have Some Sales but Less Than $10,000
= CALCULATE([Total Sales],
```

```
    FILTER(Products,
      [Total Sales] < 10000 &&
      [Total Sales] >0
    )
  )
Or: = CALCULATE([Total Sales],
     FILTER(Products, [Total Sales] <10000),
     FILTER(Products, [Total Sales] >0)
  )
60. Count of Products That Have Some Sales but Less Than $10,000
= CALCULATE(COUNTROWS(Products),
     FILTER(Products,
       [Total Sales]<10000 && [Total Sales] >0
     )
  )
Or: = CALCULATE(COUNTROWS(Products),
     FILTER(Products, [Total Sales] <10000),
     FILTER(Products, [Total Sales] >0)
  )
```

Time Intelligence These practice exercises appear in Chapter 15.

```
61. Total Sales Month to Date
= TOTALMTD([Total Sales], 'Calendar'[Date])
62. Total Sales Quarter to Date
= TOTALQTD([Total Sales], 'Calendar'[Date])
63. Total Sales FYTD 30 June
= TOTALYTD([Total Sales],'Calendar'[Date],"30/6")
64. Total Sales FYTD 31 March
= TOTALYTD([Total Sales],'Calendar'[Date],"31/3")
65. Total Sales Previous Month
= CALCULATE([Total Sales],
    PREVIOUSMONTH(Calendar[Date])
  )
66. Total Sales Previous Day
= CALCULATE([Total Sales],
     PREVIOUSDAY('Calendar'[Date])
  )
67. Total Sales Previous Quarter
= CALCULATE([Total Sales],
     PREVIOUSQUARTER('Calendar'[Date])
  )
68. Total Sales Moving Annual Total
= CALCULATE([Total Sales],
    FILTER(ALL('Calendar'),
       'Calendar'[ID] > MAX('Calendar'[ID]) - 365 &&
       'Calendar'[ID] <= MAX('Calendar'[ID])
    )
  )
69. Total Sales Rolling 90 Days
= IF(MAX('Calendar'[ID])>=90,
     CALCULATE([Total Sales],
        FILTER(ALL('Calendar'),
          'Calendar'[ID] > MAX('Calendar'[ID]) - 90 &&
          'Calendar'[ID] <= MAX('Calendar'[ID])
        )
     )
```

```
  )
```

Harvester Measures This practice exercise appears in Chapter 17.

70.
```
Total Customers Born Before Selected Year
= CALCULATE ( [Total number of Customers],
    FILTER (Customers,
      Customers[BirthDate] < DATE ( [Selected Year], 1, 1 )
    )
  )
```

Multiple Data Tables These practice exercises appear in Chapter 18.

71.
```
Total Budget = SUM(Budget[Budget])
```
This measure should be placed in the Budget table.

72.
```
Change in Sales vs. Budget
= [Total Sales] - [Total Budget]
```
This measure could be placed in either the Sales table or the Budget table. I normally place it in the Sales table because the name of the measure is [Change in Sales vs. Budget].

73.
```
% Change in Sales vs. Budget
= DIVIDE([Change in Sales vs. Budget] , [Total Budget])
```
Also place this measure in the Sales table.

Table of Here's How Sections

Index

Symbols

A

B

C

D

E

F

G

H

I

J

Need More Help?

Supercharge Power BI Online

Live with Matt Allington

- 5 video lessons that support the chapters in this book
- Live Q&A sessions each week with Matt Allington

Find Out More at

http://xbi.com.au/scpbi

Ready to Learn More?

PQ

Power Query Online Training

- Over 7 hours of video lessons
- Free sample videos
- Lessons taught using real world examples
- Sample files available for you to practice

Now that you know how to write DAX, it's time to learn how to cleanse and load data using Power Query.

Developed by Microsoft® MVP

Matt Allington

Find Out More at

http://xbi.com.au/pqtb

Inside Back Cover - This page is blank

Shelving Category: Spreadsheets
Reader Level: Intermediate

Practical, hands-on lessons for mastering the DAX language in Power BI and Excel

Microsoft Power BI is a self-service business intelligence tool. Anyone can get started with Power BI by downloading Power BI Desktop, loading up some data, and building a report. But with this basic approach, you can only scratch the surface of Power BI's capabilities. If you want to be able to supercharge Power BI, you need to learn to write DAX. Data Analysis Expressions (DAX) is the formula language of Power BI and Power Pivot for Excel. With the help of a good book that prompts readers to put their new skills to the test, Excel users can rather quickly learn the required DAX skills. This is such a book, written to give you hands-on practice using Power BI Desktop and writing DAX. Inside you will find explanations of concepts, examples, and practice exercises and answers to maximize learning retention and experience.

Supercharge Excel

This is a sister book to *Supercharge Excel 2nd Edition*. These two books use the same content, teaching, and practice format but with different software versions, as indicated in the titles. When you read and complete the exercises in this book, Supercharge Power BI, you will be learning how to supercharge Excel at the same time. The DAX language is fully transferable to Microsoft Excel.

Holy Macro! Books
PO Box 541731
Merritt Island, FL 32953
$29.95 US | $32.95 CAN

MVP Microsoft® Most Valuable Professional

Matt Allington is a career data professional who has worked in the retail and consumer packaged goods industries for more than 35 years.

Matt is a Microsoft MVP and specialises in Power BI, Power Pivot, and Power Query.

ISBN 978-1-61547-052-5
US$29.95
52995
9 781615 470525